

Федеральное государственное унитарное предприятие
Российский федеральный ядерный центр
Всероссийский научно-исследовательский институт экспериментальной физики

УТВЕРЖДЕН
07623615.00082-02 34 01-ЛУ

КОМПЛЕКС ПРОГРАММ В ЗАЩИЩЕННОМ ИСПОЛНЕНИИ
«СИСТЕМА ПОЛНОГО ЖИЗНЕННОГО ЦИКЛА ИЗДЕЛИЙ
«ЦИФРОВОЕ ПРЕДПРИЯТИЕ»

**Программный модуль
«Система управления базами данных «Синергия-БД»**

Руководство оператора

Часть 1

07623615.00082-02 34 01-1

Листов 261

Име. № подл.	Подп. и дата	Взам. инв. №	Име. № дубл.	Подп. и дата

СОДЕРЖАНИЕ

1.	Введение.....	5
2.	Синтаксис SQL	6
2.1.	Лексическая структура.....	6
2.2.	Выражения значения.....	20
2.3.	Вызов функций	42
3.	Определение данных.....	46
3.1.	Основы таблиц.....	46
3.2.	Значения по умолчанию.....	48
3.3.	Ограничения	50
3.4.	Системные колонки.....	62
3.5.	Изменение таблиц	64
3.6.	Права.....	68
3.7.	Схемы	69
3.8.	Наследование	77
3.9.	Секционирование таблиц	83
3.10.	Сторонние данные.....	108
3.11.	Другие объекты баз данных	108
3.12.	Отслеживание зависимостей.....	109
4.	Модификация данных	112
4.1.	Добавление данных.....	112
4.2.	Изменение данных	114
4.3.	Удаление данных	115
4.4.	Возврат данных из изменённых строк	116
5.	Запросы.....	118
5.1.	Обзор.....	118
5.2.	Табличные выражения	119
5.3.	Списки выборки	141
5.4.	Сочетание запросов.....	144
5.5.	Сортировка строк	145

5.6.	LIMIT и OFFSET	146
5.7.	Списки VALUES	147
5.8.	Запросы WITH (Общие табличные выражения)	149
6.	Типы данных	158
6.1.	Числовые типы	159
6.2.	Денежные типы	166
6.3.	Символьные типы.....	167
6.4.	Двоичные типы данных	170
6.5.	Типы даты/времени	173
6.6.	Логический тип.....	187
6.7.	Типы перечислений.....	188
6.8.	Геометрические типы.....	191
6.9.	Типы, описывающие сетевые адреса	194
6.10.	Битовые строки.....	198
6.11.	Типы, предназначенные для текстового поиска.....	199
6.12.	Тип UUID	203
6.13.	Тип XML	204
6.14.	Типы JSON	208
6.15.	Трансформации.....	219
6.16.	Массивы	219
6.17.	Составные типы.....	235
6.18.	Диапазонные типы	244
6.19.	Типы доменов	254
6.20.	Идентификаторы объектов	254
6.21.	Тип pg_lsn.....	257
6.22.	Псевдотипы	257
	Перечень сокращений	259
	07623615.00082-02 34 01-2 Руководство оператора. Часть 2	
	07623615.00082-02 34 01-3 Руководство оператора. Часть 3	
	07623615.00082-02 34 01-4 Руководство оператора. Часть 4	

07623615.00082-02 34 01-5 Руководство оператора. Часть 5

1. ВВЕДЕНИЕ

В настоящем документе описывается использование языка SQL в СУБД «Синергия-БД» (далее Синергия-БД). Так как Синергия-БД основана на СУБД PostgreSQL, то реализация SQL в этих базах данных совпадает.

Сначала приводится описание общего синтаксиса SQL, затем создание структуры для хранения данных, наполнение базы данных и выполнение запросов к ней. Далее перечисляются существующие типы данных и функции, применяемые с командами SQL.

В следующих частях приводится формальное и полное описание всех команд SQL и клиентских приложений.

2. СИНТАКСИС SQL

2.1. Лексическая структура

SQL-программа состоит из последовательности команд. Команда, в свою очередь, представляет собой последовательность компонентов, оканчивающуюся точкой с запятой («;»). Конец входного потока также считается концом команды. Какие именно компоненты допустимы для конкретной команды, зависит от её синтаксиса.

Компонентом команды может быть ключевое слово, идентификатор, идентификатор в кавычках, строка (или константа) или специальный символ. Компоненты обычно разделяются пробельными символами (пробел, табуляция, перевод строки), но это не требуется, если нет неоднозначности (например, когда спецсимвол оказывается рядом с компонентом другого типа).

Например, следующий текст является правильной (синтаксически) SQL-программой:

```
SELECT * FROM MY_TABLE;  
UPDATE MY_TABLE SET A = 5;  
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

Это последовательность трёх команд, по одной в строке (хотя их можно было разместить и в одну строку или наоборот, разделить команды на несколько строк).

Кроме этого, SQL-программы могут содержать комментарии. Они не являются компонентами команд и равносильны пробельным символам.

Синтаксис SQL не очень строго определяет, какие компоненты идентифицируют команды, а какие – их операнды или параметры. Первые несколько компонентов обычно содержат имя команды, так что в данном примере мы можем говорить о командах «SELECT», «UPDATE» и «INSERT». Но например, команда UPDATE требует, чтобы также в определенном положении всегда стоял компонент SET, а INSERT в приведённом виде требует наличия компонента VALUES.

2.1.1. Идентификаторы и ключевые слова

Показанные выше команды содержали компоненты SELECT, UPDATE и

VALUES, которые являются примерами ключевых слов, то есть слов, имеющих фиксированное значение в языке SQL. Компоненты MY_TABLE и A являются примерами идентификаторов. Они идентифицируют имена таблиц, колонок или других объектов баз данных, в зависимости от того, где они используются. Поэтому иногда их называют просто «именами». Ключевые слова и идентификаторы имеют одинаковую лексическую структуру, то есть, не зная языка, нельзя определить, является ли некоторый компонент ключевым словом или идентификатором.

Идентификаторы и ключевые слова SQL должны начинаться с буквы (a-z, хотя допускаются также не латинские буквы и буквы с диакритическими знаками) или подчеркивания (_). Последующими символами в идентификаторе или ключевом слове могут быть буквы, цифры (0-9), знаки доллара (\$) или подчеркивания. Заметьте, что строго следуя букве стандарта SQL, знаки доллара нельзя использовать в идентификаторах, так что их использование вредит переносимости приложений. В стандарте SQL гарантированно не будет ключевых слов с цифрами и начинающихся или заканчивающихся подчеркиванием, так что идентификаторы такого вида защищены от возможных конфликтов с будущими расширениями стандарта.

Система выделяет для идентификатора не более NAMEDATALEN-1 байт, а более длинные имена усекаются. По умолчанию NAMEDATALEN равно 64, так что максимальная длина идентификатора равна 63 байтам. Если этого недостаточно, этот предел можно увеличить, изменив константу NAMEDATALEN в файле src/include/pg_config_manual.h.

Ключевые слова и идентификаторы без кавычек воспринимаются системой без учета регистра. Таким образом:

```
UPDATE MY_TABLE SET A = 5;
```

равносильно записи:

```
uPDATE my_TabLE SeT a = 5;
```

Часто используется неформальное соглашение записывать ключевые слова заглавными буквами, а имена строчными.

Есть и другой тип идентификаторов: отделенные идентификаторы или идентификаторы в кавычках. Они образуются при заключении обычного набора

символов в двойные кавычки («»). Такие идентификаторы всегда будут считаться идентификаторами, но не ключевыми словами. Так «select» можно использовать для обозначения колонки или таблицы «select», тогда как select без кавычек будет воспринят как ключевое слово и приведет к ошибке разбора команды в месте, где ожидается имя таблицы или колонки. Тот же пример можно переписать с идентификаторами в кавычках следующим образом:

```
UPDATE «my_table» SET «a» = 5;
```

Идентификаторы в кавычках могут содержать любые символы, за исключением символа с кодом 0. Чтобы включить в такой идентификатор кавычки, продублируйте их. Это позволяет создавать таблицы и колонки с именами, которые иначе были бы невозможны, например, с пробелами или амперсандами. Ограничение длины при этом сохраняется.

Ещё один вариант идентификаторов в кавычках позволяет использовать символы Unicode по их кодам. Такой идентификатор начинается с U& (строчная или заглавная U и амперсанд), а затем сразу без пробелов идет двойная кавычка, например U&»foo». Заметьте, что при этом возникает неоднозначность с оператором &. Чтобы её избежать, окружайте этот оператор пробелами. Затем в кавычках можно записывать символы Unicode двумя способами: обратная косая черта, а за ней код символа из четырех шестнадцатеричных цифр, либо обратная косая черта, знак плюс, а затем код из шести шестнадцатеричных цифр. Например, идентификатор «data» можно записать так:

```
U&»d\0061t\+000061»
```

В следующем менее тривиальном примере закодировано русское слово «слон», записанное кириллицей:

```
U&»\0441\043B\043E\043D»
```

Если вы хотите использовать не обратную косую черту, а другой спецсимвол, его можно указать, добавив UESCAPE после строки, например:

```
U&»d!0061t!+000061» UESCAPE '!'
```

В качестве спецсимвола можно выбрать любой символ, кроме шестнадцатеричной цифры, знака плюс, апострофа, кавычки или пробельного символа. Заметьте, что спецсимвол заключается не в двойные кавычки, а в

апострофы.

Чтобы сделать спецсимволом знак апострофа, напишите его дважды.

Unicode-формат полностью поддерживается только при использовании на сервере кодировки UTF8. Когда используются другие кодировки, допускается указание только ASCII-символов (с кодами до \007F). И в четырех, и в шестизначной форме можно записывать суррогатные пары UTF-16 и таким образом составлять символы с кодами больше чем U+FFFF, хотя наличие шестизначной формы технически делает это ненужным. Суррогатные пары не сохраняются непосредственно, а объединяются в один символ, который затем кодируется в UTF-8.

Идентификатор, заключенный в кавычки, становится зависимым от регистра, тогда как идентификаторы без кавычек всегда переводятся в нижний регистр. Например, идентификаторы FOO, foo и «foo» считаются одинаковыми в Синергия-БД, но «FOO» и «FOO» отличны друг от друга и от предыдущих трех. Приведение имен без кавычек к нижнему регистру, как это делает Синергия-БД, несовместимо со стандартом SQL, который говорит о том, что имена должны приводиться к верхнему регистру. То есть, согласно стандарту foo должно быть эквивалентно «FOO», а не «foo». Поэтому при создании переносимых приложений рекомендуется либо всегда заключать определенное имя в кавычки, либо не заключать никогда.

2.1.2. Константы

В Синергия-БД есть три типа констант подразумеваемых типов: строки, битовые строки и числа. Константы можно также записывать, указывая типы явно, что позволяет представить их более точно и обработать более эффективно. Эти варианты рассматриваются в следующих подразделах.

2.1.2.1. Строковые константы

Строковая константа в SQL – это обычная последовательность символов, заключенная в апострофы ('), например: 'Это строка'. Чтобы включить апостроф в строку, напишите в ней два апострофа рядом, например: 'Жанна д"Арк'. Заметьте, это не то же самое, что двойная кавычка («).

Две строковые константы, разделенные пробельными символами и минимум одним переводом строки, объединяются в одну и обрабатываются, как если бы строка была записана в одной константе, например:

```
SELECT 'foo'
      'bar';
```

эквивалентно:

```
SELECT 'foobar';
```

но эта запись:

```
SELECT 'foo'      'bar';
```

считается синтаксической ошибкой (это несколько странное поведение определено в стандарте SQL, Синергия-БД просто следует ему).

2.1.2.2. Строковые константы со спецпоследовательностями в стиле С

Синергия-БД также принимает «спецпоследовательности», что является расширением стандарта SQL. Строка со спецпоследовательностями начинается с буквы E (заглавной или строчной), стоящей непосредственно перед апострофом, например: E'foo'. Когда константа со спецпоследовательностью разбивается на несколько строк, букву E нужно поставить только перед первым открывающим апострофом. Внутри таких строк символ обратной косой черты (\) начинает C-подобные спецпоследовательности, в которых сочетание обратной косой черты со следующим символом (ами) дает определенное байтовое значение, как показано в таблице 2.1.

Таблица 2.1 – Спецпоследовательности

Спецпоследовательность	Интерпретация
\b	символ «забой»
\f	подача формы
\n	новая строка
\r	возврат каретки
\t	табуляция
\o, \oo, \ooo (o = 0 - 7)	восьмеричное значение байта
\xh, \xhh (h = 0 - 9, A - F)	шестнадцатеричное значение байта
\uxxxx, \Uxxxxxxxx (x = 0 - 9, A - F)	16- или 32-битный шестнадцатеричный код символа Unicode

Любой другой символ, идущий после обратной косой черты, воспринимается буквально. Таким образом, чтобы включить в строку обратную косую черту, нужно написать две косых черты (\\). Так же можно включить в строку апостроф, написав \', в дополнение к обычному способу ".

Вы должны позаботиться, чтобы байтовые последовательности, которые вы создаете таким образом, особенно в восьмеричной и шестнадцатеричной записи, образовывали допустимые символы в серверной кодировке. Когда сервер работает с кодировкой UTF-8, вместо такой записи байт следует использовать спецпоследовательности Unicode или альтернативный синтаксис Unicode, описанный в подразделе 2.1.2.3. В противном случае придется кодировать символы UTF-8 вручную и выписывать их по байтам, что очень неудобно.

Спецпоследовательности с Unicode полностью поддерживаются только при использовании на сервере кодировки UTF8. Когда используются другие кодировки, допускается указание только ASCII-символов (с кодами до \u007F). И в четырех, и в восьмизначной форме можно записывать суррогатные пары UTF-16 и таким образом составлять символы с кодами больше чем U+FFFF, хотя наличие восьмизначной формы технически делает это ненужным. Когда суррогатные пары используются с серверной кодировкой UTF8, они сначала объединяются в один символ, который затем кодируется в UTF-8.

Примечание. Если параметр конфигурации `standard_conforming_strings` имеет значение `off`, Синергия-БД распознает обратную косую черту как спецсимвол и в обычных строках, и в строках со спецпоследовательностями. Однако в версии Синергия-БД 1 по умолчанию принято значение `on`, и в этом случае обратная косая черта распознается только в спецстроках. Это поведение больше соответствует стандарту, хотя может нарушить работу приложений, рассчитанных на предыдущий режим, когда обратная косая черта распознавалась везде. В качестве временного решения вы можете изменить этот параметр на `off`, но лучше уйти от такой практики. Если вам нужно, чтобы обратная косая черта представляла специальный символ, задайте строковую константу с `E`.

В дополнение к `standard_conforming_strings` поведением обратной косой черты в строковых константах управляют параметры `escape_string_warning` и `backslash_quote`.

Строковая константа не может включать символ с кодом «0».

2.1.2.3. Строковые константы со спецпоследовательностями Unicode

Синергия-БД также поддерживает ещё один вариант спецпоследовательностей, позволяющий включать в строки символы Unicode по их кодам. Строковая константа со спецпоследовательностями Unicode начинается с `U&`

(строчная или заглавная U и амперсанд), а затем сразу без пробелов идет апостроф, например U&'foo'. Заметьте, что при этом возникает неоднозначность с оператором &. Чтобы её избежать, окружайте этот оператор пробелами. Затем в апострофах можно записывать символы Unicode двумя способами: обратная косая черта, а за ней код символа из четырех шестнадцатеричных цифр, либо обратная косая черта, знак плюс, а затем код из шести шестнадцатеричных цифр. Например, строку 'data' можно записать так:

```
U&'d\0061t\+000061'
```

В следующем менее тривиальном примере закодировано русское слово «слон», записанное кириллицей:

```
U&' \0441\043B\043E\043D'
```

Если вы хотите использовать не обратную косую черту, а другой спецсимвол, его можно указать, добавив UESCAPE после строки, например:

```
U&'d!0061t!+000061' UESCAPE '!'
```

В качестве спецсимвола можно выбрать любой символ, кроме шестнадцатеричной цифры, знака плюс, апострофа, кавычки или пробельного символа.

Спецпоследовательности с Unicode поддерживаются только при использовании на сервере кодировки UTF8. Когда используются другие кодировки, допускается указание только ASCII-символов (с кодами до \007F). И в четырех, и в шестизначной форме можно записывать суррогатные пары UTF-16 и таким образом составлять символы с кодами больше чем U+FFFF, хотя наличие шестизначной формы технически делает это ненужным. Когда суррогатные пары используются с серверной кодировкой UTF8, они сначала объединяются в один символ, который затем кодируется в UTF-8.

Также заметьте, что спецпоследовательности Unicode в строковых константах работают, только когда параметр конфигурации `standard_conforming_strings` равен `on`. Это объясняется тем, что иначе клиентские программы, проверяющие SQL-операторы, можно будет ввести в заблуждение и эксплуатировать это как уязвимость, например, для SQL-инъекций. Если этот

параметр имеет значение `off`, эти спецпоследовательности будут вызывать ошибку.

Чтобы включить спецсимвол в строку буквально, напишите его дважды.

2.1.2.4. Строковые константы, заключенные в доллары

Хотя стандартный синтаксис для строковых констант обычно достаточно удобен, он может плохо читаться, когда строка содержит много апострофов или обратных косых черт, так как каждый такой символ приходится дублировать. Чтобы и в таких случаях запросы оставались читаемыми, Синергия-БД предлагает ещё один способ записи строковых констант – «заключение строк в доллары». Строковая константа, заключенная в доллары, начинается со знака доллара (\$), необязательного «тега» из нескольких символов и ещё одного знака доллара, затем содержит обычную последовательность символов, составляющую строку, и оканчивается знаком доллара, тем же тегом и замыкающим знаком доллара. Например, строку «Жанна д'Арк» можно записать в долларах двумя способами:

```
$Жанна д'Арк$
```

```
$SomeTag$Жанна д'Арк$SomeTag$
```

Заметьте, что внутри такой строки апострофы не нужно записывать особым образом. На самом деле, в строке, заключенной в доллары, все символы можно записывать в чистом виде: содержимое строки всегда записывается буквально. Ни обратная косая черта, ни даже знак доллара не являются спецсимволами, если только они не образуют последовательность, соответствующую открывающему тегу.

Строковые константы в долларах можно вкладывать друг в друга, выбирая на разных уровнях вложенности разные теги. Чаще всего это используется при написании определений функций, например:

```
$function$
```

```
BEGIN
```

```
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
```

```
END;
```

```
$function$
```

Здесь последовательность `$q$[\t\r\n\v\\]$q$` представляет в долларах текстовую строку `[\t\r\n\v\\]`, которая будет обработана, когда Синергия-БД будет выполнять эту функцию. Но так как эта последовательность не соответствует

внешнему тегу в долларах (`$function$`), с точки зрения внешней строки это просто обычные символы внутри константы.

Тег строки в долларах, если он присутствует, должен соответствовать правилам, определенным для идентификаторов без кавычек, и к тому же не должен содержать знак доллара. Теги регистрозависимы, так что `$tag$String content$tag$` – правильная строка, а `$TAG$String content$tag$` – нет.

Строка в долларах, следующая за ключевым словом или идентификатором, должна отделяться от него пробельными символами, иначе доллар будет считаться продолжением предыдущего идентификатора.

Заключение строк в доллары не является частью стандарта SQL, но часто это более удобный способ записывать сложные строки, чем стандартный вариант с апострофами. Он особенно полезен, когда нужно представить строковую константу внутри другой строки, что часто требуется в определениях процедурных функций. Ограничившись только апострофами, каждую обратную косую черту в приведенном примере пришлось бы записывать четырьмя такими символами, которые бы затем уменьшились до двух при разборе внешней строки, и наконец, до одного при обработке внутренней строки во время выполнения функции.

2.1.2.5. Битовые строковые константы

Битовые строковые константы похожи на обычные с дополнительной буквой В (заглавной или строчной), добавленной непосредственно перед открывающим апострофом (без промежуточных пробелов), например: `B'1001'`. В битовых строковых константах допускаются лишь символы 0 и 1.

Битовые константы могут быть записаны и по-другому, в шестнадцатеричном виде, с начальной буквой Х (заглавной или строчной), например: `X'1FF'`. Такая запись эквивалентна двоичной, только четыре двоичных цифры заменяются одной шестнадцатеричной.

Обе формы записи допускают перенос строк так же, как и обычные строковые константы. Однако заключать в доллары битовые строки нельзя.

2.1.2.6. Числовые константы

Числовые константы могут быть заданы в следующем общем виде:
цифры

цифры . [цифры] [е [+ -] цифры]
[цифры] .цифры[е [+ -] цифры]
цифрые [+ -] цифры

где цифры — это одна или несколько десятичных цифр (0..9). До или после десятичной точки (при её наличии) должна быть минимум одна цифра. Как минимум одна цифра должна следовать за обозначением экспоненты (е), если оно присутствует. В числовой константе не может быть пробелов или других символов. Заметьте, что любой знак минус или плюс в начале строки не считается частью числа; это оператор, примененный к константе.

Несколько примеров допустимых числовых констант:

42
3.5
4.
.001
5e2
1.925e-3

Числовая константа, не содержащая точки и экспоненты, изначально рассматривается как константа типа `integer`, если её значение уместается в 32-битный тип `integer`; затем как константа типа `bigint`, если её значение уместается в 64-битный `bigint`; в противном случае она принимает тип `numeric`. Константы, содержащие десятичные точки и/или экспоненты, всегда считаются константами типа `numeric`.

Изначально назначенный тип данных числовой константы это только отправная точка для алгоритмов определения типа. В большинстве случаев константа будет автоматически приведена к наиболее подходящему типу для данного контекста. При необходимости вы можете принудительно интерпретировать числовое значение как значение определенного типа, приведя его тип к нужному. Например, вы можете сделать, чтобы числовое значение рассматривалось как имеющее тип `real (float4)`, написав:

```
REAL '1.23'    -- строковый стиль
1.23::REAL     -- стиль PostgreSQL (исторический)
```

На самом деле это только частные случаи синтаксиса приведения типов,

который будет рассматриваться далее.

2.1.2.7. Константы других типов

Константу обычного типа можно ввести одним из следующих способов:

```
type 'string'  
'string'::type  
CAST ( 'string' AS type )
```

Текст строковой константы передается процедуре преобразования ввода для типа, обозначенного здесь `type`. Результатом становится константа указанного типа. Явное приведение типа можно опустить, если нужный тип константы определяется однозначно (например, когда она присваивается непосредственно колонке таблицы), так как в этом случае приведение происходит автоматически.

Строковую константу можно записать, используя как обычный синтаксис SQL, так и формат с долларами.

Также можно записать приведение типов, используя синтаксис функций:

```
typename ( 'string' )
```

но это работает не для всех имен типов; подробнее об этом написано в подразделе 2.2.9.

Конструкцию `::`, `CAST()` и синтаксис вызова функции можно также использовать для преобразования типов обычных выражений во время выполнения, как описано в подразделе 2.2.9. Во избежание синтаксической неопределенности, запись тип 'строка' можно использовать только для указания типа простой текстовой константы. Ещё одно ограничение записи тип 'строка': она не работает для массивов; для таких констант следует использовать `::` или `CAST()`.

Синтаксис `CAST()` соответствует SQL, а запись `type 'string'` является обобщением стандарта: в SQL такой синтаксис поддерживает только некоторые типы данных, но Синергия-БД позволяет использовать его для всех. Синтаксис с `::` имеет исторические корни в Синергия-БД, как и запись в виде вызова функции.

2.1.3. Операторы

Имя оператора образует последовательность не более чем `NAMEDATALEN-1` (по умолчанию 63) символов из следующего списка:

+ - * / < > = ~ ! @ # % ^ & | ` ?

Однако для имен операторов есть ещё несколько ограничений:

- сочетания символов -- и /* не могут присутствовать в имени оператора, так как они будут обозначать начало комментария;
- многосимвольное имя оператора не может заканчиваться знаками + или -, если только оно не содержит также один из этих символов:

~ ! @ # % ^ & | ` ?

Например, @- – допустимое имя оператора, а *- – нет. Благодаря этому ограничению Синергия-БД, может разбирать корректные SQL-запросы без пробелов между компонентами.

Записывая не стандартные SQL-операторы, обычно для однозначности нужно отделять имена соседних операторов пробелами. Например, если вы определили левый унарный оператор с именем @, вы не можете написать X*@Y, а должны написать X* @Y, чтобы Синергия-БД однозначно прочитала это как два оператора, а не один.

2.1.4. Специальные знаки

Некоторые не алфавитно-цифровые символы имеют специальное значение, но при этом не являются операторами. Подробнее их использование будет рассмотрено при описании соответствующего элемента синтаксиса. Здесь они упоминаются только для сведения и обобщения их предназначения:

1. Знак доллара (\$), предваряющий число, используется для представления позиционного параметра в теле определения функции или подготовленного оператора. В других контекстах знак доллара может быть частью идентификатора или строковой константы, заключенной в доллары.
2. Круглые скобки (()) имеют обычное значение и применяются для группировки выражений и повышения приоритета операций. В некоторых случаях скобки – это необходимая часть синтаксиса определенных SQL-команд.
3. Квадратные скобки ([]) применяются для выделения элементов массива. Подробнее массивы рассматриваются в разделе 6.15.
4. Запятые (,) используются в некоторых синтаксических конструкциях для разделения элементов списка.

5. Точка с запятой (;) завершает команду SQL. Она не может находиться нигде внутри команды, за исключением строковых констант или идентификаторов в кавычках.

6. Двоеточие (:) применяется для выборки «срезов» массивов (см. раздел 6.15.) В некоторых диалектах SQL (например, в Embedded SQL) двоеточие может быть префиксом в имени переменной.

7. Звездочка (*) используется в некоторых контекстах как обозначение всех полей строки или составного значения. Она также имеет специальное значение, когда используется как аргумент некоторых агрегатных функций, а именно функций, которым не нужны явные параметры.

8. Точка (.) используется в числовых константах, а также для отделения имен схемы, таблицы и колонки.

2.1.5. Комментарии

Комментарий – это последовательность символов, которая начинается с двух минусов и продолжается до конца строки, например:

```
-- Это стандартный комментарий SQL
```

Кроме этого, блочные комментарии можно записывать в стиле C:

```
/* многострочный комментарий  
 * с вложенностью: /* вложенный блок комментария */  
 */
```

где комментарий начинается с /* и продолжается до соответствующего вхождения */. Блочные комментарии можно вкладывать друг в друга, как разрешено по стандарту SQL (но не разрешено в C), так что вы можете комментировать большие блоки кода, которые при этом уже могут содержать блоки комментариев.

Комментарий удаляется из входного потока в начале синтаксического анализа и фактически заменяется пробелом.

2.1.6. Приоритеты операторов

В таблице 2.2 показаны приоритеты и очередность операторов, действующие в Синергия-БД. Большинство операторов имеют одинаковый приоритет и

вычисляются слева направо. Приоритет и очередность операторов жестко фиксированы в синтаксическом анализаторе. Вследствие этого их поведение может быть не интуитивным, например, логические операторы < и > имеют приоритет, отличный от логических операторов <= и >=. Кроме того, иногда вам потребуется добавлять скобки, когда вы комбинируете унарные и бинарные операторы. Например, выражение:

```
SELECT 5 ! - 6;
```

будет разобрано как:

```
SELECT 5 ! (- 6);
```

так как анализатор до последнего не знает, что оператор ! определен как постфиксный, а не инфиксный (внутренний). Чтобы получить желаемый результат в этом случае, нужно написать:

```
SELECT (5 !) - 6;
```

Такова цена расширяемости.

Таблица 2.2 – Приоритет операторов (по убыванию)

Оператор/элемент	Очерёдность	Описание
.	слева-направо	разделитель имён таблицы и столбца
::	слева-направо	приведение типов в стиле PostgreSQL
[]	слева-направо	выбор элемента массива
+ -	справа-налево	унарный плюс, унарный минус
^	слева-направо	возведение в степень
* / %	слева-направо	умножение, деление, остаток от деления
+ -	слева-направо	сложение, вычитание
(любой другой оператор)	слева-направо	все другие встроенные и пользовательские операторы
BETWEEN IN LIKE ILIKE SIMILAR	-	проверка диапазона, проверка членства, сравнение строк
< > = <= >= <>	-	операторы сравнения
IS ISNULL NOTNULL	-	IS TRUE, IS FALSE, IS NULL, IS DISTINCT FROM и т. д.
NOT	справа-налево	логическое отрицание
AND	слева-направо	логическая конъюнкция
OR	слева-направо	логическая дизъюнкция

Заметьте, что правила приоритета операторов также применяются к операторам, определенным пользователем с теми же именами, что и вышеперечисленные встроенные операторы. Например, если вы определите оператор «+» для некоторого нестандартного типа данных, он будет иметь тот же приоритет, что и встроенный оператор «+», независимо от того, что он у вас делает.

Когда в конструкции OPERATOR используется имя оператора со схемой,

например так:

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

тогда OPERATOR имеет приоритет по умолчанию, соответствующий в таблице 2.2 строке «все остальные». Это не зависит от того, какие именно операторы находятся в конструкции OPERATOR () .

2.2. Выражения значения

Выражения значения применяются в самых разных контекстах, например в списке результатов команды SELECT, в значениях колонок в INSERT или UPDATE или в условиях поиска во многих командах. Результат такого выражения иногда называют скаляром, чтобы отличить его от результата табличного выражения (который представляет собой таблицу). А сами выражения значения часто называют скалярными (или просто выражениями). Синтаксис таких выражений позволяет вычислять значения из примитивных частей, используя арифметические, логические и другие операции.

Выражениями значения являются:

- константа или непосредственное значение;
- ссылка на колонку;
- ссылка на позиционный параметр в теле определения функции или подготовленного оператора;
- выражение с индексом;
- выражение выбора поля;
- применение оператора;
- вызов функции;
- агрегатное выражение;
- вызов оконной функции;
- приведение типов;
- применение правил сортировки;
- скалярный подзапрос;
- конструктор массива;

- конструктор табличной строки.

Кроме того, выражением значения являются скобки (предназначенные для группировки подвыражений и переопределения приоритета).

В дополнение к этому списку есть ещё несколько конструкций, которые можно классифицировать как выражения, хотя они не соответствуют общим синтаксическим правилам. Они обычно имеют вид функции или оператора. Пример такой конструкции – предложение `IS NULL`.

Мы уже обсудили константы в подразделе 2.1.2. В следующих разделах рассматриваются остальные варианты.

2.2.1. Ссылки на колонки

Ссылку на колонку можно записать в форме:

`отношение.имя_колонки`

Здесь отношение – имя таблицы (возможно, полное, с именем схемы) или её псевдоним, определенный в предложении `FROM`. Это имя и разделяющую точку можно опустить, если имя колонки уникально среди всех таблиц, задействованных в текущем запросе.

2.2.2. Позиционные параметры

Ссылка на позиционный параметр применяется для обращения к значению, переданному в SQL-оператор извне. Параметры используются в определениях SQL-функций и подготовленных операторов. Некоторые клиентские библиотеки также поддерживают передачу значений данных отдельно от самой SQL-команды, и в этом случае параметры позволяют ссылаться на такие значения. Ссылка на параметр записывается в следующей форме:

`$число`

Например, рассмотрим следующее определение функции `dept`:

```
CREATE FUNCTION dept(text) RETURNS dept
AS $$ SELECT * FROM dept WHERE name = $1 $$
LANGUAGE SQL;
```

Здесь `$1` всегда будет ссылаться на значение первого аргумента функции.

2.2.3. Индексы элементов

Если в выражении вы имеете дело с массивом, то можно извлечь определенный его элемент, написав:

выражение [индекс]

или несколько соседних элементов («срез массива»):

выражение [нижний_индекс:верхний_индекс]

Здесь квадратные скобки [] должны быть в явном виде. Каждый индекс сам по себе является выражением, результатом которого должно быть целое число.

В общем случае выражение массива должно заключаться в круглые скобки, но их можно опустить, когда выражение с индексом – это просто ссылка на колонку или позиционный параметр. Кроме того, можно соединить несколько индексов, если исходный массив многомерный, например:

моя_таблица.колонка_массив [4]

моя_таблица.колонка_массив_2d [17] [34]

\$1 [10:42]

(функция_массив (a, b)) [42]

В последней строке круглые скобки необходимы. Подробнее массивы рассматриваются в разделе 6.15.

2.2.4. Выбор поля

Если результат выражения – значение составного типа (строка таблицы), тогда определенное поле этой строки можно извлечь, написав:

выражение.имя_поля

В общем случае выражение такого типа должно заключаться в круглые скобки, но их можно опустить, когда это ссылка на таблицу или позиционный параметр, например:

моя_таблица.колонка

\$1.колонка

(функция_кортеж (a, b)).кол3

Таким образом, полная ссылка на колонку – это просто частный случай выбора

поля. Важный особый случай здесь – извлечение поля из колонки составного типа:

```
(составная_колонка) .поле
```

```
(моя_таблица.составная_колонка) .поле
```

Здесь скобки нужны, чтобы показать, что `составная_колонка` – это имя колонки, а не таблицы, и что `моя_таблица` – имя таблицы, а не схемы.

В списке выборки (см. раздел 5.3) вы можете запросить все поля составного значения, написав `.*`:

```
(составная_колонка) . *
```

2.2.5. Применение оператора

Существуют три возможных синтаксиса применения операторов:

```
выражение оператор выражение (бинарный инфиксный оператор)
```

```
оператор выражение (унарный префиксный оператор)
```

```
выражение оператор (унарный постфиксный оператор)
```

где оператор соответствует синтаксическим правилам, описанным в подразделе 2.1.3, либо это одно из ключевых слов AND, OR и NOT, либо полное имя оператора в форме:

```
OPERATOR (схема.имя_оператора)
```

Существование конкретных операторов и их тип (унарный или бинарный) зависит от того, как и какие операторы определены системой и пользователем.

2.2.6. Вызовы функций

Вызов функции записывается просто как имя функции (возможно, дополненное именем схемы) и список аргументов в скобках:

```
имя_функции ([выражение [, выражение ... ]])
```

Например, так вычисляется квадратный корень из 2:

```
sqrt(2)
```

Пользователь также может определить и другие функции.

Выполняя запросы в базе данных, где одни пользователи могут не доверять другим, в записи вызовов функций соблюдайте меры предосторожности.

Аргументам могут быть присвоены необязательные имена. Подробнее об этом

см. раздел 2.3.

Примечание. Функцию, принимающую один аргумент составного типа, можно также вызывать, используя синтаксис выбора поля, и наоборот, выбор поля можно записать в функциональном стиле. То есть записи `col(table)` и `table.col` равносильны и взаимозаменяемы. Это поведение не оговорено стандартом SQL, но реализовано в Синергия-БД, так как это позволяет использовать функции для эмуляции «вычисляемых полей».

2.2.7. Агрегатные выражения

Агрегатное выражение представляет собой применение агрегатной функции к строкам, выбранным запросом. Агрегатная функция сводит множество входных значений к одному выходному, как например, сумма или среднее. Агрегатное выражение может записываться следующим образом:

```
агрегатная_функция (выражение [ , ... ]
[ предложение_order_by ] ) [ FILTER ( WHERE
условие_фильтра ) ]

агрегатная_функция (ALL выражение [ , ... ]
[ предложение_order_by ] ) [ FILTER ( WHERE
условие_фильтра ) ]

агрегатная_функция (DISTINCT выражение [ , ... ]
[ предложение_order_by ] ) [ FILTER ( WHERE
условие_фильтра ) ]

агрегатная_функция ( * ) [ FILTER ( WHERE
условие_фильтра ) ]

агрегатная_функция ( [ выражение [ , ... ] ] ) WITHIN
GROUP ( предложение_order_by ) [ FILTER ( WHERE
условие_фильтра ) ]
```

Здесь `агрегатная_функция` – имя ранее определенной агрегатной функции (возможно, дополненное именем схемы), `выражение` – любое выражение значения, не содержащее в себе агрегатного выражения или вызова оконной функции. Необязательные предложения `предложение_order_by` и `условие_фильтра` описываются ниже.

В первой форме агрегатного выражения агрегатная функция вызывается для каждой строки. Вторая форма эквивалентна первой, так как указание `ALL` подразумевается по умолчанию. В третьей форме агрегатная функция вызывается для всех различных значений выражения (или набора различных значений, для

нескольких выражений), выделенных во входных данных. В четвертой форме агрегатная функция вызывается для каждой строки, так как никакого конкретного значения не указано (обычно это имеет смысл только для функции `count(*)`). В последней форме используются сортирующие агрегатные функции, которые будут описаны ниже.

Большинство агрегатных функций игнорируют значения `NULL`, так что строки, для которых выражения выдают одно или несколько значений `NULL`, отбрасываются. Это можно считать истинным для всех встроенных операторов, если явно не говорится об обратном.

Например, `count(*)` подсчитает общее количество строк, а `count(f1)` только количество строк, в которых `f1` не `NULL` (так как `count` игнорирует `NULL`), а `count(distinct f1)` подсчитает число различных и отличных от `NULL` значений колонки `f1`.

Обычно строки данных передаются агрегатной функции в неопределенном порядке и во многих случаях это не имеет значения, например функция `min` выдает один и тот же результат независимо от порядка поступающих данных. Однако некоторые агрегатные функции (такие как `array_agg` и `string_agg`) выдают результаты, зависящие от порядка данных. Для таких агрегатных функций можно добавить предложение `order_by` и задать нужный порядок. Это предложение `order_by` имеет тот же синтаксис, что и предложение `ORDER BY` на уровне запроса, как описано в разделе 5.5, за исключением того, что его выражения должны быть просто выражениями, но не именами результирующих колонок или числами, например:

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

Заметьте, что при использовании агрегатных функций с несколькими аргументами, предложение `ORDER BY` идет после всех аргументов. Например, надо писать так:

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

а не так:

```
SELECT string_agg(a ORDER BY a, ',') FROM table; --
```

неправильно

Последний вариант синтаксически допустим, но он представляет собой вызов агрегатной функции одного аргумента с двумя ключами ORDER BY (при этом второй не имеет смысла, так как это константа).

Если предложение `_order_by` дополнено указанием DISTINCT, тогда все выражения ORDER BY должны соответствовать обычным аргументам агрегатной функции; то есть вы не можете сортировать строки по выражению, не включенному в список DISTINCT.

Примечание. Возможность указывать и DISTINCT, и ORDER BY в агрегатной функции – это расширение Синергия-БД.

При добавлении ORDER BY в обычный список аргументов агрегатной функции, описанном до этого, выполняется сортировка входных строк для универсальных и статистических агрегатных функций, для которых сортировка необязательна. Но есть подмножество агрегатных функций, сортирующие агрегатные функции, для которых предложение `_order` является обязательным, обычно потому, что вычисление этой функции имеет смысл только при определённой сортировке входных строк. Типичными примерами сортирующих агрегатных функций являются вычисления ранга и процентиля. Для сортирующей агрегатной функции предложение `_order_by` записывается внутри WITHIN GROUP (...), что иллюстрирует последний пример, приведённый выше. Выражения в предложении `_order_by` вычисляются однократно для каждой входной строки как аргументы обычной агрегатной функции, сортируются в соответствии с требованием предложения `_order_by` и поступают в агрегатную функцию как входящие аргументы. Если же предложение `_order_by` находится не в WITHIN GROUP, оно не передаётся как аргумент(ы) агрегатной функции. Выражения-аргументы, предшествующие WITHIN GROUP, (если они есть), называются непосредственными аргументами, а выражения, указанные в предложении `_order_by` – агрегируемыми аргументами. В отличие от аргументов обычной агрегатной функции, непосредственные аргументы вычисляются однократно для каждого вызова функции, а не для каждой строки. Это

значит, что они могут содержать переменные, только если эти переменные сгруппированы в GROUP BY; это суть то же ограничение, что действовало бы, будь эти непосредственные аргументы вне агрегатного выражения. Непосредственные аргументы обычно используются, например, для указания значения процентиля, которое имеет смысл, только если это конкретное число для всего расчёта агрегатной функции. Список непосредственных аргументов может быть пуст; в этом случае запишите просто (), но не (*). На самом деле Синергия-БД примет обе записи, но только первая соответствует стандарту SQL.

Пример вызова сортирующей агрегатной функции:

```
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY
income) FROM households;
percentile_cont
-----
                    50489
```

она получает 50-ый перцентиль, или медиану, значения столбца income из таблицы households. В данном случае 0.5 – это непосредственный аргумент; если бы дробь процентиля менялась от строки к строке, это не имело бы смысла.

Если добавлено предложение FILTER, агрегатной функции подаются только те входные строки, для которых условие_фильтра вычисляется как истинное; другие строки отбрасываются, например:

```
SELECT
    count(*) AS unfiltered,
    count(*) FILTER (WHERE i < 5) AS filtered
FROM generate_series(1,10) AS s(i);
unfiltered | filtered
-----+-----
          10 |          4
(1 row)
```

Агрегатное выражение может фигурировать только в списке результатов или в предложении HAVING команды SELECT. Во всех остальных предложениях, например WHERE, они запрещены, так как эти предложения логически вычисляются

до того, как формируются результаты агрегатных функций.

Когда агрегатное выражение используется в подзапросе (см. подраздел 2.2.11), оно обычно вычисляется для всех строк подзапроса. Но если в аргументах (или в условии `_filter`) агрегатной функции есть только переменные внешнего уровня, агрегатная функция относится к ближайшему внешнему уровню и вычисляется для всех строк соответствующего запроса. Такое агрегатное выражение в целом является внешней ссылкой для своего подзапроса и на каждом вычислении считается константой. При этом допустимое положение агрегатной функции ограничивается списком результатов и предложением `HAVING` на том уровне запросов, где она находится.

2.2.8. Вызовы оконных функций

Вызов оконной функции представляет собой применение функции, подобной агрегатной, к некоторому набору строк, выбранному запросом. В отличие от вызовов не оконных агрегатных функций, при этом не происходит группировка выбранных строк в одну – каждая строка остаётся отдельной в результате запроса. Однако оконная функция имеет доступ ко всем строкам, вошедшим в группу текущей строки согласно указанию группировки (списку `PARTITION BY`) в вызове оконной функции. Вызов оконной функции может иметь следующие формы:

```
имя_функции ([выражение [, выражение ... ]]) [ FILTER
( WHERE условие_фильтра ) ] OVER имя_окна
имя_функции ([выражение [, выражение ... ]]) [ FILTER
( WHERE условие_фильтра ) ] OVER ( определение_окна )
имя_функции ( * ) [ FILTER ( WHERE условие_фильтра ) ]
OVER имя_окна
имя_функции ( * ) [ FILTER ( WHERE условие_фильтра ) ]
OVER ( определение_окна )
```

Здесь определение_окна записывается в виде:

```
[ имя_существующего_окна ]
[ PARTITION BY выражение [, ...] ]
[ ORDER BY выражение [ ASC | DESC | USING оператор ]
[ NULLS { FIRST | LAST } ] [, ...] ]
[ определение_рамки ]
```


и необязательное определение `_рамки` может иметь вид:

`{ RANGE | ROWS } начало_рамки`

`{ RANGE | ROWS } BETWEEN начало_рамки AND конец_рамки`

Здесь `начало_рамки` и `конец_рамки` задаются одним из следующих способов:

`UNBOUNDED PRECEDING`

значение `PRECEDING`

`CURRENT ROW`

значение `FOLLOWING`

`UNBOUNDED FOLLOWING`

и исключение `_рамки` может быть следующим:

`EXCLUDE CURRENT ROW`

`EXCLUDE GROUP`

`EXCLUDE TIES`

`EXCLUDE NO OTHERS`

Здесь выражение – это любое выражение значения, не содержащее вызовов оконных функций.

`имя_окна` – ссылка на именованное окно, определенное предложением `WINDOW` в данном запросе. Также возможно написать в скобках полное определение `_окна`, используя тот же синтаксис определения именованного окна в предложении `WINDOW`; подробнее это описано в справке по `SELECT`. Стоит отметить, что запись `OVER имя_окна` не полностью равнозначна `OVER (имя_окна)`; последний вариант подразумевает копирование и изменение определения окна и не будет допустимым, если определение этого окна включает определение рамки.

Указание `PARTITION BY` группирует строки запроса в разделы, которые затем обрабатываются оконной функцией независимо друг от друга. `PARTITION BY` работает подобно предложению `GROUP BY` на уровне запроса, за исключением того, что его аргументы всегда просто выражения, а не имена выходных колонок или числа. Без `PARTITION BY` все строки, выдаваемые запросом, рассматриваются как один раздел. Указание `ORDER BY` определяет порядок, в котором оконная функция

обрабатывает строки раздела. Оно так же подобно предложению ORDER BY на уровне запроса и так же не принимает имена выходных колонок или числа. Без ORDER BY строки обрабатываются в неопределенном порядке.

определение_рамки задает набор строк, образующих рамку окна, которая представляет собой подмножество строк текущего раздела и используется для оконных функций, работающих с рамкой, а не со всем разделом. Подмножество строк в рамке может меняться в зависимости от того, какая строка является текущей. Рамку можно указать в режимах RANGE или ROWS; в любом случае она начинается с положения начало_рамки и заканчивается положением конец_рамки. Если конец_рамки опущен, подразумевается CURRENT ROW (текущая строка).

Если начало_рамки задано как UNBOUNDED PRECEDING, рамка начинается с первой строки раздела, а если конец_рамки определен как UNBOUNDED FOLLOWING, рамка заканчивается последней строкой раздела.

В режиме RANGE начало_рамки, заданное как CURRENT ROW, определяет в качестве начала первую родственную строку (строку, которую ORDER BY считает равной текущей), тогда как конец_рамки, заданный как CURRENT ROW, определяет концом рамки последнюю родственную (для ORDER BY) строку. В режиме ROWS вариант CURRENT ROW просто обозначает текущую строку.

В вариантах определения рамки смещение PRECEDING и смещение FOLLOWING в качестве смещения должно задаваться выражение, не содержащее какие-либо переменные и вызовы агрегатных или оконных функций. Что именно будет означать смещение, определяется в зависимости от режима рамки:

1. В режиме ROWS смещение должно задаваться отличным от NULL неотрицательным целым числом, и это число определяет сдвиг, с которым начало рамки позиционируется перед текущей строкой, а конец – после текущей строки.
2. В режиме GROUPS смещение также должно задаваться отличным от NULL неотрицательным целым числом, и это число определяет сдвиг (по количеству групп родственных строк), с которым начало рамки позиционируется перед группой строк, родственных текущей, а конец – после этой группы. Группу

родственных строк образуют строки, которые считаются равными согласно ORDER BY. Для использования режима GROUPS определение окна должно содержать предложение ORDER BY.

3. В режиме RANGE для использования этих указаний предложение ORDER BY должно содержать ровно один столбец. В этом случае смещение задаёт максимальную разницу между значением этого столбца в текущей строке и значением его же в предшествующих или последующих строках рамки. Тип данных выражения смещение зависит от типа данных упорядочивающего столбца. Для числовых столбцов это обычно тот же числовой тип, а для столбцов с типом дата/время – тип interval. Например, если упорядочивающий столбец имеет тип date или timestamp, возможна такая запись: RANGE BETWEEN '1 day' PRECEDING AND '10 days' FOLLOWING. Значение смещение при этом может так же быть отличным от NULL и неотрицательным, хотя что считать «неотрицательным», будет зависит от типа данных.

В любом случае расстояние до конца рамки ограничивается расстоянием до конца раздела, так что для строк, которые находятся у конца раздела, рамка может содержать меньше строк, чем для других.

Заметьте, что в режимах ROWS и GROUPS указания 0 PRECEDING и 0 FOLLOWING равнозначны указанию CURRENT ROW. Обычно это справедливо и для режима RANGE, в случае подходящего для типа данных определения значения «нуля».

Дополнение `исключение_рамки` позволяет исключить из рамки строки, которые окружают текущую строку, даже если они должны быть включены согласно указаниям, определяющим начало и конец рамки. `EXCLUDE CURRENT ROW` исключает из рамки текущую строку. `EXCLUDE GROUP` исключает из рамки текущую строку и родственные ей согласно порядку сортировки. `EXCLUDE TIES` исключает из рамки все родственные строки для текущей, но не собственно текущую строку. `EXCLUDE NO OTHERS` просто явно выражает поведение по умолчанию – не исключает ни текущую строку, ни родственные ей.

По умолчанию рамка определяется как `RANGE UNBOUNDED PRECEDING`, что равносильно расширенному определению `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. С указанием `ORDER BY` это означает, что рамка будет включать все строки от начала раздела до последней строки, родственной текущей (для `ORDER BY`). Без `ORDER BY` в рамку включаются все строки раздела, так как все они считаются родственными текущей.

Действуют также ограничения: начало_рамки не может определяться как `UNBOUNDED FOLLOWING`, а конец_рамки – `UNBOUNDED PRECEDING`, и конец_рамки не может определяться раньше, чем начало_рамки – например, запись `RANGE BETWEEN CURRENT ROW AND` значение `PRECEDING` недопустима.

Если добавлено предложение `FILTER`, оконной функции подаются только те входные строки, для которых условие_фильтра вычисляется как истинное; другие строки отбрасываются. Предложение `FILTER` допускается только для агрегирующих оконных функций.

Встроенные оконные функции описаны в таблице 7-53, но этот набор можно расширить, создавая собственные функции. Кроме того, в качестве оконных функций можно использовать любые встроенные или пользовательские обычные агрегатные функции (сортирующие агрегатные функции использовать в качестве оконных нельзя).

Запись со звездочкой (*) применяется при вызове агрегатных функций в качестве оконных, например `count(*) OVER (PARTITION BY x ORDER BY y)`. Звездочка (*) обычно не применяется для не агрегатных оконных функций. Агрегатные оконные функции, в отличие от обычных агрегатных функций, не допускают использования `DISTINCT` и `ORDER BY` в списке аргументов.

Вызовы оконных функций разрешены в запросах только в списке `SELECT` и в предложении `ORDER BY`.

Дополнительно об оконных функциях можно узнать в подразделе 5.2.4.

2.2.9. Приведения типов

Приведение типа определяет преобразование данных из одного типа в другой.

Синергия-БД воспринимает две равносильные записи приведения типов:

```
CAST ( выражение AS тип )  
выражение :: тип
```

Запись с CAST соответствует стандарту SQL, тогда как вариант с :: – историческое наследие Синергия-БД.

Когда приведению подвергается значение выражения известного типа, происходит преобразование типа во время выполнения. Это приведение будет успешным, только если определен подходящий оператор преобразования типов. Обратите внимание на небольшое отличие от приведения констант, описанного в подразделе 2.1.2.7. Приведение строки в чистом виде представляет собой начальное присваивание строковой константы и оно будет успешным для любого типа (конечно, если строка содержит значение, приемлемое для данного типа данных).

Неявное приведение типа можно опустить, если возможно однозначно определить, какой тип должно иметь выражение (например, когда оно присваивается колонке таблицы); в таких случаях система автоматически преобразует тип. Однако автоматическое преобразование выполняется только для приведений с пометкой «допускается неявное применение» в системных каталогах. Все остальные приведения должны записываться явно. Это ограничение позволяет избежать сюрпризов с неявным преобразованием.

Также можно записать приведение типа как вызов функции:

```
имя_типа ( выражение )
```

Однако это будет работать только для типов, имена которых являются также допустимыми именами функций. Например, `double precision` так использовать нельзя, а `float8` (альтернативное название того же типа) – можно. Кроме того, имена типов `interval`, `time` и `timestamp` из-за синтаксического конфликта можно использовать в такой записи только в кавычках. Таким образом, запись приведения типа в виде вызова функции провоцирует несоответствия и, возможно, лучше будет её не применять.

Примечание. Приведение типа, представленное в виде вызова функции, на самом деле соответствует внутреннему механизму. Даже при использовании двух стандартных типов записи внутри происходит вызов зарегистрированной функции,

выполняющей преобразование. По соглашению, именем такой функции преобразования является имя выходного типа и таким образом запись «в виде вызова функции» есть не что иное, как прямой вызов нижележащей функции преобразования. При создании переносимого приложения на это поведение, конечно, не следует рассчитывать. Подробнее это описано в справке CREATE CAST.

2.2.10. Применение правил сортировки

Предложение COLLATE переопределяет правило сортировки выражения. Оно добавляется после выражения:

выражение COLLATE правило_сортировки

где правило_сортировки – идентификатор правила, возможно дополненный именем схемы. Предложение COLLATE связывает выражение сильнее, чем операторы, так что при необходимости следует использовать скобки.

Если правило сортировки не определено явно, система либо выбирает его по колонкам, которые используются в выражении, либо, если таких колонок нет, переключается на установленное для базы данных правило сортировки по умолчанию.

Предложение COLLATE имеет два распространенных применения: переопределение порядка сортировки в предложении ORDER BY, например:

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE  
«C»;
```

и переопределение правил сортировки при вызове функций или операторов, возвращающих языкозависимые результаты, например:

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE «C»;
```

Заметьте, что в последнем случае предложение COLLATE добавлено к аргументу оператора, на действие которого мы хотим повлиять. При этом не имеет значения, к какому именно аргументу оператора или функции добавляется COLLATE, так как правило сортировки, применяемое к оператору или функции, выбирается при рассмотрении всех аргументов, а явное предложение COLLATE переопределяет правила сортировки для всех других аргументов. Однако добавление разных предложений COLLATE к нескольким аргументам будет ошибкой. Таким

образом, эта команда выдаст тот же результат:

```
SELECT * FROM tbl WHERE a COLLATE «C» > 'foo';
```

Но это будет ошибкой:

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE «C»;
```

здесь правило сортировки нельзя применить к результату оператора >, который имеет несравнимый тип данных boolean.

2.2.11. Скалярные подзапросы

Скалярный подзапрос – это обычный запрос SELECT в скобках, который возвращает ровно одну строку и одну колонку. Написание запросов освещается в главе 5. После выполнения запроса SELECT его единственный результат используется в окружающем его выражении. В качестве скалярного подзапроса нельзя использовать запросы, возвращающие более одной строки или колонки. Но если в результате выполнения подзапрос не вернет строк, скалярный результат считается равным NULL. В подзапросе можно ссылаться на переменные из окружающего запроса; в процессе одного вычисления подзапроса они будут считаться константами.

Например, следующий запрос находит самый населенный город в каждом штате:

```
SELECT name, (SELECT max(pop) FROM cities WHERE  
cities.state = states.name)  
FROM states;
```

2.2.12. Конструкторы массивов

Конструктор массива – это выражение, которое создает массив, определяя значения его элементов. Конструктор простого массива состоит из ключевого слова ARRAY, открывающей квадратной скобки [, списка выражений (разделенных запятыми), задающих значения элементов массива, и закрывающей квадратной скобки].

Пример.

```
SELECT ARRAY[1, 2, 3+4];  
array
```

```
{1,2,7}
(1 row)
```

По умолчанию типом элементов массива считается общий тип для всех выражений, определенный по правилам, действующим и для конструкций UNION и CASE. Вы можете переопределить его явно, приведя конструктор массива к требуемому типу, например:

```
SELECT ARRAY[1,2,22.7]::integer[];
      array
```

```
{1,2,23}
(1 row)
```

Это равносильно тому, что привести к нужному типу каждое выражение по отдельности. Подробнее приведение типов описано в подразделе 2.2.9.

Многомерные массивы можно образовывать, вкладывая конструкторы массивов. При этом во внутренних конструкторах слово ARRAY можно опускать. Например, результат работы этих конструкторов одинаков:

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
      array
```

```
{{1,2},{3,4}}
(1 row)
```

```
SELECT ARRAY[[1,2],[3,4]];
      array
```

```
{{1,2},{3,4}}
(1 row)
```

Многомерные массивы должны быть прямоугольными, и поэтому внутренние конструкторы одного уровня должны создавать вложенные массивы одинаковой размерности. Любое приведение типа, примененное к внешнему конструктору ARRAY, автоматически распространяется на все внутренние.

Элементы многомерного массива можно создавать не только вложенными конструкторами ARRAY, но и другими способами, позволяющими получить массивы нужного типа, например:

```
CREATE TABLE arr(f1 int[], f2 int[]);
```

```
INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]],  
ARRAY[[5,6],[7,8]]);
```

```
SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM  
arr;
```

array

```
-----  
{{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}  
(1 row)
```

Вы можете создать и пустой массив, но так как массив не может быть не типизированным, вы должны явно привести пустой массив к нужному типу, например:

```
SELECT ARRAY[]::integer[];
```

array

```
-----  
{}  
(1 row)
```

Также возможно создать запрос из результатов подзапроса. В этом случае конструктор массива записывается так же с ключевым словом ARRAY, за которым в круглых скобках следует подзапрос, например:

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE  
'bytea%');
```

array

```
-----  
-----  
  
{2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,  
31,2412,2413}  
(1 row)
```

```
SELECT ARRAY(SELECT ARRAY[i, i*2] FROM
generate_series(1,5) AS a(i));

      array
```

```
-----
{{1,2},{2,4},{3,6},{4,8},{5,10}}
(1 row)
```

Такой подзапрос должен возвращать один столбец. Если этот столбец имеет тип, отличный от массива, результирующий одномерный массив будет включать элементы для каждой строки-результата подзапроса и типом элемента будет тип столбца результата. Если же тип столбца – массив, будет создан массив того же типа, но большей размерности; в любом случае во всех строках подзапроса должны выдаваться массивы одинаковой размерности, чтобы можно было получить прямоугольный результат.

Индексы массива, созданного конструктором ARRAY, всегда начинаются с одного. Подробнее о массивах вы узнаете в разделе 6.15.

2.2.13. Конструкторы табличных строк

Конструктор табличной строки – это выражение, создающее строку или кортеж (или составное значение) из значений его аргументов-полей. Конструктор строки состоит из ключевого слова ROW, открывающей круглой скобки, нуля или нескольких выражений (разделенных запятыми), определяющих значения полей, и закрывающей скобки, например:

```
SELECT ROW(1,2.5,'this is a test');
```

Если в списке более одного выражения, ключевое слово ROW можно опустить.

Конструктор строки поддерживает запись `составное_значение.*`, при этом данное значение будет развернуто в список элементов, так же, как в записи `.*` на верхнем уровне списка SELECT. Например, если таблица t содержит колонки f1 и f2, эти записи эквивалентны:

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

По умолчанию значение, созданное выражением ROW, имеет тип анонимной

записи. Если необходимо, его можно привести к именованному составному типу – либо к типу строки таблицы, либо составному типу, созданному оператором CREATE TYPE AS. Явное приведение может потребоваться для достижения однозначности, например:

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);
```

```
CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT  
$1.f1' LANGUAGE SQL;
```

```
-- Приведение не требуется, так как существует только  
одна getf1()
```

```
SELECT getf1(ROW(1,2.5,'this is a test'));
```

```
getf1  
-----  
1  
(1 row)
```

```
CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);
```

```
CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT  
$1.f1' LANGUAGE SQL;
```

```
-- Теперь приведение необходимо для однозначного выбора  
функции:
```

```
SELECT getf1(ROW(1,2.5,'this is a test'));
```

```
ОШИБКА: функция getf1(record) не уникальна
```

```
SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
```

```
getf1  
-----  
1  
(1 row)
```

```
SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS
```

```
myrowtype)) ;  
  getf1  
-----  
      11  
(1 row)
```

Используя конструктор строк (кортежей), можно создавать составное значение для сохранения в колонке составного типа или для передачи функции, принимающей составной параметр. Также вы можете сравнить два составных значения или проверить их с помощью IS NULL или IS NOT NULL, например:

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the  
same') ;
```

```
-- выбрать все строки, содержащие только NULL  
SELECT ROW(table.*) IS NULL FROM table;
```

2.2.14. Правила вычисления выражений

Порядок вычисления подвыражений не определен. В частности, аргументы оператора или функции не обязательно вычисляются слева направо или в любом другом фиксированном порядке.

Более того, если результат выражения можно получить, вычисляя только некоторые его части, тогда другие подвыражения не будут вычисляться вовсе. Например, если написать:

```
SELECT true OR somefunc() ;
```

тогда функция somefunc() не будет вызываться (возможно). То же самое справедливо для записи:

```
SELECT somefunc() OR true;
```

Заметьте, что это отличается от «оптимизации» вычисления логических операторов слева направо, реализованной в некоторых языках программирования.

Как следствие, в сложных выражениях не стоит использовать функции с побочными эффектами. Особенно опасно рассчитывать на порядок вычисления или побочные эффекты в предложениях WHERE и HAVING, так как эти предложения тщательно оптимизируются при построении плана выполнения. Логические

выражения (сочетания AND/OR/NOT) в этих предложениях могут быть видоизменены любым способом, допустимым законами Булевой алгебры.

Когда порядок вычисления важен, его можно зафиксировать с помощью конструкции CASE. Например, такой способ избежать деления на ноль в предложении WHERE ненадежен:

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

Безопасный вариант:

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE  
false END;
```

Применяемая так конструкция CASE защищает выражение от оптимизации, поэтому использовать её нужно только при необходимости. В данном случае было бы лучше решить проблему, переписав условие как $y > 1.5 * x$.

Однако, CASE не всегда спасает в подобных случаях. Показанный выше прием плох тем, что не предотвращает раннее вычисление константных подвыражений. Функции и операторы, помеченные как IMMUTABLE, могут вычисляться при планировании, а не выполнении запроса. Поэтому в примере

```
SELECT CASE WHEN x > 0 THEN x ELSE 1/0 END FROM tab;
```

скорее всего, произойдет деление на ноль из-за того, что планировщик попытается упростить константное подвыражение, даже если во всех строках в таблице $x > 0$, а значит во время выполнения ветвь ELSE никогда не будет выполняться.

Хотя этот конкретный пример может показаться надуманным, похожие ситуации, в которых неявно появляются константы, могут возникать и в запросах внутри функций, так как значения аргументов функции и локальных переменных при планировании могут быть заменены константами. Поэтому например, в функциях PL/pgSQL гораздо безопаснее для защиты от рискованных вычислений использовать конструкцию IF-THEN-ELSE, чем выражение CASE.

Ещё один подобный недостаток этого подхода в том, что CASE не может предотвратить вычисление заключенного в нем агрегатного выражения, так как

агрегатные выражения вычисляются перед всеми остальными в списке `SELECT` или предложении `HAVING`. Например, в следующем запросе может возникнуть ошибка деления на ноль, несмотря на то, что он вроде бы защищен от нее:

```
SELECT CASE WHEN min(employees) > 0
           THEN avg(expenses / employees)
           END
FROM departments;
```

Агрегатные функции `min()` и `avg()` вычисляются независимо по всем входным строкам, так что если в какой-то строке поле `employees` окажется равным нулю, деление на ноль произойдет раньше, чем станет возможным проверить результат функции `min()`. Поэтому, чтобы проблемные входные строки изначально не попали в агрегатную функцию, следует воспользоваться предложениями `WHERE` или `FILTER`.

2.3. Вызов функций

Синергия-БД позволяет вызывать функции с именованными параметрами, используя запись с позиционной или именной передачей аргументов. Именная передача особенно полезна для функций со множеством параметров, так как она делает связь параметров и аргументов более явной и надежной. В позиционной записи значения аргументов функции указываются в том же порядке, в каком они описаны в определении функции. При именной передаче аргументы сопоставляются с параметрами функции по именам и указывать их можно в любом порядке.

Для каждого варианта вызова также учитывайте влияние типов аргументов функций.

При записи любым способом параметры, для которых в определении функции заданы значения по умолчанию, можно вовсе не указывать. Но это особенно полезно при именной передаче, так как опустить можно любой набор параметров, тогда как при позиционной параметры можно опускать только последовательно, справа налево.

Синергия-БД также поддерживает смешанную передачу, когда параметры

передаются и по именам, и по позиции. В этом случае позиционные параметры должны идти перед параметрами, передаваемыми по именам.

Мы рассмотрим все три варианта записи на примере следующей функции:

```
CREATE FUNCTION concat_lower_or_upper(a text, b text,  
    uppercase boolean DEFAULT false)  
RETURNS text  
AS  
$$  
    SELECT CASE  
        WHEN $3 THEN UPPER($1 || ' ' || $2)  
        ELSE LOWER($1 || ' ' || $2)  
    END;  
$$  
LANGUAGE SQL IMMUTABLE STRICT;
```

Функция `concat_lower_or_upper` имеет два обязательных параметра: `a` и `b`. Кроме того, есть один необязательный параметр `uppercase`, который по умолчанию имеет значение `false`. Аргументы `a` и `b` будут сложены вместе и переведены в верхний или нижний регистр, в зависимости от параметра `uppercase`. Остальные тонкости реализации функции сейчас не важны.

2.3.1. Позиционная передача

Позиционная передача – это традиционный механизм передачи аргументов функции в Синергия-БД. Пример такой записи:

```
SELECT concat_lower_or_upper('Hello', 'World', true);  
concat_lower_or_upper  
-----  
HELLO WORLD  
(1 row)
```

Все аргументы указаны в заданном порядке. Результат возвращен в верхнем регистре, так как параметр `uppercase` имеет значение `true`. Ещё один пример:

```
SELECT concat_lower_or_upper('Hello', 'World');  
concat_lower_or_upper
```

```
-----  
hello world  
(1 row)
```

Здесь параметр `uppercase` опущен, и поэтому он принимает значение по умолчанию (`false`) и результат переводится в нижний регистр. В позиционной записи любые аргументы с определенным значением по умолчанию можно опускать справа налево.

2.3.2. Именная передача

При именной передаче для аргумента добавляется имя, которое отделяется от выражения значения знаками `:=`, например:

```
SELECT concat_lower_or_upper(a => 'Hello', b =>  
'World');  
concat_lower_or_upper  
-----  
hello world  
(1 row)
```

Здесь аргумент `uppercase` был так же опущен, так что он неявно получил значение `false`. Преимуществом такой записи является возможность записывать аргументы в любом порядке, например:

```
SELECT concat_lower_or_upper(a := 'Hello', b := 'World',  
uppercase := true);  
concat_lower_or_upper  
-----  
HELLO WORLD  
(1 row)  
SELECT concat_lower_or_upper(a := 'Hello', uppercase :=  
true, b := 'World');  
concat_lower_or_upper  
-----  
HELLO WORLD  
(1 row)
```


Для обратной совместимости поддерживается и старый синтаксис с ":=":

```
SELECT concat_lower_or_upper(a := 'Hello', uppercase :=
true, b := 'World');
  concat_lower_or_upper
-----
HELLO WORLD
(1 row)
```

2.3.3. Смешанная передача

При смешанной передаче параметры передаются и по именам, и по позиции. Однако, как уже было сказано, именованные аргументы не могут стоять перед позиционными, например:

```
SELECT concat_lower_or_upper('Hello', 'World',
uppercase := true);
  concat_lower_or_upper
-----
HELLO WORLD
(1 row)
```

В данном запросе аргументы `a` и `b` передаются по позиции, а `uppercase` – по имени. Единственное обоснование такого вызова здесь – он стал чуть более читаемым. Однако для более сложных функций с множеством аргументов, часть из которых имеют значения по умолчанию, именная или смешанная передача позволяют записать вызов эффективнее и уменьшить вероятность ошибок.

Примечание. Именная и смешанная передача в настоящий момент не может использоваться при вызове агрегатной функции (но они допускаются, если агрегатная функция используется в качестве оконной).

3. ОПРЕДЕЛЕНИЕ ДАННЫХ

Эта глава рассказывает, как создавать структуры базы данных, в которых будут храниться данные. В реляционной базе данных данные хранятся в таблицах, так что большая часть этой главы будет посвящена созданию и изменению таблиц, а также средствам управления данными в них. Затем мы обсудим, как таблицы можно объединять в схемы и как ограничивать доступ к ним. Наконец, мы кратко рассмотрим другие возможности, связанные с хранением данных, в частности наследование, секционирование таблиц, представления, функции и триггеры.

3.1. Основы таблиц

Таблица в реляционной базе данных похожа на таблицу на бумаге: она так же состоит из строк и колонок. Число и порядок колонок фиксированы, а каждая колонка имеет имя. Число строк переменное – оно отражает текущее количество находящихся в ней данных. SQL не дает никаких гарантий относительно порядка строк таблицы. При чтении таблицы строки выводятся в произвольном порядке, если только явно не требуется сортировка. Подробнее это рассматривается в главе 5. Более того, SQL не назначает строкам уникальные идентификаторы, так что можно иметь в таблице несколько полностью идентичных строк. Это вытекает из математической модели, которую реализует SQL, но обычно такое дублирование нежелательно. Позже в этой главе мы увидим, как его избежать.

Каждой колонке сопоставлен тип данных. Тип данных ограничивает набор допустимых значений, которые можно присвоить колонке, и определяет смысловое значение данных для вычислений. Например, в колонку числового типа нельзя записать обычные текстовые строки, но зато её данные можно использовать в математических вычислениях. И наоборот, если колонка имеет тип текстовой строки, для нее допустимы практически любые данные, но она непригодна для математических действий (хотя другие операции, например конкатенация строк, возможны).

В Синергия-БД есть внушительный набор встроенных типов данных, удовлетворяющий большинство приложений. Пользователи также могут определять

собственные типы данных. Большинство встроенных типов данных имеют понятные имена и семантику, так что мы отложим их подробное рассмотрение до главы 6. Наиболее часто применяются следующие типы данных: `integer` для целых чисел, `numeric` для чисел, которые могут быть дробными, `text` для текстовых строк, `date` для дат, `time` для времени и `timestamp` для значений, включающих дату и время.

Для создания таблицы используется команда `CREATE TABLE`. В этой команде вы должны указать как минимум имя новой таблицы и имена и типы данных каждой колонки, например:

```
CREATE TABLE my_first_table (  
    first_column text,  
    second_column integer  
);
```

Так вы создадите таблицу `my_first_table` с двумя колонками. Первая колонка называется `first_column` и имеет тип данных `text`; вторая колонка называется `second_column` и имеет тип `integer`. Имена таблицы и колонок соответствуют синтаксису идентификаторов, описанному в подразделе 2.1.1. Имена типов также являются идентификаторами, хотя есть некоторые исключения. Заметьте, что список колонок заключается в скобки, а его элементы разделяются запятыми.

Конечно, предыдущий пример ненатурален. Обычно в именах таблиц и колонок отражается, какие данные они будут содержать. Поэтому давайте взглянем на более реалистичный пример:

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric  
);
```

Тип `numeric` может хранить дробные числа, в которых обычно выражаются денежные суммы.

Примечание. Когда создается много взаимосвязанных таблиц, имеет смысл заранее выбрать единый шаблон именования таблиц и колонок. Например, решить,

будут ли в именах таблиц использоваться существительные во множественном или в единственном числе (есть соображения в пользу каждого варианта).

Число колонок в таблице не может быть бесконечным. Это число ограничивается максимумом в пределах от 250 до 1600, в зависимости от типов колонок. Однако создавать таблицы с таким большим числом колонок обычно не требуется, а если такая потребность возникает, это скорее признак сомнительного дизайна.

Если таблица вам больше не нужна, вы можете удалить ее, выполнив команду `DROP TABLE`, например:

```
DROP TABLE my_first_table;  
DROP TABLE products;
```

Попытка удаления несуществующей таблицы считается ошибкой. Тем не менее в SQL-скриптах часто применяют безусловное удаление таблиц перед созданием, игнорируя все сообщения об ошибках, так что они выполняют свою задачу независимо от того, существовали таблицы или нет. Если вы хотите избежать таких ошибок, можно использовать вариант `DROP TABLE IF EXISTS`, но это не будет соответствовать стандарту SQL.

Как изменить существующую таблицу, будет рассмотрено в этой главе позже, в разделе 3.5.

Имея средства, которые мы обсудили, вы уже можете создавать полностью функциональные таблицы. В продолжении этой главы рассматриваются дополнительные возможности, призванные обеспечить целостность данных, безопасность и удобство. Если вам не терпится наполнить свои таблицы данными, вы можете вернуться к этой главе позже, а сейчас перейти к главе 4.

3.2. Значения по умолчанию

Колонке можно назначить значение по умолчанию. Когда добавляется новая строка и каким-то её колонкам не присваиваются значения, эти колонки принимают значения по умолчанию. Также команда управления данными может явно указать, что колонке должно быть присвоено значение по умолчанию, не зная его (подробнее команды управления данными описаны в главе 4).

Если значение по умолчанию не объявлено явно, им считается значение NULL. Обычно это имеет смысл, так как можно считать, что NULL вводимая строка. Допустимые типы Описание NULL представляет неизвестные данные.

В определении таблицы значения по умолчанию указываются после типа данных колонки, например:

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric DEFAULT 9.99  
);
```

Значение по умолчанию может быть выражением, которое в этом случае вычисляется в момент присваивания значения по умолчанию (а не когда создается таблица). Например, колонке timestamp в качестве значения по умолчанию часто присваивается CURRENT_TIMESTAMP, чтобы в момент добавления строки в ней оказалось текущее время. Ещё один распространенный пример – генерация «последовательных номеров» для всех строк. В Синергия-БД это обычно делается примерно так:

```
CREATE TABLE products (  
    product_no integer DEFAULT  
    nextval('products_product_no_seq'),  
    ...  
);
```

здесь функция nextval() выбирает очередное значение из последовательности. Это употребление настолько распространено, что для него есть специальная короткая запись:

```
CREATE TABLE products (  
    product_no SERIAL,  
    ...  
);
```

SERIAL обсуждается позже в подразделе 6.1.4.

3.3. Ограничения

Типы данных сами по себе ограничивают множество данных, которые можно сохранить в таблице. Однако для многих приложений такие ограничения слишком грубые. Например, колонка, содержащая цену продукта, должна, вероятно, принимать только положительные значения. Но такого стандартного типа данных нет. Возможно, вы также захотите ограничить данные колонки по отношению к другим колонкам или строкам. Например, в таблице с информацией о товаре должна быть только одна строка с определенным кодом товара.

Для решения подобных задач SQL позволяет вам определять ограничения для колонок и таблиц. Ограничения дают вам возможность управлять данными в таблицах так, как вы захотите. Если пользователь попытается сохранить в колонке значение, нарушающее ограничения, возникнет ошибка. Ограничения будут действовать, даже если это значение по умолчанию.

3.3.1. Ограничения-проверки

Ограничение-проверка – наиболее общий тип ограничений. В его определении вы можете указать, что значение данной колонки должно удовлетворять логическому выражению (проверке истинности). Например, цену товара можно ограничить положительными значениями так:

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CHECK (price > 0)  
);
```

Как вы видите, ограничение определяется после типа данных, как и значение по умолчанию. Значения по умолчанию и ограничения могут указываться в любом порядке. Ограничение-проверка состоит из ключевого слова CHECK, за которым идет выражение в скобках. Это выражение должно включать колонку, для которой задается ограничение, иначе оно не имеет большого смысла.

Вы можете также присвоить ограничению отдельное имя. Это улучшит сообщения об ошибках и позволит вам ссылаться на это ограничение, когда вам

понадобится изменить его. Сделать это можно так:

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CONSTRAINT positive_price CHECK  
    (price > 0)  
);
```

То есть, чтобы создать именованное ограничение, напишите ключевое слово `CONSTRAINT`, а за ним идентификатор и собственно определение ограничения (если вы не определите имя ограничения таким образом, система выберет для него имя за вас).

Ограничение-проверка может также ссылаться на несколько колонок. Например, если вы храните обычную цену и цену со скидкой, так вы можете гарантировать, что цена со скидкой будет всегда меньше обычной:

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CHECK (price > 0),  
    discounted_price numeric CHECK (discounted_price >  
0),  
    CHECK (price > discounted_price)  
);
```

Первые два ограничения определяются похожим образом, но для третьего используется новый синтаксис. Оно не связано с определенной колонкой, а представлено отдельным элементом в списке. Определения колонок и такие определения ограничений можно переставлять в произвольном порядке.

Про первые два ограничения можно сказать, что это ограничения колонок, тогда как третье является ограничением таблицы, так как оно написано отдельно от определений колонок. Ограничения колонок также можно записать в виде ограничений таблицы, тогда как обратное не всегда возможно, так как подразумевается, что ограничение колонки ссылается только на связанную колонку (хотя Синергия-БД этого не требует, но для совместимости с другими СУБД лучше

следовать это правилу). Ранее приведенный пример можно переписать и так:

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric,  
    CHECK (price > 0),  
    discounted_price numeric,  
    CHECK (discounted_price > 0),  
    CHECK (price > discounted_price)  
);
```

Или даже так:

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CHECK (price > 0),  
    discounted_price numeric,  
    CHECK (discounted_price > 0 AND price >  
discounted_price)  
);
```

Ограничениям таблицы можно присваивать имена так же, как и ограничениям колонок:

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric,  
    CHECK (price > 0),  
    discounted_price numeric,  
    CHECK (discounted_price > 0),  
    CONSTRAINT valid_discount CHECK (price >  
discounted_price)  
);
```

Следует заметить, что ограничение-проверка удовлетворяется, если выражение принимает значение true или NULL. Так как результатом многих

выражений с операндами NULL будет значение NULL, такие ограничения не будут препятствовать записи NULL в связанные колонки. Чтобы гарантировать, что колонка не содержит значения NULL, можно использовать ограничение NOT NULL, описанное в следующем разделе.

3.3.2. Ограничения NOT NULL

Ограничение NOT NULL просто указывает, что колонке нельзя присваивать значение NULL. Пример синтаксиса:

```
CREATE TABLE products (  
    product_no integer NOT NULL,  
    name text NOT NULL,  
    price numeric  
);
```

Ограничение NOT NULL всегда записывается как ограничение колонки и функционально эквивалентно ограничению CHECK (имя_колонки IS NOT NULL), но в Синергия-БД явное ограничение NOT NULL работает более эффективно. Хотя у такой записи есть недостаток – назначить имя таким ограничениям нельзя.

Естественно, для колонки можно определить больше одного ограничения. Для этого их нужно просто указать одно за другим:

```
CREATE TABLE products (  
    product_no integer NOT NULL,  
    name text NOT NULL,  
    price numeric NOT NULL CHECK (price > 0)  
);
```

Порядок здесь не имеет значения, он не обязательно соответствует порядку проверки ограничений.

Для ограничения NOT NULL есть и обратное: ограничение NULL. Это не означает, что колонка должна иметь только значение NULL, что конечно было бы бессмысленно. Суть же его в простом указании, что колонка может иметь значение NULL (это поведение по умолчанию). Ограничение NULL отсутствует в стандарте SQL и использовать его в переносимых приложениях не следует (оно было

добавлено в Синергия-БД только для совместимости с некоторыми другими СУБД). Однако некоторые пользователи любят его использовать, так как оно позволяет легко переключать ограничения в скрипте. Например, вы можете начать с:

```
CREATE TABLE products (  
    product_no integer NULL,  
    name text NULL,  
    price numeric NULL  
);
```

и затем вставить ключевое слово NOT, где потребуется.

Примечание. При проектировании баз данных чаще всего большинство колонок должны быть помечены как NOT NULL.

3.3.3. Ограничения уникальности

Ограничения уникальности гарантируют, что данные в определенной колонке или группе колонок уникальны для всех строк таблицы. Ограничение записывается так:

```
CREATE TABLE products (  
    product_no integer UNIQUE,  
    name text,  
    price numeric  
);
```

в виде ограничения колонки и так:

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric,  
    UNIQUE (product_no)  
);
```

в виде ограничения таблицы.

Если ограничение уникальности определяется для группы колонок, колонки перечисляются через запятую:

```
CREATE TABLE example (  
    a integer,
```

```
b integer,  
c integer,  
UNIQUE (a, c)  
);
```

Такое ограничение указывает, что сочетание значений перечисленных колонок должно быть уникально во всей таблице, тогда как значения каждой колонки по отдельности не должны быть (и обычно не будут) уникальными.

Вы можете назначить уникальному ограничению имя обычным образом:

```
CREATE TABLE products (  
    product_no integer CONSTRAINT must_be_different  
    UNIQUE,  
    name text,  
    price numeric  
);
```

При добавлении ограничения уникальности будет автоматически создан уникальный индекс В-дерева для колонки или группы колонок, задействованных в ограничении. Ограничение уникальности для только некоторых строк можно ввести, создав частичный индекс.

Вообще говоря, ограничение уникальности нарушается, когда в таблице оказывается несколько строк, у которых совпадают значения всех колонок, включенных в ограничение. Однако два значения NULL при сравнении не считаются равными. Это означает, что даже при наличии ограничения уникальности в таблице можно сохранить строки с дублирующимися значениями, если они содержат NULL в одной или нескольких колонках ограничения. Это поведение соответствует стандарту SQL, но мы слышали о СУБД, которые ведут себя по-другому. Имейте в виду эту особенность, разрабатывая переносимые приложения.

3.3.4. Первичные ключи

Ограничение первичного ключа означает, что образующий его столбец или группа столбцов может быть уникальным идентификатором строк в таблице. Для этого требуется, чтобы значения были одновременно уникальными и отличными от NULL. Таким образом, таблицы со следующими двумя определениями будут

принимать одинаковые данные:

```
CREATE TABLE products (  
    product_no integer UNIQUE NOT NULL,  
    name text,  
    price numeric  
);  
  
CREATE TABLE products (  
    product_no integer PRIMARY KEY,  
    name text,  
    price numeric  
);
```

Первичные ключи, как и ограничения уникальности, могут включать несколько колонок и записывается это так же:

```
CREATE TABLE example (  
    a integer,  
    b integer,  
    c integer,  
    PRIMARY KEY (a, c)  
);
```

Первичный ключ выбирает колонку или группу колонок, которую можно использовать как уникальный идентификатор строк таблицы (это прямое следствие определения первичного ключа. Заметьте, что ограничение уникальности само по себе не является таким уникальным идентификатором, так как оно не исключает значения NULL). Это полезно и для документирования, и для клиентских приложений. Например, графическому приложению с возможностями редактирования содержимого таблицы, вероятно, потребуется знать первичный ключ таблицы, чтобы однозначно идентифицировать её строки.

При добавлении первичного ключа автоматически создаётся уникальный индекс-B-дерево для столбца или группы столбцов, перечисленных в первичном ключе, и данные столбцы помечаются как NOT NULL.

Таблица может иметь максимум один первичный ключ (ограничений уникальности и ограничений NOT NULL, которые функционально почти

равнозначны первичным ключам, может быть сколько угодно, но назначить ограничением первичного ключа можно только одно). Теория реляционных баз данных говорит, что первичный ключ должен быть в каждой таблице. В Синергия-БД такого жёсткого требования нет, но обычно лучше ему следовать.

Первичные ключи полезны и для документирования, и для клиентских приложений. Например, графическому приложению с возможностями редактирования содержимого таблицы, вероятно, потребуется знать первичный ключ таблицы, чтобы однозначно идентифицировать её строки. Первичные ключи находят и другое применение в СУБД; в частности, первичный ключ в таблице определяет целевые столбцы по умолчанию для сторонних ключей, ссылающихся на эту таблицу.

3.3.5. Внешние ключи

Ограничение внешнего ключа указывает, что значения колонки (или группы колонок) должны соответствовать значениям в некоторой строке другой таблицы. Это называется ссылочной целостностью двух связанных таблиц.

Пусть у вас уже есть таблица продуктов, которую мы неоднократно использовали ранее:

```
CREATE TABLE products (  
    product_no integer PRIMARY KEY,  
    name text,  
    price numeric  
);
```

Давайте предположим, что у вас есть таблица с заказами этих продуктов. Мы хотим, чтобы в таблице заказов содержались только заказы действительно существующих продуктов. Поэтому мы определим в ней ограничение внешнего ключа, ссылающееся на таблицу продуктов:

```
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    product_no integer REFERENCES products (product_no),  
    quantity integer
```

```
);
```

С таким ограничением создать заказ со значением `product_no`, отсутствующим в таблице `products` (и не равным `NULL`), будет невозможно.

В такой схеме таблицу `orders` называют подчиненной таблицей, а `products` – главной. Соответственно, колонки называют так же подчиненной и главной (или ссылающейся и целевой).

Предыдущую команду можно сократить так:

```
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    product_no integer REFERENCES products,  
    quantity integer  
);
```

то есть, если опустить список колонок, внешний ключ будет неявно связан с первичным ключом главной таблицы.

Внешний ключ также может ссылаться на группу колонок. В этом случае его нужно записать в виде обычного ограничения таблицы, например:

```
CREATE TABLE t1 (  
    a integer PRIMARY KEY,  
    b integer,  
    c integer,  
    FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)  
);
```

Естественно, число и типы колонок в ограничении должны соответствовать числу и типам целевых колонок.

Ограничению внешнего ключа можно назначить имя стандартным способом.

Таблица может содержать несколько ограничений внешнего ключа. Это полезно для связи таблиц в отношении многие-ко-многим. Скажем, у вас есть таблицы продуктов и заказов, но вы хотите, чтобы один заказ мог содержать несколько продуктов (что невозможно в предыдущей схеме). Для этого вы можете использовать такую схему:

```
CREATE TABLE products (  

```

```
product_no integer PRIMARY KEY,  
name text,  
price numeric  
);
```

```
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    shipping_address text,  
    ...  
);
```

```
CREATE TABLE order_items (  
    product_no integer REFERENCES products,  
    order_id integer REFERENCES orders,  
    quantity integer,  
    PRIMARY KEY (product_no, order_id)  
);
```

Заметьте, что в последней таблице первичный ключ покрывает внешние ключи.

Мы знаем, что внешние ключи запрещают создание заказов, не относящихся ни к одному продукту. Но что делать, если после создания заказов с определенным продуктом мы захотим удалить его? SQL справится с этой ситуацией. Интуиция подсказывает следующие варианты поведения:

- запретить удаление продукта;
- удалить также связанные заказы;
- что-то еще?.

Для иллюстрации давайте реализуем следующее поведение в вышеприведенном примере: при попытке удаления продукта, на который ссылаются заказы (через таблицу `order_items`), мы запрещаем эту операцию. Если же кто-то попытается удалить заказ, то удалится и его содержимое:

```
CREATE TABLE products (  
    product_no integer PRIMARY KEY,
```

```
    name text,  
    price numeric  
);
```

```
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    shipping_address text,  
    ...  
);
```

```
CREATE TABLE order_items (  
    product_no integer REFERENCES products ON DELETE  
RESTRICT,  
    order_id integer REFERENCES orders ON DELETE  
CASCADE,  
    quantity integer,  
    PRIMARY KEY (product_no, order_id)  
);
```

Ограничивающие и каскадные удаления – два наиболее распространенных варианта. RESTRICT предотвращает удаление связанной строки. NO ACTION означает, что если зависимые строки продолжают существовать при проверке ограничения, возникает ошибка (это поведение по умолчанию. Главным отличием этих двух вариантов является то, что NO ACTION позволяет отложить проверку в процессе транзакции, а RESTRICT – нет). CASCADE указывает, что при удалении связанных строк зависимые от них будут так же автоматически удалены. Есть ещё два варианта: SET NULL и SET DEFAULT. При удалении связанных строк они назначают зависимым колонкам в подчиненной таблице значения NULL или значения по умолчанию, соответственно. Заметьте, что это не будет основанием для нарушения ограничений. Например, если в качестве действия задано SET DEFAULT, но значение по умолчанию не удовлетворяет ограничению внешнего ключа, операция закончится ошибкой.

Аналогично указанию ON DELETE существует ON UPDATE, которое

срабатывает при изменении заданной колонки. При этом возможные действия те же, а CASCADE в данном случае означает, что измененные значения связанных колонок будут скопированы в зависимые строки.

Обычно зависимая строка не должна удовлетворять ограничению внешнего ключа, если одна из связанных колонок содержит NULL. Если в объявление внешнего ключа добавлено MATCH FULL, строка будет удовлетворять ограничению, только если все связанные колонки равны NULL (то есть при разных значениях (NULL и не NULL) гарантируется невыполнение ограничения MATCH FULL). Если вы хотите, чтобы зависимые строки не могли избежать и этого ограничения, объявите связанные колонки как NOT NULL.

Внешний ключ должен ссылаться на колонки, образующие первичный ключ или ограничение уникальности. Таким образом, для связанных колонок всегда будет существовать индекс (определенный соответствующим первичным ключом или ограничением), а значит проверки соответствия связанной строки будут выполняться эффективно. Так как команды DELETE для строк главной таблицы или UPDATE для зависимых колонок потребуют просканировать подчиненную таблицу и найти строки, ссылающиеся на старые значения, полезно будет иметь индекс и для подчиненных колонок. Но это нужно не всегда и создать соответствующий индекс можно по-разному, поэтому объявление внешнего ключа не создает автоматически индекс по связанным колонкам.

Подробнее об изменении и удалении данных рассказывается в главе 4. Вы также можете подробнее узнать о синтаксисе ограничений внешнего ключа в справке CREATE TABLE.

3.3.6. Ограничения-исключения

Ограничения-исключения гарантируют, что при сравнении любых двух строк по указанным колонкам или выражениям с помощью заданных операторов, минимум одно из этих сравнений возвратит false или NULL. Записывается это так:

```
CREATE TABLE circles (  
    c circle,
```

```
EXCLUDE USING gist (c WITH &&)
);
```

При добавлении ограничения-исключения будет автоматически создан индекс того типа, который указан в объявлении ограничения.

3.4. Системные колонки

В каждой таблице есть несколько системных колонок, неявно определенных системой. Как следствие, их имена нельзя использовать в качестве имен пользовательских колонок (заметьте, что это не зависит от того, является ли имя ключевым словом или нет; заключение имени в кавычки не поможет избежать этого ограничения). Эти колонки не должны вас беспокоить, вам лишь достаточно знать об их существовании.

`oid`

Идентификатор объекта (object ID) для строки. Эта колонка присутствует, только если таблица была создана с указанием `WITH OIDS` или если в момент её создания была установлена переменная конфигурации `default_with_oids`. Эта колонка имеет тип `oid` (с тем же именем, что и сама колонка); подробнее об этом типе см. раздел 6.18.

`tableoid`

Идентификатор объекта для таблицы, содержащей строку. Эта колонка особенно полезна для запросов, имеющих дело с иерархией наследования (см. раздел 3.9), так как без нее сложно определить, из какой таблицы выбрана строка. Связав `tableoid` с колонкой `oid` в таблице `pg_class`, можно будет получить имя таблицы.

`xmin`

Идентификатор (код) транзакции, добавившей строку этой версии. Версия строки – это её индивидуальное состояние; при каждом изменении создается новая версия одной и той же логической строки.

`cmin`

Номер команды (начиная с нуля) внутри транзакции, добавившей строку.

`xmax`

Идентификатор транзакции, удалившей строку, или 0 для не удаленной версии строки. Значение этой колонки может быть ненулевым и для видимой версии строки. Это обычно означает, что удаляющая транзакция ещё не была зафиксирована или удаление было отменено.

`cmx`

Номер команды в удаляющей транзакции или ноль.

`maclabel`

Мандатная метка строки. Используется для разграничения доступа к строкам на основе мандатных ПРД.

`ctid`

Физическое расположение данной версии строки в таблице. Заметьте, что хотя по `ctid` можно очень быстро найти версию строки, значение `ctid` изменится при выполнении `VACUUM FULL`. Таким образом, `ctid` нельзя применять в качестве долгосрочного идентификатора строки. Для идентификации логических строк лучше использовать `OID` или даже дополнительный последовательный номер.

Коды `OID` представляют собой 32-битные значения и выбираются из единого для всей СУБД счетчика. В больших или долгоживущих базах данных этот счетчик может пойти по кругу. Таким образом, не рекомендуется рассчитывать на уникальность `OID`, если только вы не обеспечите её дополнительно. Если вам нужно идентифицировать строки таблицы, настоятельно рекомендуется использовать последовательности. Однако можно использовать и коды `OID`, при выполнении следующих условий:

1. Когда для идентификации строк таблиц применяется `OID`, в каждой такой таблице должно создаваться ограничение уникальности для колонки `OID`. Когда такое ограничение уникальности (или уникальный индекс) существует, система позаботится о том, чтобы `OID` новой строки не совпал с уже существующими. Конечно, это возможно, только если в таблице меньше 2³² (4 миллиардов) строк, а на практике таблицы должны быть гораздо меньше, иначе может пострадать производительность системы.

2. Никогда не следует рассчитывать, что `OID` будут уникальны среди всех таблиц; в качестве глобального идентификатора в рамках базы данных используйте комбинацию `tableoid` и `OID` строки.

3. Конечно, все эти таблицы должны быть созданы с указанием `WITH OIDS`.

Идентификаторы транзакций также являются 32-битными. В долгоживущей базе данных они могут пойти по кругу. Это не критично при правильном обслуживании БД. Однако полагаться на уникальность кодов транзакций в течение длительного времени (при более чем миллиарде транзакций) не следует.

Идентификаторы команд также 32-битные. Это создает жесткий лимит на 232 (4 миллиарда) команд SQL в одной транзакции. На практике это не проблема – заметьте, что это лимит числа команд SQL, а не количества обрабатываемых строк. Кроме того, идентификатор получают только те команды, которые фактически изменяют содержимое базы данных.

3.5. Изменение таблиц

Если вы создали таблицы, а затем поняли, что допустили ошибку, или изменились требования вашего приложения, вы можете удалить её и создать заново. Но это будет неудобно, если таблица уже заполнена данными или если на нее ссылаются другие объекты базы данных (например, по внешнему ключу). Поэтому Синергия-БД предоставляет набор команд для модификации таблиц. Заметьте, что это по сути отличается от изменения данных, содержащихся в таблице: здесь мы обсуждаем модификацию определения, или структуры, таблицы.

Вы можете:

- добавлять колонки;
- удалять колонки;
- добавлять ограничения;
- удалять ограничения;
- изменять значения по умолчанию;
- изменять типы колонок;
- переименовывать колонки;

- переименовывать таблицы.

Все эти действия выполняются с помощью команды `ALTER TABLE`; подробнее о ней вы можете узнать в её справке.

3.5.1. Добавление колонки

Добавить колонку вы можете так:

```
ALTER TABLE products ADD COLUMN description text;
```

Новая колонка заполняется заданным для нее значением по умолчанию (или значением `NULL`, если вы не добавите указание `DEFAULT`).

При этом вы можете сразу определить ограничения колонки, используя обычный синтаксис:

```
ALTER TABLE products ADD COLUMN description text CHECK  
(description <> '');
```

На самом деле здесь можно использовать все конструкции, допустимые в определении колонки в команде `CREATE TABLE`. Помните однако, что значение по умолчанию должно удовлетворять данным ограничениям, чтобы операция `ADD` выполнялась успешно. Вы также можете сначала заполнить колонку правильно, а затем добавить ограничения.

Примечание. Добавление колонки со значением по умолчанию приводит к изменению всех строк таблицы (в них будет сохранено новое значение). Однако, если значение по умолчанию не указано, Синергия-БД может обойтись без физического изменения. Поэтому, если вы планируете заполнить колонку в основном не значениями по умолчанию, лучше будет добавить колонку без значения по умолчанию, затем вставить требуемые значения с помощью `UPDATE`, а потом определить значение по умолчанию, как описано ниже.

3.5.2. Удаление колонки

Удалить колонку можно так:

```
ALTER TABLE products DROP COLUMN description;
```

Данные, которые были в этой колонке, исчезают. Вместе с колонкой удаляются и включающие её ограничения таблицы. Однако, если на колонку ссылается ограничение внешнего ключа другой таблицы, Синергия-БД не удалит это ограничение неявно. Разрешить удаление всех зависящих от этой колонки объектов

можно, добавив указание CASCADE:

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

Общий механизм, стоящий за этим, описывается в разделе 3.13.

3.5.3. Добавление ограничения

Для добавления ограничения используется синтаксис ограничения таблицы, например:

```
ALTER TABLE products ADD CHECK (name <> '');
```

```
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE  
(product_no);
```

```
ALTER TABLE products ADD FOREIGN KEY (product_group_id)  
REFERENCES product_groups;
```

Чтобы добавить ограничение NOT NULL, которое нельзя записать в виде ограничения таблицы, используйте такой синтаксис:

```
ALTER TABLE products ALTER COLUMN product_no SET NOT  
NULL;
```

Ограничение проходит проверку автоматически и будет добавлено, только если ему удовлетворяют данные таблицы.

3.5.4. Удаление ограничения

Для удаления ограничения вы должны знать его имя. Если вы не присваивали ему имя, это неявно сделала система, и вы должны выяснить его. Здесь может быть полезна команда `psql \d имя_таблицы` (или другие программы, показывающие подробную информацию о таблицах). Зная имя, вы можете использовать команду:

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

Если вы имеете дело с именем ограничения вида \$2, не забудьте заключить его в кавычки, чтобы это был допустимый идентификатор.

Как и при удалении колонки, если вы хотите удалить ограничение с зависимыми объектами, добавьте указание CASCADE. Примером такой зависимости может быть ограничение внешнего ключа, связанное с колонками ограничения первичного ключа.

Так можно удалить ограничения любых типов, кроме NOT NULL. Чтобы

удалить ограничение NOT NULL, используйте команду:

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

Вспомните, что у ограничений NOT NULL нет имен.

3.5.5. Изменение значения по умолчанию

Назначить колонке новое значение по умолчанию можно так:

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

Заметьте, что это никак не влияет на существующие строки таблицы, а просто задает значение по умолчанию для последующих команд INSERT.

Чтобы удалить значение по умолчанию, выполните:

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

При этом по сути значению по умолчанию просто присваивается NULL. Как следствие, ошибки не будет, если вы попытаетесь удалить значение по умолчанию, не определенное явно, так как неявно оно существует и равно NULL.

3.5.6. Изменение типа данных колонки

Чтобы преобразовать колонку в другой тип данных, используйте команду:

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

Она будет успешна, только если все существующие значения в колонке могут быть неявно приведены к новому типу. Если требуется более сложное преобразование, вы можете добавить указание USING, определяющее, как получить новые значения из старых.

Синергия-БД попытается также преобразовать к новому типу значение колонки по умолчанию (если оно определено) и все связанные с ней ограничения. Но преобразование может оказаться неправильным и тогда вы получите неожиданные результаты. Поэтому обычно лучше удалить все ограничения колонки, перед тем как менять её тип, а затем воссоздать модифицированные должным образом ограничения.

3.5.7. Переименование колонки

Чтобы переименовать колонку, выполните:

```
ALTER TABLE products RENAME COLUMN product_no TO  
product_number;
```

3.5.8. Переименование таблицы

Таблицу можно переименовать так:

```
ALTER TABLE products RENAME TO items;
```

3.6. Права

Когда в базе данных создается объект, ему назначается владелец. Владельцем обычно становится роль, с которой был выполнен оператор создания. Для большинства типов объектов в исходном состоянии только владелец (или суперпользователь) может делать с объектом все, что угодно. Чтобы разрешить использовать его другим ролям, нужно дать им права.

Существует несколько типов прав: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, CREATE, CONNECT, TEMPORARY, EXECUTE и USAGE. Набор прав, применимых к определенному объекту, зависит от типа объекта (таблица, функция и т. д.) Полную информацию о различных типах прав, поддерживаемых Синергия-БД, вы найдете на странице справки GRANT.

Неотъемлемое право изменять или удалять объект имеет только владелец объекта.

Объекту можно назначить нового владельца с помощью команды ALTER для соответствующего типа объекта, например ALTER TABLE. Суперпользователь может делать это без ограничений, а обычный пользователь, только если он является одновременно текущим владельцем объекта (или членом роли владельца) и членом новой роли.

Для назначения прав применяется команда GRANT. Например, если в базе данных есть роль joe и таблица accounts, право на изменение таблицы можно дать ему так:

```
GRANT UPDATE ON accounts TO joe;
```


Если вместо конкретного права написать ALL, роль получит все права, применимые для объекта этого типа.

Для назначения права всем роль в системе можно использовать специальное имя «пользователя»: PUBLIC. Также для упрощения управления ролями, когда в базе данных есть множество пользователей, можно настроить «групповые» роли.

Чтобы лишить пользователей прав, используйте команду REVOKE:

```
REVOKE ALL ON accounts FROM PUBLIC;
```

Особые права владельца объекта (то есть права на выполнение DROP, GRANT, REVOKE и т.д.) всегда неявно закреплены за владельцем и их нельзя назначить или отобрать. Но владелец объекта может лишить себя обычных прав, например, разрешить всем, включая себя, только чтение таблицы.

Обычно распоряжаться правами может только владелец объекта (или суперпользователь). Однако возможно дать право доступа к объекту «с правом передачи», что позволит получившему такое право назначать его другим. Если такое право передачи впоследствии будет отозвано, то все, кто получил данное право доступа (непосредственно или по цепочке передачи), потеряют его.

3.7. Схемы

Кластер баз данных Синергия-БД содержит один или несколько именованных экземпляров баз. На уровне кластера создаются пользователи и группы, но данные могут относиться только к базам данных. При этом в рамках одного подключения к серверу можно обращаться к данным только одной базы данных, указанной при установлении соединения.

Примечание: Пользователи кластера не обязательно будут иметь доступ ко всем базам данных этого кластера. То, что пользователи создаются на уровне кластера, означает только, что в нем не может быть двух ролей joe в разных базах данных, хотя система позволяет ограничить доступ joe только некоторыми базами данных.

База данных содержит одну или несколько именованных схем, которые в свою очередь содержат таблицы. Схемы также содержат именованные объекты других видов, включая типы данных, функции и операторы. Одно и то же имя объекта можно свободно использовать в разных схемах, например и schema1, и myschema

могут содержать таблицы с именем `mytable`. В отличие от баз данных, схемы не ограничивают доступ к данным: пользователь может обращаться к объектам в любой схеме текущей базы данных, если ему назначены соответствующие права.

Есть несколько возможных объяснений, для чего стоит применять схемы:

- чтобы одну базу данных могли использовать несколько пользователей, независимо друг от друга;
- чтобы объединить объекты базы данных в логические группы для облегчения управления ими;
- чтобы в одной базе сосуществовали разные приложения и при этом не возникало конфликтов имен.

Схемы в некотором смысле подобны каталогам в операционной системе, но они не могут быть вложенными.

3.7.1. Создание схемы

Для создания схемы используется команда `CREATE SCHEMA`. При этом вы определяете имя схемы по своему выбору, например:

```
CREATE SCHEMA myschema;
```

Чтобы создать объекты в схеме или обратиться к ним, указывайте полное имя, состоящее из имен схемы и объекта, разделенных точкой:

```
схема.таблица
```

Этот синтаксис работает везде, где ожидается имя таблицы, включая команды модификации таблицы и команды обработки данных, обсуждаемые в следующих главах (для краткости мы будем говорить только о таблицах, но все это распространяется и на другие типы именованных объектов, например, типы и функции).

Есть ещё более общий синтаксис:

```
база_данных.схема.таблица
```

но в настоящее время он поддерживается только для формального соответствия стандарту SQL. Если вы указываете базу данных, это может быть только база данных, к которой вы подключены.

Таким образом, создать таблицу в новой схеме можно так:

```
CREATE TABLE myschema.mytable (  
    ...  
);
```

Чтобы удалить пустую схему (не содержащую объектов), выполните:

```
DROP SCHEMA myschema;
```

Удалить схему со всеми содержащимися в ней объектами можно так:

```
DROP SCHEMA myschema CASCADE;
```

Стоящий за этим общий механизм описан в разделе 3.13.

Часто бывает нужно создать схему, владельцем которой будет другой пользователь (это один из способов ограничения пользователей пространствами имен). Сделать это можно так:

```
CREATE SCHEMA имя_схемы AUTHORIZATION имя_пользователя;
```

Вы даже можете опустить имя схемы, в этом случае именем схемы станет имя пользователя. Как это можно применять, описано в подразделе 3.8.6.

Схемы с именами, начинающимися с `pg_`, являются системными; пользователям не разрешено использовать такие имена.

3.7.2. Схема `public`

До этого мы создавали таблицы, не указывая никакие имена схем. По умолчанию такие таблицы (и другие объекты) автоматически помещаются в схему «`public`». Она содержится во всех создаваемых базах данных. Таким образом, команда:

```
CREATE TABLE products ( ... );
```

эквивалентна:

```
CREATE TABLE public.products ( ... );
```

3.7.3. Путь поиска схемы

Везде писать полные имена утомительно, и часто все равно лучше не привязывать приложения к конкретной схеме. Поэтому к таблицам обычно обращаются по неполному имени, состоящему просто из имени таблицы. Система определяет, какая именно таблица подразумевается, используя путь поиска, который

представляет собой список просматриваемых схем. Подразумеваемой таблицей считается первая подходящая таблица, найденная в схемах пути. Если подходящая таблица не найдена, возникает ошибка, даже если таблица с таким именем есть в других схемах базы данных.

Возможность создавать одноимённые объекты в разных схемах усложняет написание запросов, которые должны всегда обращаться к конкретным объектам. Это также потенциально позволяет пользователям влиять на поведение запросов других пользователей, злонамеренно или случайно. Ввиду преобладания неполных имён в запросах и их использования внутри Синергия-БД, добавить схему в `search_path` – по сути значит доверять всем пользователям, имеющим право `CREATE` в этой схеме. Когда вы выполняете обычный запрос, злонамеренный пользователь может создать объекты в схеме, включённой в ваш путь поиска, и таким образом перехватывать управление и выполнять произвольные функции SQL как если бы их выполняли вы.

Первая схема в пути поиска называется текущей. Эта схема будет использоваться не только при поиске, но и при создании объектов – она будет включать таблицы, созданные командой `CREATE TABLE` без указания схемы.

Чтобы узнать текущий тип поиска, выполните следующую команду:

```
SHOW search_path;
```

В конфигурации по умолчанию она возвращает:

```
search_path
-----
«$user»,public
```

Первый элемент ссылается на схему с именем текущего пользователя. Если такой схемы не существует, ссылка на нее игнорируется. Второй элемент ссылается на схему `public`, которую мы уже видели.

Первая существующая схема в пути поиска также считается схемой по умолчанию для новых объектов. Именно поэтому по умолчанию объекты создаются в схеме `public`. При указании неполной ссылки на объект в любом контексте (при модификации таблиц, изменении данных или в запросах) система просматривает

путь поиска, пока не найдет соответствующий объект. Таким образом, в конфигурации по умолчанию неполные имена могут относиться только к объектам в схеме `public`.

Чтобы добавить в путь нашу новую схему, мы выполняем:

```
SET search_path TO myschema,public;
```

Мы опускаем компонент `$user`, так как здесь в нем нет необходимости.

Теперь мы можем обращаться к таблице без указания схемы:

```
DROP TABLE mytable;
```

И так как `myschema` – первый элемент в пути, новые объекты будут по умолчанию создаваться в этой схеме.

Мы можем также написать:

```
SET search_path TO myschema;
```

Тогда мы больше не сможем обращаться к схеме `public`, не написав полное имя объекта. Единственное, что отличает схему `public` от других, это то, что она существует по умолчанию, хотя её так же можно удалить.

Как и для имен таблиц, путь поиска аналогично работает для имен типов данных, имен функций и имен операторов. Имена типов данных и функций можно записать в полном виде так же, как и имена таблиц. Если же вам нужно использовать в выражении полное имя оператора, для этого есть специальный способ – вы должны написать:

```
OPERATOR (схема.оператор)
```

Такая запись необходима для избежания синтаксической неоднозначности.

Пример такого выражения:

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

На практике пользователи часто полагаются на путь поиска, чтобы не приходилось писать такие замысловатые конструкции.

3.7.4. Схемы и права

По умолчанию пользователь не может обращаться к объектам в чужих схемах. Чтобы изменить это, владелец схемы должен дать пользователю право `USAGE` для данной схемы. Чтобы пользователи могли использовать объекты схемы, может

понадобиться назначить дополнительные права на уровне объектов.

Пользователю также можно разрешить создавать объекты в не принадлежащей ему схеме. Для этого ему нужно дать право CREATE в требуемой схеме. Заметьте, что по умолчанию все имеют права CREATE и USAGE в схеме public. Благодаря этому все пользователи могут подключаться к заданной базе данных и создавать объекты в её схеме public. Некоторые шаблоны использования требуют запретить это:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

Первое слово «public» обозначает схему, а второе «PUBLIC» подразумевает «все пользователи». В первом случае это идентификатор, а во втором – ключевое слово, поэтому оно написано в разном регистре; вспомните рекомендации из подраздела 2.1.1.

3.7.5. Схема системного каталога

В дополнение к схеме public и схемам, создаваемым пользователями, любая база данных содержит схему pg_catalog, в которой находятся системные таблицы и все встроенные типы данных, функции и операторы. pg_catalog фактически всегда является частью пути поиска. Если даже эта схема не добавлена в путь явно, она неявно просматривается до всех схем, указанных в пути. Так обеспечивается доступность встроенных имен при любых условиях. Однако вы можете явным образом поместить pg_catalog в конец пути поиска, если вам нужно, чтобы пользовательские имена переопределяли встроенные.

Так как имена системных таблиц начинаются с pg_, такие имена лучше не использовать во избежание конфликта имен, возможного при появлении в будущем системной таблицы с тем же именем, что и ваша. С путем поиска по умолчанию неполная ссылка будет воспринята как обращение к системной таблице. Системные таблицы будут и дальше содержать в имени приставку pg_, так что они не будут конфликтовать с неполными именами пользовательских таблиц, если пользователи со своей стороны не будут использовать приставку pg_.

3.7.6. Шаблоны использования

Схемам можно найти множество применений. Хотя есть несколько шаблонов использования, легко поддерживаемых стандартной конфигурацией, только один из них достаточно безопасен, когда одни пользователи базы данных не доверяют другим:

1. Ограничить обычных пользователей личными схемами. Для реализации этого подхода выполните `REVOKE CREATE ON SCHEMA public FROM PUBLIC` и создайте для каждого пользователя схему с его именем. Если затрагиваемые пользователи подключались к базе ранее, проведите аудит схемы на предмет наличия таких же имён, как в схеме `pg_catalog`. Вспомните, что путь поиска по умолчанию начинается со значения `$user`, которое разрешается в имя пользователя. Таким образом, если у всех пользователей будет отдельная схема, они по умолчанию будут обращаться к собственным схемам.
2. Удалить схему `public` из пути поиска по умолчанию для каждого пользователя с помощью команды `ALTER ROLE пользователь SET search_path = "$user"`. Все сохраняют возможность создавать объекты в общедоступной схеме, но обращаться к ним будут только по полным именам. Хотя обращение к таблицам по полным именам вполне безопасно, вызовы функций в схеме `public` будут небезопасными или ненадёжными. Кроме того, пользователь, имеющий право `CREATEROLE`, может отменить это назначение и выполнять произвольные запросы от имени пользователей, полагающихся на этот путь. Если вы создаёте функции или расширения в схеме `public` или даёте пользователям право `CREATEROLE`, но не хотите, чтобы они стали практически суперпользователями, вам нужно использовать первый шаблон.
3. Удалить схему `public` из пути поиска `search_path`. Это будет иметь такое же влияние на пользователей, что и предыдущий шаблон. В дополнение к его особенностям относительно функций и права `CREATEROLE`, данный шаблон подразумевает также доверие к владельцам базам данных, как к имеющим право `CREATEROLE`. Если вы создаёте функции или расширения в схеме

`public`, даёте пользователям права `CREATEROLE`, `CREATEDB` или делаете их владельцами отдельных баз данных, но не хотите, чтобы они стали практически суперпользователями, вам нужно использовать первый шаблон.

4. Сохранить поведение по умолчанию. Все пользователи неявно обращаются к схеме `public`. Тем самым имитируется ситуация с полным отсутствием схем, что позволяет осуществить плавный переход из среды без схем. Однако при этом любой пользователь может выполнять произвольные запросы от имени любого пользователя, который не позаботится о своей защите специально. Этот шаблон подходит, только если в базе данных имеется всего один или несколько взаимно доверяющих пользователей.

5. При любом подходе, устанавливая совместно используемые приложения (таблицы, которые нужны всем, дополнительные функции сторонних разработчиков и т. д.), помещайте их в отдельные схемы. Не забудьте дать другим пользователям права для доступа к этим схемам. Тогда пользователи смогут обращаться к этим дополнительным объектам по полному имени или при желании добавят эти схемы в свои пути поиска.

3.7.7. Переносимость

Стандарт SQL не поддерживает обращение в одной схеме к разным объектам, принадлежащим разным пользователям. Более того, в ряде реализаций СУБД нельзя создавать схемы с именем, отличным от имени владельца. На практике, в СУБД, реализующих только базовую поддержку схем согласно стандарту, концепции пользователя и схемы очень близки. Таким образом, многие пользователи полагают, что полное имя на самом деле образуется как `имя_пользователя.имя_таблицы`. И именно так будет вести себя Синергия-БД, если вы создадите схемы для каждого пользователя.

В стандарте SQL нет и понятия схемы `public`. Для максимального соответствия стандарту использовать схему `public` не следует.

Конечно, есть СУБД, в которых вообще не реализованы схемы или пространства имен поддерживают (возможно, с ограничениями) обращения к

другим базам данных. Если вам потребуется работать с этими системами, максимальной переносимости вы достигнете, вообще не используя схемы.

3.8. Наследование

Синергия-БД реализует наследование таблиц, что может быть полезно для проектировщиков баз данных. Стандарт SQL:1999 и более поздние версии определяют возможность наследования типов, но это во многом отличается от того, что описано здесь.

Давайте начнем со следующего примера: предположим, что мы создаем модель данных для городов. В каждом штате есть множество городов, но лишь одна столица. Мы хотим иметь возможность быстро получать город-столицу для любого штата. Это можно сделать, создав две таблицы: одну для столиц штатов, а другую для городов, не являющихся столицами. Однако, что делать, если нам нужно получить информацию о любом городе, будь то столица штата или нет? В решении этой проблемы может помочь наследование. Мы определим таблицу `capitals` как наследника `cities`:

```
CREATE TABLE cities (  
    name                text,  
    population          float,  
    altitude            int      -- в футах  
);
```

```
CREATE TABLE capitals (  
    state               char(2)  
) INHERITS (cities);
```

В этом случае таблица `capitals` наследует все колонки своей родительской таблицы, `cities`. Столицы штатов также имеют дополнительную колонку `state`, в которой будет указан штат.

В Синергия-БД таблица может наследоваться от нуля или нескольких других таблиц, а запросы могут выбирать все строки родительской таблицы или все строки родительской и всех дочерних таблиц. По умолчанию принят последний вариант.

Например, следующий запрос найдет названия всех городов, включая столицы штатов, расположенных выше 500 футов:

```
SELECT name, altitude
FROM cities
WHERE altitude > 500;
```

Для данных из введения он выдаст:

name		altitude
Las Vegas		2174
Mariposa		1953
Madison		845

А следующий запрос находит все города, которые не являются столицами штатов, но также находятся на высоте выше 500 футов:

```
SELECT name, altitude
FROM ONLY cities
WHERE altitude > 500;
```

name		altitude
Las Vegas		2174
Mariposa		1953

Здесь ключевое слово ONLY указывает, что запрос должен применяться только к таблице `cities`, но не к таблицам, расположенным ниже `cities` в иерархии наследования. Многие операторы, которые мы уже обсудили, – `SELECT`, `UPDATE` и `DELETE` – поддерживают ключевое слово `ONLY`.

Вы также можете добавить после имени таблицы `*`, чтобы явно указать, что должны включаться и дочерние таблицы:

```
SELECT name, altitude
FROM cities*
WHERE altitude > 500;
```

Указывать `*` не обязательно, так как теперь это поведение всегда

подразумевается по умолчанию. Однако такая запись всё ещё поддерживается для совместимости со старыми версиями, где поведение по умолчанию могло быть изменено.

В некоторых ситуациях бывает необходимо узнать, из какой таблицы выбрана конкретная строка. Для этого вы можете воспользоваться системным столбцом `tableoid`, присутствующим в каждой таблице:

```
SELECT c.tableoid, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
```

этот запрос выдаст:

tableoid	name	altitude
139793	Las Vegas	2174
139793	Mariposa	1953
139798	Madison	845

Если вы попытаетесь выполнить его у себя, скорее всего вы получите другие значения `OID`. Собственно имена таблиц вы можете получить, обратившись к `pg_class`:

```
SELECT p.relname, c.name, c.altitude
FROM cities c, pg_class p
WHERE c.altitude > 500 AND c.tableoid = p.oid;
```

в результате вы получите:

relname	name	altitude
cities	Las Vegas	2174
cities	Mariposa	1953
capitals	Madison	845

Тот же эффект можно получить другим способом, используя альтернативный тип `regclass`; при этом `OID` таблицы выводится в символьном виде:

```
SELECT c.tableoid::regclass, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
```

Механизм наследования не способен автоматически распределять данные команд INSERT или COPY по таблицам в иерархии наследования. Поэтому в нашем примере этот оператор INSERT не выполнится:

```
INSERT INTO cities (name, population, altitude, state)
VALUES ('New York', NULL, NULL, 'NY');
```

Мы могли надеяться на то, что данные каким-то образом попадут в таблицу capitals, но этого не происходит: INSERT всегда вставляет данные непосредственно в указанную таблицу. В некоторых случаях добавляемые данные можно перенаправлять, используя правила. Однако в нашем случае это не поможет, так как таблица cities не содержит колонки state и команда будет отвергнута до применения правила.

Дочерние таблицы автоматически наследуют от родительской таблицы ограничения-проверки и ограничения NOT NULL (если только для них не задано явно NO INHERIT). Все остальные ограничения (уникальности, первичный ключ и внешние ключи) не наследуются.

Таблица может наследоваться от нескольких родительских таблиц, в этом случае она будет объединять в себе все столбцы этих таблиц, а также столбцы, описанные непосредственно в её определении. Если в определениях родительских и дочерней таблиц встретятся столбцы с одним именем, эти столбцы будут «объединены», так что в дочерней таблице окажется только один столбец. Чтобы такое объединение было возможно, столбцы должны иметь одинаковый тип данных, в противном случае произойдёт ошибка. Наследуемые ограничения-проверки и ограничения NOT NULL объединяются подобным образом. Так, например, объединяемый столбец получит свойство NOT NULL, если какое-либо из порождающих его определений имеет свойство NOT NULL. Ограничения-проверки объединяются, если они имеют одинаковые имена; но если их условия различаются, происходит ошибка.

Отношение наследования между таблицами обычно устанавливается при создании дочерней таблицы с использованием предложения INHERITS оператора CREATE TABLE. Другой способ добавить такое отношение для таблицы,

определенной подходящим образом – использовать `INHERIT` с оператором `ALTER TABLE`. Для этого будущая дочерняя таблица должна уже включать те же колонки (с совпадающими именами и типами), что и родительская таблица. Также она должна включать аналогичные ограничения-проверки (с теми же именами и выражениями). Удалить отношение наследования можно с помощью указания `NO INHERIT` оператора `ALTER TABLE`. Динамическое добавление и удаление отношений наследования может быть полезно при реализации разбиения таблиц (см. раздел 3.10).

Для создания таблицы, которая затем может стать наследником другой, удобно воспользоваться предложением `LIKE` оператора `CREATE TABLE`. Такая команда создаст новую таблицу с теми же колонками, что и исходная. Если в исходной таблице определены ограничения `CHECK`, для создания полностью совместимой таблицы их тоже нужно скопировать, и это можно сделать, добавив к предложению `LIKE` параметр `INCLUDING CONSTRAINTS`.

Родительскую таблицу нельзя удалить, пока существуют унаследованные от нее. Так же как в дочерних таблицах нельзя удалять или модифицировать колонки или ограничения-проверки, унаследованные от родительских таблиц. Если вы хотите удалить таблицу вместе со всеми её потомками, это легко сделать, добавив в команду удаления родительской таблицы параметр `CASCADE`.

При изменениях определений и ограничений колонок команда `ALTER TABLE` распространяет эти изменения вниз в иерархии наследования. Однако удалить колонки, унаследованные дочерними таблицами, можно только с помощью параметра `CASCADE`. При создании отношений наследования команда `ALTER TABLE` следует тем же правилам объединения дублирующихся колонок, что и `CREATE TABLE`.

В запросах с наследуемыми таблицами проверка прав доступа выполняется только в родительской таблице. Так, например, наличие разрешения `UPDATE` для таблицы `cities` подразумевает право на изменение строк также в таблице `capitals`, когда к ним происходит обращение через таблицу `cities`. Это

сохраняет видимость того, что эти данные (также) находятся в родительской таблице. Но изменить таблицу `capitals` напрямую без дополнительного разрешения нельзя. Подобным образом, политики безопасности на уровне строк для родительской таблицы применяются к строкам, получаемым из дочерних таблиц при выполнении запроса с наследованием. Политики же дочерних таблиц, если они определены, действуют только когда такие таблицы явно задействуются в запросе; в этом случае все политики, связанные с родительскими таблицами, игнорируются.

Сторонние таблицы могут также входить в иерархию наследования как родительские или дочерние таблицы, так же, как и обычные. Если в иерархию наследования входит сторонняя таблица, все операции, не поддерживаемые ей, не будут поддерживаться иерархией в целом.

3.8.1. Ограничения

Заметьте, что не все SQL-команды могут работать с иерархиями наследования. Команды, выполняющие выборку данных, изменение данных или модификацию схемы (например, `SELECT`, `UPDATE`, `DELETE`, большинство вариантов `ALTER TABLE`, но не `INSERT` и `ALTER TABLE ... RENAME`), обычно по умолчанию обрабатывают данные дочерних таблиц и могут исключать их, если поддерживают указание `ONLY`. Команды для обслуживания и настройки базы данных (например `REINDEX` и `VACUUM`) обычно работают только с отдельными физическими таблицами и не поддерживают рекурсивную обработку отношений наследования. Соответствующее поведение каждой команды описано в её справке.

Возможности наследования серьезно ограничены тем, что индексы (включая ограничения уникальности) и ограничения внешних ключей относятся только к отдельным таблицам, но не к их потомкам. Это касается обеих сторон ограничений внешних ключей. Таким образом, применительно к нашему примеру:

1. Если мы объявим `cities.name` с ограничением `UNIQUE` или `PRIMARY KEY`, это не мешает добавить в таблицу `capitals` строки с названиями городов, уже существующими в таблице `cities`. И эти дублирующиеся строки по умолчанию будут выводиться в результате запросов к `cities`. На деле

таблица `capitals` по умолчанию вообще не будет содержать ограничение уникальности, так что в ней могут оказаться несколько строк с одним названием. Хотя вы можете добавить в `capitals` соответствующее ограничение, но это не предотвратит дублирование при объединении с `cities`.

2. Подобным образом, если мы укажем, что `cities.name` ссылается (`REFERENCES`) на какую-то другую таблицу, это ограничение не будет автоматически распространено на `capitals`. В этом случае решением может стать явное добавление такого же ограничения `REFERENCES` в таблицу `capitals`.

3. Если вы сделаете, чтобы колонка другой таблицы ссылалась на `cities(name)`, в этой колонке можно будет указывать только названия городов, но не столиц. В этом случае хорошего решения нет.

Некоторая функциональность, не реализованная для иерархий наследования, реализована для декларативного секционирования. Поэтому обязательно взвесьте все за и против, прежде чем применять в своих приложениях секционирование с использованием устаревшего наследования.

3.9. Секционирование таблиц

Синергия-БД поддерживает простое секционирование таблиц. В этом разделе описывается, как и почему бывает полезно применять разделение при проектировании баз данных.

3.9.1. Обзор

Секционированием данных называется разбиение одной большой логической таблицы на несколько меньших физических секций. Секционирование может принести следующую пользу:

1. В определённых ситуациях оно кардинально увеличивает быстродействие, особенно когда большой процент часто запрашиваемых строк таблицы относится к одной или лишь нескольким секциям. Секционирование может сыграть роль ведущих столбцов в индексах, что позволит уменьшить размер ин-

декса и увеличит вероятность нахождения наиболее востребованных частей индексов в памяти.

2. Когда в выборке или изменении данных задействована большая часть одной секции, последовательное сканирование этой секции может выполняться гораздо быстрее, чем случайный доступ по индексу к данным, разбросанным по всей таблице.

3. Массовую загрузку и удаление данных можно осуществлять, добавляя и удаляя секции, если это было предусмотрено при проектировании секционированных таблиц. Операция `ALTER TABLE DETACH PARTITION` или удаление отдельной секции с помощью команды `DROP TABLE` выполняются гораздо быстрее, чем массовая обработка. Эти команды также полностью исключают накладные расходы, связанные с выполнением `VACUUM` после `DELETE`.

4. Редко используемые данные можно перенести на более дешёвые и медленные носители.

Всё это обычно полезно только для очень больших таблиц. Какие именно таблицы выиграют от секционирования, зависит от конкретного приложения, хотя, как правило, это следует применять для таблиц, размер которых превышает объём ОЗУ сервера.

Синергия-БД предлагает поддержку следующих видов секционирования:

1. Секционирование по диапазонам: таблица секционируется по «диапазонам», определённым по ключевому столбцу или набору столбцов, и не пересекающимся друг с другом. Например, можно секционировать данные по диапазонам дат или по диапазонам идентификаторов определённых бизнес-объектов.

2. Секционирование по списку: таблица секционируется с помощью списка, явно указывающего, какие значения ключа должны относиться к каждой секции.

3. Секционирование по хешу: таблица секционируется по определённым модулям и остаткам, которые указываются для каждой секции. Каждая секция

содержит строки, для которых хеш-значение ключа разбиения, делённое на модуль, равняется заданному остатку.

Если вашему приложению требуются другие формы секционирования, можно также прибегнуть к альтернативным реализациям, с использованием наследования и представлений с `UNION ALL`. Такие подходы дают гибкость, но не дают такого выигрыша в производительности, как встроенное декларативное секционирование.

3.9.2. Декларативное секционирование

Синергия-БД предоставляет возможность указать, как разбить таблицу на части, называемые секциями. Разделённая таким способом таблица называется секционированной таблицей. Указание секционирования состоит из определения метода секционирования и списка столбцов или выражений, которые будут составлять ключ разбиения.

Все строки, вставляемые в секционированную таблицу, будут направляться в одну из секций в зависимости от значения ключа разбиения. В каждой секции содержится подмножество данных, определяемое границами секции. В настоящее время поддерживается секционирование по диапазонам, по списку и по хешу.

Сами секции могут представлять собой секционируемые таблицы, благодаря применению так называемого вложенного секционирования. В каждой секции могут быть определены свои индексы, ограничения и значения по умолчанию, отличные от других секций. Подробнее о создании секционированных таблиц и секций рассказывается в описании `CREATE TABLE`.

Преобразовать обычную таблицу в секционированную и наоборот нельзя. Однако в секционированную таблицу можно добавить в качестве секции обычную или секционированную таблицу с данными, а также можно удалить секцию из секционированной таблицы и превратить её в отдельную таблицу; обратитесь к описанию `ALTER TABLE`, чтобы узнать больше о подкомандах `ATTACH PARTITION` и `DETACH PARTITION`.

За кулисами отдельные секции связываются с секционируемой таблицей средствами наследования; однако с секционированными таблицами и секциями

нельзя использовать некоторые общие возможности наследования, описанные ниже. Например, секция не может иметь никаких других родителей, кроме секционированной таблицы, к которой она присоединена, так же как обычная таблица не может наследоваться от секционированной таблицы. Это означает, что секционированные таблицы и их секции не совмещаются в иерархии наследования с обычными таблицами. Но так как иерархия наследования, включающая секционированную таблицу и её секции, остаётся иерархией наследования, на неё распространяются все обычные правила наследования, с некоторыми исключениями. А именно:

1. Ограничения CHECK вместе с NOT NULL, определённые в секционированной таблице, всегда наследуются всеми её секциями. Ограничения CHECK с характеристикой NO INHERIT в секционированных таблицах создавать нельзя.
2. Использование указания ONLY при добавлении или удалении ограничения только в секционированной таблице поддерживается лишь когда в ней нет секций. Если секции существуют, при попытке использования ONLY возникнет ошибка, так как добавление или удаление ограничений только в секционированной таблице при наличии секций не поддерживается. С другой стороны, ограничения можно добавлять или удалять непосредственно в секциях (если они отсутствуют в родительской таблице).
3. Так как секционированная таблица сама по себе не содержит данные, использование TRUNCATE ONLY для секционированной таблицы всегда будет считаться ошибкой.
4. В секциях не может быть столбцов, отсутствующих в родительской таблице. Такие столбцы невозможно определить ни при создании секций командой CREATE TABLE, ни путём последующего добавления в секции командой ALTER TABLE. Таблицы могут быть подключены в качестве секций командой ALTER TABLE ... ATTACH PARTITION, только если их столбцы в точности соответствуют родительской таблице, включая столбец oid (если он есть).

5. Ограничение NOT NULL для столбца в секции нельзя удалить, если это ограничение существует в родительской таблице.

Секции также могут быть сторонними таблицами, хотя при этом накладываются некоторые ограничения, отсутствующие с обычными таблицами; за подробностями обратитесь к описанию CREATE FOREIGN TABLE.

Изменение ключа разбиения в строке может привести к перемещению её в другую секцию, ограничению которой соответствует изменённая строка.

3.9.2.1. Пример

Предположим, что мы создаём базу данных для большой компании, торгующей мороженым. Компания учитывает максимальную температуру и продажи мороженого каждый день в разрезе регионов. По сути нам нужна следующая таблица:

```
CREATE TABLE measurement (  
    city_id          int not null,  
    logdate          date not null,  
    peaktemp        int,  
    unitsales       int  
);
```

Мы знаем, что большинство запросов будут работать только с данными за последнюю неделю, месяц или квартал, так как в основном эта таблица нужна для формирования текущих отчётов для руководства. Чтобы сократить объём хранящихся старых данных, мы решили оставлять данные только за 3 последних года. Ненужные данные мы будем удалять в начале каждого месяца. В этой ситуации мы можем использовать секционирование для удовлетворения всех наших требований к таблице показателей.

Чтобы использовать декларативное секционирование в этом случае, выполните следующее:

1. Создайте таблицу measurement как секционированную таблицу с предложением PARTITION BY, указав метод разбиения (в нашем случае RANGE) и список столбцов, которые будут образовывать ключ разбиения.

```
CREATE TABLE measurement (  
    city_id          int not null,  
    logdate          date not null,  
    peaktemp        int,  
    unitsales       int  
);
```

```

    city_id          int not null,
    logdate           date not null,
    peaktemp         int,
    unitsales        int
) PARTITION BY RANGE (logdate);

```

При разбиении по диапазонам в качестве ключа разбиения при желании можно использовать набор из нескольких столбцов. Конечно, при этом скорее всего увеличится количество секций, и каждая из них будет меньше. И напротив, использование меньшего числа столбцов может привести к менее дробному критерию разбиения с меньшим числом секций. Запрос, обращающийся к секционированной таблице, будет сканировать меньше секций, если в условии поиска фигурируют некоторые или все эти столбцы. Например, в таблице, секционируемой по диапазонам, в качестве ключа разбиения можно выбрать столбцы `lastname` и `firstname` (в таком порядке).

2. Создайте секции. В определении каждой секции должны задаваться границы, соответствующие методу и ключу разбиения родительской таблицы. Заметьте, что указание границ, при котором множество значений новой секции пересекается со множеством значений в одной или нескольких существующих секциях, будет ошибочным. При попытке добавления в родительскую таблицу данных, которые не соответствуют ни одной из существующей секций, произойдет ошибка; соответствующий раздел нужно добавлять вручную.

Секции, создаваемые таким образом, во всех отношениях являются обычными таблицами Синергия-БД (или, возможно, сторонними таблицами). В частности, для каждой секции можно независимо задать табличное пространство и параметры хранения.

Для таблиц-секций нет необходимости определять ограничения с условиями, задающими границы значений. Нужные ограничения секций выводятся неявно из определения границ секции, когда требуется к ним обратиться.

```

CREATE TABLE measurement_y2006m02 PARTITION OF
measurement
    FOR VALUES FROM ('2006-02-01') TO ('2006-03-01');

```

```
CREATE TABLE measurement_y2006m03 PARTITION OF
measurement
    FOR VALUES FROM ('2006-03-01') TO ('2006-04-01');
```

...

```
CREATE TABLE measurement_y2007m11 PARTITION OF
measurement
    FOR VALUES FROM ('2007-11-01') TO ('2007-12-01');
```

```
CREATE TABLE measurement_y2007m12 PARTITION OF
measurement
    FOR VALUES FROM ('2007-12-01') TO ('2008-01-01')
    TABLESPACE fasttablespace;
```

```
CREATE TABLE measurement_y2008m01 PARTITION OF
measurement
    FOR VALUES FROM ('2008-01-01') TO ('2008-02-01')
    WITH (parallel_workers = 4)
    TABLESPACE fasttablespace;
```

Для реализации вложенного секционирования укажите предложение PARTITION BY в командах, создающих отдельные секции, например:

```
CREATE TABLE measurement_y2006m02 PARTITION OF
measurement
    FOR VALUES FROM ('2006-02-01') TO ('2006-03-01')
    PARTITION BY RANGE (peaktemp);
```

Когда будут созданы секции measurement_y2006m02, данные, добавляемые в measurement и попадающие в measurement_y2006m02 (или данные, непосредственно добавляемые в measurement_y2006m02, с учётом соответствия ограничению секции) будут затем перенаправлены в одну из вложенных секций в зависимости от значения столбца peaktemp. Указанный ключ разбиения может пересекаться с ключом разбиения родителя, хотя определять границы вложенной секции нужно осмотрительно, чтобы

множество данных, которое она принимает, входило во множество, допускаемое собственными границами секции; система не пытается контролировать это сама.

3. Создайте в секционируемой таблице индекс по ключевому столбцу (или столбцам), а также любые другие индексы, которые могут понадобиться. Индекс по ключу, строго говоря, создавать не обязательно, но в большинстве случаев он будет полезен. При этом будет автоматически создан отдельный индекс в каждой секции, и все секции, которые вы будете создавать или присоединять позднее, тоже будут содержать такой индекс.

```
CREATE INDEX ON measurement (logdate);
```

4. Убедитесь в том, что параметр конфигурации `enable_partition_pruning` не выключен в `postgresql.conf`. Иначе запросы не будут оптимизироваться должным образом.

В данном примере нам потребуется создавать секцию каждый месяц, так что было бы разумно написать скрипт, который бы формировал требуемый код DDL автоматически.

3.9.2.2. Обслуживание секций

Обычно набор секций, образованный изначально при создании таблиц, не предполагается сохранять неизменным. Чаще наоборот, планируется удалять старые секции данных и периодически добавлять новые. Одно из наиболее важных преимуществ секционирования состоит именно в том, что оно позволяет практически моментально выполнять трудоёмкие операции, изменяя структуру секций, а не физически перемещая большие объёмы данных.

Самый лёгкий способ удалить старые данные – просто удалить секцию, ставшую ненужной:

```
DROP TABLE measurement_y2006m02;
```

Так можно удалить миллионы записей гораздо быстрее, чем удалять их по одной. Заметьте, однако, что приведённая выше команда требует установления блокировки `ACCESS EXCLUSIVE`.

Ещё один часто более предпочтительный вариант – убрать секцию из главной

таблицы, но сохранить возможность обращаться к ней как к самостоятельной таблице:

```
ALTER TABLE measurement DETACH PARTITION  
measurement_y2006m02;
```

При этом можно будет продолжать работать с данными, пока таблица не будет удалена. Например, в этом состоянии очень кстати будет сделать резервную копию данных, используя COPY, pg_dump или подобные средства. Возможно, эти данные также можно будет агрегировать, перевести в компактный формат, выполнить другую обработку или построить отчёты.

Аналогичным образом можно добавлять новую секцию с данными. Мы можем создать пустую секцию в главной таблице так же, как мы создавали секции в исходном состоянии до этого:

```
CREATE TABLE measurement_y2008m02 PARTITION OF  
measurement  
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01')  
    TABLESPACE fasttablespace;
```

А иногда удобнее создать новую таблицу вне структуры секций и сделать её полноценной секцией позже. При таком подходе данные можно будет загрузить, проверить и преобразовать до того, как они появятся в секционированной таблице:

```
CREATE TABLE measurement_y2008m02  
    (LIKE measurement INCLUDING DEFAULTS INCLUDING  
    CONSTRAINTS)  
    TABLESPACE fasttablespace;
```



```
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02  
    CHECK ( logdate >= DATE '2008-02-01' AND logdate <  
    DATE '2008-03-01' );
```



```
\copy measurement_y2008m02 from 'measurement_y2008m02'  
-- possibly some other data preparation work
```



```
ALTER TABLE measurement ATTACH PARTITION  
measurement_y2008m02
```

```
FOR VALUES FROM ('2008-02-01') TO ('2008-03-01' );
```

Прежде чем выполнять команду ATTACH PARTITION, рекомендуется создать ограничение CHECK в присоединяемой таблице, описывающее желаемое ограничение секции. Благодаря этому система сможет не сканировать таблицу для проверки выполнения неявного ограничения секции. Без этого ограничения таблицу нужно будет просканировать и убедиться в выполнении ограничения секции, удерживая блокировку ACCESS EXCLUSIVE в родительской таблице. После выполнения команды ATTACH PARTITION это ограничение можно будет удалить за ненадобностью.

Как говорилось выше, в секционированных таблицах можно создавать индексы, и они будут автоматически применяться ко всей иерархии. Это очень удобно, так как индексироваться будут не только все существующие секции, но и любые секции, создаваемые в будущем. Но есть одно ограничение – такой секционированный индекс нельзя создать в неблокирующем режиме (с указанием CONCURRENTLY). Чтобы не допустить блокировки на долгое время, для создания индекса в самой секционированной таблице можно использовать команду CREATE INDEX ON ONLY; такой индекс будет помечен как нерабочий, и он не будет автоматически применён к секциям. Индексы собственно в секциях можно создать с указанием CONCURRENTLY, а затем присоединить их к индексу родителя, используя команду ALTER INDEX .. ATTACH PARTITION. После того как индексы всех секций будут присоединены к родительскому, последний автоматически перейдёт в рабочее состояние, например:

```
CREATE INDEX measurement_usls_idx ON ONLY measurement
(unitsales);
```

```
CREATE INDEX measurement_usls_200602_idx
ON measurement_y2006m02 (unitsales);
ALTER INDEX measurement_usls_idx
ATTACH PARTITION measurement_usls_200602_idx;
...
```

Этот приём можно применять и с ограничениями UNIQUE и PRIMARY KEY;

для них индексы создаются неявно при создании ограничения, например:

```
ALTER TABLE ONLY measurement ADD UNIQUE (city_id,  
logdate);
```

```
ALTER TABLE measurement_y2006m02 ADD UNIQUE (city_id,  
logdate);
```

```
ALTER INDEX measurement_city_id_logdate_key  
ATTACH PARTITION  
measurement_y2006m02_city_id_logdate_key;  
...
```

3.9.2.3. Ограничения

С секционированными таблицами связаны следующие ограничения:

1. Нельзя создать ограничение-исключение, охватывающее все секции; можно только добавить независимые ограничения в каждую отдельную секцию.
2. Так как в секционированных таблицах первичные ключи не поддерживаются, на секционированные таблицы не могут ссылаться внешние ключи. Ссылки на внешние ключи других таблиц из секционированных таблиц поддерживаются.
3. Триггеры BEFORE ROW при необходимости должны определяться в отдельных секциях, а не в секционированной таблице.
4. Смешивание временных и постоянных отношений в одном дереве секционирования не допускается. Таким образом, если секционированная таблица постоянная, такими же должны быть её секции; с временными таблицами аналогично. В случае с временными отношениями все таблицы дерева секционирования должны быть из одного сеанса.

3.9.3. Реализация с использованием наследования

Хотя встроенное декларативное секционирование полезно во многих часто возникающих ситуациях, бывают обстоятельства, требующие более гибкого подхода. В этом случае секционирование можно реализовать, применив механизм наследования таблиц, что даст ряд возможностей, неподдерживаемых при

декларативном секционировании, например:

1. При декларативном секционировании все секции должны иметь в точности тот же набор столбцов, что и секционируемая таблица, тогда как обычное наследование таблиц допускает наличие в дочерних таблицах дополнительных столбцов, отсутствующих в родителе.
2. Механизм наследования таблиц поддерживает множественное наследование.
3. С декларативным секционированием поддерживается только разбиение по спискам, по диапазонам и по хешу, тогда как с наследованием таблиц данные можно разделять по любому критерию, выбранному пользователем. Однако заметьте, что если исключение по ограничению не позволяет эффективно удалять дочерние таблицы из планов запросов, производительность запросов будет очень низкой.
4. Для некоторых операций с декларативным секционированием требуется более сильная блокировка, чем с использованием наследования. Например, для добавления или удаления секций из секционированной таблицы требуется установить блокировку `ACCESS EXCLUSIVE` в родительской таблице, тогда как в случае с обычным наследованием достаточно блокировки `SHARE UPDATE EXCLUSIVE`.

3.9.3.1. Пример

Мы будем использовать ту же таблицу `measurement`, что и раньше. Чтобы реализовать секционирование с использованием наследования, выполните следующие действия:

1. Создайте «главную» таблицу, от которой будут наследоваться все «дочерние» таблицы. Главная таблица не будет содержать данные. Не определяйте в ней никакие ограничения-проверки, если только вы не намерены применить их во всех дочерних таблицах. Также не имеет смысла определять в ней какие-либо индексы или ограничения уникальности. В нашем примере главной таблицей будет `measurement` со своим изначальным определением.

2. Создайте несколько «дочерних» таблиц, унаследовав их все от главной. Обычно в таких таблицах не будет никаких дополнительных столбцов, кроме унаследованных. Как и с декларативным секционированием, эти таблицы во всех отношениях будут обычными таблицами Синергия-БД (или сторонними таблицами).

```
CREATE TABLE measurement_y2006m02 () INHERITS
(measurement);
CREATE TABLE measurement_y2006m03 () INHERITS
(measurement);
...
CREATE TABLE measurement_y2007m11 () INHERITS
(measurement);
CREATE TABLE measurement_y2007m12 () INHERITS
(measurement);
CREATE TABLE measurement_y2008m01 () INHERITS
(measurement);
```

3. Добавьте в дочерние таблицы неперекрывающиеся ограничения, определяющие допустимые значения ключей для каждой из них. Типичные примеры таких ограничений:

```
CHECK ( x = 1 )
CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire',
'Warwickshire' ))
CHECK ( outletID >= 100 AND outletID < 200 )
```

Убедитесь в том, что ограничения не пересекаются, то есть никакие значения ключа не относятся сразу к нескольким дочерним таблицам. Например, часто допускают такую ошибку в определении диапазонов:

```
CHECK ( outletID BETWEEN 100 AND 200 )
CHECK ( outletID BETWEEN 200 AND 300 )
```

Это не будет работать, так как неясно, к какой дочерней таблице должно относиться значение 200.

Дочерние таблицы лучше будет создать следующим образом:

```
CREATE TABLE measurement_y2006m02 (
    CHECK ( logdate >= DATE '2006-02-01' AND logdate <
DATE '2006-03-01' )
```

```
) INHERITS (measurement);
```

```
CREATE TABLE measurement_y2006m03 (  
    CHECK ( logdate >= DATE '2006-03-01' AND logdate <  
DATE '2006-04-01' )  
    ) INHERITS (measurement);
```

...

```
CREATE TABLE measurement_y2007m11 (  
    CHECK ( logdate >= DATE '2007-11-01' AND logdate <  
DATE '2007-12-01' )  
    ) INHERITS (measurement);
```

```
CREATE TABLE measurement_y2007m12 (  
    CHECK ( logdate >= DATE '2007-12-01' AND logdate <  
DATE '2008-01-01' )  
    ) INHERITS (measurement);
```

```
CREATE TABLE measurement_y2008m01 (  
    CHECK ( logdate >= DATE '2008-01-01' AND logdate <  
DATE '2008-02-01' )  
    ) INHERITS (measurement);
```

4. Для каждой дочерней таблицы создайте индекс по ключевому столбцу (или столбцам), а также любые другие индексы по своему усмотрению:

```
CREATE INDEX measurement_y2006m02_logdate ON  
measurement_y2006m02 (logdate);
```

```
CREATE INDEX measurement_y2006m03_logdate ON  
measurement_y2006m03 (logdate);
```

```
CREATE INDEX measurement_y2007m11_logdate ON  
measurement_y2007m11 (logdate);
```

```
CREATE INDEX measurement_y2007m12_logdate ON  
measurement_y2007m12 (logdate);
```

```
CREATE INDEX measurement_y2008m01_logdate ON  
measurement_y2008m01 (logdate);
```

5. Мы хотим, чтобы наше приложение могло сказать INSERT INTO measurement ... и данные оказались в соответствующей дочерней таблице. Мы можем добиться этого, добавив подходящую триггерную функцию в главную таблицу. Если данные всегда будут добавляться только в последнюю дочернюю таблицу, нам будет достаточно очень простой функции:

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()  
RETURNS TRIGGER AS $$  
BEGIN  
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);  
    RETURN NULL;  
END;  
$$  
LANGUAGE plpgsql;
```

Создав эту функцию, мы создадим вызывающий её триггер:

```
CREATE TRIGGER insert_measurement_trigger  
    BEFORE INSERT ON measurement  
    FOR EACH ROW EXECUTE FUNCTION  
measurement_insert_trigger();
```

Мы должны менять определение триггерной функции каждый месяц, чтобы она всегда указывала на текущую дочернюю таблицу. Определение самого триггера, однако, менять не требуется.

Но мы можем также сделать, чтобы сервер автоматически находил дочернюю таблицу, в которую нужно направить добавляемую строку. Для этого нам потребуется более сложная триггерная функция:

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()  
RETURNS TRIGGER AS $$  
BEGIN  
    IF ( NEW.logdate >= DATE '2006-02-01' AND  
        NEW.logdate < DATE '2006-03-01' ) THEN  
        INSERT INTO measurement_y2006m02 VALUES (NEW.*);  
    ELSIF ( NEW.logdate >= DATE '2006-03-01' AND  
           NEW.logdate < DATE '2006-04-01' ) THEN
```

```

        INSERT INTO measurement_y2006m03 VALUES (NEW.*);
    ...
    ELSIF ( NEW.logdate >= DATE '2008-01-01' AND
           NEW.logdate < DATE '2008-02-01' ) THEN
        INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION
        'Date out of range. Fix the
        measurement_insert_trigger() function!';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;

```

Определение триггера остаётся прежним. Заметьте, что все условия IF должны в точности отражать ограничения CHECK соответствующих дочерних таблиц. Хотя эта функция сложнее, чем вариант с одним текущим месяцем, её не придётся так часто модифицировать, так как ветви условий можно добавить заранее.

Примечание. На практике будет лучше сначала проверять условие для последней дочерней таблицы, если строки добавляются в неё чаще всего, но для простоты мы расположили проверки триггера в том же порядке, как и в других фрагментах кода для этого примера.

Другой способ перенаправления добавляемых строк в соответствующую дочернюю таблицу можно реализовать, определив для главной таблицы не триггер, а правила, например:

```

CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-02-01' AND logdate < DATE
      '2006-03-01' )
DO INSTEAD
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...
CREATE RULE measurement_insert_y2008m01 AS

```

```
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2008-01-01' AND logdate < DATE
      '2008-02-01' )
DO INSTEAD
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

С правилами связано гораздо больше накладных расходов, чем с триггером, но они относятся к запросу в целом, а не к каждой строке. Поэтому этот способ может быть более выигрышным при массовом добавлении данных. Однако в большинстве случаев триггеры будут работать быстрее.

Учтите, что команда COPY игнорирует правила. Если вы хотите вставить данные с помощью COPY, вам придётся копировать их сразу в нужную дочернюю таблицу, а не в главную таблицу. С другой стороны, COPY не отменяет триггеры, так что с триггерами вы сможете использовать её обычным образом.

Ещё один недостаток подхода с правилами связан с невозможностью выдать ошибку, если добавляемая строка не подпадает ни под одно из правил; в этом случае данные просто попадут в главную таблицу.

6. Убедитесь в том, что параметр конфигурации `constraint_exclusion` не выключен в `postgresql.conf`. В противном случае дочерние таблицы могут сканироваться, когда это не требуется.

Как уже можно понять, для реализации сложной иерархии таблиц может потребоваться DDL-код значительного объёма. В данном примере нам потребуется создавать дочернюю таблицу каждый месяц, так что было бы разумно написать скрипт, формирующий требуемый код DDL автоматически.

3.9.3.2. Обслуживание таблиц, секционированных через наследование

Чтобы быстро удалить старые данные, просто удалите ставшую ненужной дочернюю таблицу:

```
DROP TABLE measurement_y2006m02;
```

Чтобы удалить дочернюю таблицу из иерархии наследования, но сохранить к ней доступ как к самостоятельной таблице:

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

Чтобы добавить новую дочернюю таблицу для новых данных, создайте

пустую дочернюю таблицу так же, как до этого создавали начальные:

```
CREATE TABLE measurement_y2008m02 (  
    CHECK ( logdate >= DATE '2008-02-01' AND logdate <  
    DATE '2008-03-01' )  
    ) INHERITS (measurement);
```

Можно также создать новую таблицу и наполнить её данными до добавления в иерархию таблиц. Это позволит загрузить, проверить и при необходимости преобразовать данные до того, как запросы к главной таблице смогут их увидеть.

```
CREATE TABLE measurement_y2008m02  
    (LIKE measurement INCLUDING DEFAULTS INCLUDING  
    CONSTRAINTS);  
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02  
    CHECK ( logdate >= DATE '2008-02-01' AND logdate <  
    DATE '2008-03-01' );  
\copy measurement_y2008m02 from 'measurement_y2008m02'  
-- возможна дополнительная подготовка данных  
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

3.9.3.3. Ограничения

С реализацией секционирования через наследование связаны следующие ограничения:

1. Система не может проверить автоматически, являются ли все ограничения CHECK взаимно исключающими. Поэтому безопаснее будет написать и отладить код для формирования дочерних таблиц и создания и/или изменения связанных объектов, чем делать это вручную.
2. Показанные здесь схемы подразумевают, что ключевой столбец (или столбцы) в строке никогда не меняется, или меняется не настолько, чтобы строку потребовалось перенести в другую секцию. Если же попытаться выполнить такой оператор UPDATE, произойдёт ошибка из-за нарушения ограничения CHECK. Если вам нужно обработать и такие случаи, вы можете установить подходящие триггеры на обновление в дочерних таблицах, но это ещё больше усложнит управление всей конструкцией.

3. Если вы выполняете команды `VACUUM` или `ANALYZE` вручную, не забывайте, что их нужно запускать для каждой дочерней таблицы в отдельности. Команда:

```
ANALYZE measurement;
```

обработает только главную таблицу.

4. Операторы `INSERT` с предложениями `ON CONFLICT` скорее всего не будут работать ожидаемым образом, так как действие `ON CONFLICT` предпринимается только в случае нарушений уникальности в указанном целевом отношении, а не его дочерних отношениях.

5. Для направления строк в нужные дочерние таблицы потребуются триггеры или правила, если только приложение не знает непосредственно о схеме секционирования. Разработать триггеры может быть довольно сложно, и они будут работать гораздо медленнее, чем внутреннее распределение кортежей при декларативном секционировании.

3.9.4. Устранение секций

Устранение секций – это приём оптимизации запросов, который ускоряет работу с декларативно секционированными таблицами, например:

```
SET enable_partition_pruning = on;                -- по
умолчанию
SELECT count(*) FROM measurement WHERE logdate >= DATE
'2008-01-01';
```

Без устранения секций показанный запрос должен будет просканировать все секции таблицы `measurement`. Когда устранение секций включено, планировщик рассматривает определение каждой секции и может заключить, что какую-либо секцию сканировать не нужно, так как в ней не может быть строк, удовлетворяющих предложению `WHERE` в запросе. Когда планировщик может сделать такой вывод, он исключает (устраняет) секцию из плана запроса.

Используя команду `EXPLAIN` и параметр конфигурации `enable_partition_pruning`, можно увидеть отличие плана, из которого были устранены секции, от плана без устранения. Типичный неоптимизированный план

для такой конфигурации таблицы будет выглядеть так:

```
SET enable_partition_pruning = off;
EXPLAIN SELECT count(*) FROM measurement WHERE
logdate >= DATE '2008-01-01';
```

QUERY PLAN

```
-----
Aggregate  (cost=188.76..188.77 rows=1 width=8)
  -> Append  (cost=0.00..181.05 rows=3085 width=0)
        -> Seq Scan on measurement_y2006m02
(cost=0.00..33.12 rows=617 width=0)
      Filter: (logdate >= '2008-01-01'::date)
        -> Seq Scan on measurement_y2006m03
(cost=0.00..33.12 rows=617 width=0)
      Filter: (logdate >= '2008-01-01'::date)
...
        -> Seq Scan on measurement_y2007m11
(cost=0.00..33.12 rows=617 width=0)
      Filter: (logdate >= '2008-01-01'::date)
        -> Seq Scan on measurement_y2007m12
(cost=0.00..33.12 rows=617 width=0)
      Filter: (logdate >= '2008-01-01'::date)
        -> Seq Scan on measurement_y2008m01
(cost=0.00..33.12 rows=617 width=0)
      Filter: (logdate >= '2008-01-01'::date)
```

В некоторых или всех секциях может применяться не полное последовательное сканирование, а сканирование по индексу, но основная идея примера в том, что для удовлетворения запроса не нужно сканировать старые секции. И когда мы включаем устранение секций, мы получаем значительно более эффективный план, дающий тот же результат:

```
SET enable_partition_pruning = on;
EXPLAIN SELECT count(*) FROM measurement WHERE
logdate >= DATE '2008-01-01';
```

QUERY PLAN

```
-----
```

```
-----
Aggregate  (cost=37.75..37.76 rows=1 width=8)
  -> Append  (cost=0.00..36.21 rows=617 width=0)
        -> Seq Scan on measurement_y2008m01
            (cost=0.00..33.12 rows=617 width=0)
            Filter: (logdate >= '2008-01-01'::date)
```

Заметьте, что механизм устранения секций учитывает только ограничения, определённые неявно ключами разбиения, но не наличие индексов. Поэтому определять индексы для столбцов ключа не обязательно. Нужно ли создавать индекс для определённой секции, зависит от того, какую часть секции (меньшую или большую), по вашим представлениям, будут сканировать запросы, обращающиеся к этой секции. Индекс будет полезен в первом случае, но не во втором.

Устранение секций может производиться не только при планировании конкретного запроса, но и в процессе его выполнения. Благодаря этому может быть устранено больше секций, когда условные выражения содержат значения, неизвестные во время планирования; например, это могут быть параметры, определённые оператором PREPARE, значения, получаемые из подзапросов, или параметризованные значения во внутренней стороне соединения с вложенным циклом. Устранение секций во время выполнения запроса возможно в следующие моменты времени:

1. Во время подготовки плана запроса. В этот момент можно устранить секции, учитывая значения параметров, известные при подготовке выполнения запроса. Секции, устранённые на этом этапе, не будут видны в выводе EXPLAIN или EXPLAIN ANALYZE. Их общее количество можно определить по свойству «Subplans Removed» в выводе EXPLAIN.
2. В процессе собственно выполнения плана запроса. Устранение секций также может выполняться на этом этапе и позволяет отфильтровать секции, используя значения, которые становятся известны, когда запрос выполняется фактически. В частности это могут быть значения из подзапросов и значения параметров времени выполнения, например из параметризованных соединений с вложенными циклами. Так как значения параметров могут меняться

многократно при выполнении запроса, устранение секций выполняется при изменении любого из параметров, анализируемых механизмом устранения. Чтобы определить, были ли секции устранены на данном этапе, нужно внимательно изучить свойство `loops` в выводе `EXPLAIN ANALYZE`. Подпланы, соответствующие разным секциям, могут иметь разные значения, в зависимости от того, сколько раз они устранялись во время выполнения. Некоторые из них могут даже иметь значение `(never executed)` (никогда не выполнялись), если они устранялись всегда.

Устранение секций можно отключить, воспользовавшись параметром `enable_partition_pruning`.

Примечание. В настоящее время устранение секций в процессе выполнения распространяется только на узлы типа `Append`, но не `MergeAppend` или `ModifyTable`. Вполне возможно, что это изменится в будущих версиях Синергия-БД.

3.9.5. Секционирование и исключение по ограничению

Исключение по ограничению – приём оптимизации запросов, подобный устранению секций. Прежде всего он применяется, когда секционирование осуществляется с использованием старого метода наследования, но он может быть полезен и для других целей, включая декларативное секционирование.

Исключение по ограничению работает во многом так же, как и устранение секций; отличие состоит в том, что оно использует ограничения `CHECK` всех таблиц (поэтому оно так и называется), тогда как для устранения секций используются границы секции, которые существуют только в случае декларативного секционирования. Ещё одно различие состоит в том, что исключение по ограничению применяется только во время планирования; во время выполнения секции из плана удаляться не будут.

То, что исключение по ограничению использует ограничения `CHECK` (вследствие чего оно работает медленнее устранения секций), иногда может быть и преимуществом. Ограничения могут быть определены даже для декларативно секционированных таблиц, в дополнение к внутренним границам секций, и тогда

исключение по ограничению сможет дополнительно убрать некоторые секции из плана запроса.

По умолчанию параметр `constraint_exclusion` имеет значение не `on` и не `off`, а промежуточное (и рекомендуемое) значение `partition`, при котором этот приём будет применяться только к запросам, где предположительно будут задействованы таблицы, секционированные с использованием наследования. Значение `on` обязывает планировщик просматривать ограничения CHECK во всех запросах, даже в самых простых, где выигрыш от исключения по ограничению маловероятен.

Применяя исключения по ограничению, необходимо учитывать следующее:

1. Исключение по ограничению применяется только при планировании запроса, в отличие от устранения секций, которое может осуществляться и при выполнении запроса.
2. Исключение по ограничению работает только когда предложение WHERE в запросе содержит константы (или получаемые извне параметры). Например, сравнение с функцией переменной природы, такой как `CURRENT_TIMESTAMP`, нельзя оптимизировать, так как планировщик не знает, в какую дочернюю таблицу попадёт значение функции во время выполнения.
3. Ограничения секций должны быть простыми, иначе планировщик не сможет вычислить, какие дочерние таблицы не нужно обрабатывать. Для секционирования по спискам используйте простые условия на равенства, а для секционирования по диапазонам – простые проверки диапазонов, подобные показанным в примерах. Рекомендуется создавать ограничения секций, содержащие только такие сравнения секционирующих столбцов с константами, в которых используются операторы, поддерживающие B-деревья. Это объясняется тем, что в ключе разбиения допускаются только такие столбцы, которые могут быть проиндексированы в B-дереве.
4. При анализе для исключения по ограничению исследуются все ограничения всех дочерних таблиц, относящихся к главной, так что при большом их коли-

честве время планирования запросов может значительно увеличиться. Поэтому устаревший вариант секционирования, основанный на наследовании, будет работать хорошо, пока количество дочерних таблиц не превышает примерно ста; не пытайтесь применять его с тысячами дочерних таблиц.

3.9.6. Рекомендации по декларативному секционированию

К секционированию таблицы следует подходить продуманно, так как неудачное решение может отрицательно повлиять на скорость планирования и выполнения запросов.

Одним из самых важных факторов является выбор столбца или столбцов, по которым будут секционироваться ваши данные. Часто оптимальным будет секционирование по столбцу или набору столбцов, которые практически всегда присутствуют в предложении WHERE в запросах, обращающихся к секционируемой таблице. Элементы предложения WHERE, совместимые с ключом секционирования и соответствующие ему, могут применяться для устранения ненужных для выполнения запроса секций. Однако наличие ограничений PRIMARY KEY или UNIQUE может подтолкнуть к выбору и других столбцов в качестве секционирующих. Также при планировании секционирования следует учитывать, как будут удаляться данные. Секцию целиком можно отсоединить очень быстро, поэтому может иметь смысл разработать стратегию секционирования так, чтобы массово удаляемые данные оказывались в одной секции.

Также важно правильно выбрать число секций, на которые будет разбиваться таблица. Если их будет недостаточно много, индексы останутся большими, не улучшится и локальность данных, вследствие чего процент попаданий в кеш окажется низким. Однако и при слишком большом количестве секций возможны проблемы. С большим количеством секций увеличивается время планирования и потребление памяти как при планировании, так и при выполнении запросов. Выбирая стратегию секционирования таблицы, также важно учитывать, какие изменения могут произойти в будущем. Например, если вы решите создавать отдельные секции для каждого клиента, и в данный момент у вас всего несколько

больших клиентов, подумайте, что будет, если через несколько лет у вас будет много мелких клиентов. В этом случае может быть лучше произвести секционирование по хешу (HASH) и выбрать разумное количество секций, но не создавать секции по списку (LIST) в надежде, что количество клиентов не увеличится до такой степени, что секционирование данных окажется непрактичным.

Вложенное секционирование позволяет дополнительно разделить те секции, которые предположительно окажутся больше других секций. Однако, если этой возможностью злоупотреблять, может образоваться большое количество секций, порождающее те же проблемы, что описаны в предыдущем абзаце.

Также важно учитывать издержки секционирования, которые отражаются на планировании и выполнении запросов. Планировщик запросов обычно довольно неплохо справляется с иерархиями, включающими до нескольких сотен секций, если с типичными запросами ему удаётся устранить почти все секции. Однако по мере увеличения числа секций будет увеличиваться и время планирования запросов, и объём потребляемой памяти. В наибольшей степени это касается команд UPDATE и DELETE. Наличие большого количества секций нежелательно ещё и потому, что потребление памяти сервером может значительно возрасть с течением времени, особенно когда множество сеансов обращаются к множеству секций. Это объясняется тем, что в локальную память каждого сеанса, который обращается к секциям, должны быть загружены метаданные всех этих секций.

С нагрузкой, присущей информационным хранилищам, может иметь смысл создавать больше секций, чем с нагрузкой OLTP. Как правило, в информационных хранилищах время планирования запроса второстепенно, так как гораздо больше времени тратится на выполнение запроса. Однако при любом типе нагрузки важно принимать правильные решения на ранней стадии реализации, так как процесс переразбиения таблиц большого объёма может оказаться чрезвычайно длительным. Для оптимизации стратегии секционирования часто бывает полезно предварительно эмулировать ожидаемую нагрузку. Но ни в коем случае нельзя полагать, что чем больше секций, тем лучше, равно как и наоборот.

3.10.Сторонние данные

Синергия-БД частично реализует спецификацию SQL/MED, позволяя вам обращаться к данным, находящимся снаружи, используя обычные SQL-запросы. Такие данные называются сторонними.

Сторонние данные доступны в Синергия-БД через обертку сторонних данных. Обертка сторонних данных – это библиотека, взаимодействующая с внешним источником данных и скрывающая в себе внутренние особенности подключения и получения данных. Несколько готовых оберток предоставляются в виде модулей contrib. Также вы можете найти другие обертки, выпускаемые как дополнительные продукты. Если ни одна из существующих оберток вас не устраивает, вы можете написать свою собственную.

Чтобы обратиться к сторонним данным, вы должны создать объект сторонний сервер, в котором настраивается подключение к внешнему источнику данных, определяются параметры соответствующей обертки сторонних данных. Затем вы должны создать одну или несколько сторонних таблиц, определив тем самым структуру внешних данных. Сторонние таблицы можно использовать в запросах так же, как и обычные, но их данные не хранятся на сервере Синергия-БД. При каждом запросе Синергия-БД обращается к обертке сторонних данных, которая, в свою очередь, получает данные из внешнего источника или передает их ему (в случае команд INSERT или UPDATE).

При обращении к внешним данным удаленный источник может потребовать аутентификации клиента. Соответствующие учетные данные можно предоставить с помощью сопоставлений пользователей, позволяющих определить в частности имена и пароли, в зависимости от текущей роли пользователя Синергия-БД.

Дополнительную информацию вы найдете в CREATE FOREIGN DATA WRAPPER, CREATE SERVER, CREATE USER MAPPING и CREATE FOREIGN TABLE.

3.11.Другие объекты баз данных

Таблицы – центральные объекты в структуре реляционной базы данных, так

как они содержат ваши данные. Но это не единственные объекты, которые могут в ней существовать. Помимо них вы можете создавать и использовать объекты и других типов, призванные сделать управление данными эффективнее и удобнее. Они не обсуждаются в этой главе, но мы просто перечислим некоторые из них, чтобы вы знали об их существовании:

- представления;
- функции, процедуры и операторы;
- типы данных и домены;
- триггеры и правила перезаписи.

3.12.Отслеживание зависимостей

Когда вы создаете сложные структуры баз данных, включающие множество таблиц с внешними ключами, представлениями, триггерами, функциями и т.п., вы неявно создаете сеть зависимостей между объектами. Например, таблица с ограничением внешнего ключа зависит от таблицы, на которую она ссылается.

Чтобы обеспечить целостность всей структуры базы данных, Синергия-БД не позволяет удалять объекты, от которых зависят другие. Например, попытка удалить таблицу `products` (мы рассматривали её в подразделе 3.3.5), от которой зависит таблица `orders`, приведет к ошибке примерно такого содержания:

```
DROP TABLE products;
```

ОШИБКА: удалить объект "таблица `products`" нельзя, так как от него зависят другие

ПОДРОБНОСТИ: ограничение `orders_product_no_fkey` в отношении "таблица `orders`" зависит от объекта "таблица `products`"

ПОДСКАЗКА: Для удаления зависимых объектов используйте `DROP ... CASCADE`.

Сообщение об ошибке включает полезную подсказку: если вы не хотите заниматься ликвидацией зависимостей по отдельности, вы можете выполнить:

```
DROP TABLE products CASCADE;
```

и все зависимые объекты, а также объекты, зависящие от них, будут удалены

рекурсивно. В этом случае таблица `orders` останется, а удалено будет только её ограничение внешнего ключа. Удаление не распространится на другие объекты, так как ни один объект не зависит от этого ограничения. Если вы хотите проверить, что произойдёт при выполнении `DROP ... CASCADE`, запустите `DROP` без `CASCADE` и прочитайте ПОДРОБНОСТИ (DETAIL).

Все команды `DROP` в Синергия-БД поддерживают указание `CASCADE`. Конечно, вид возможных зависимостей зависит от типа объекта. Вы также можете написать `RESTRICT` вместо `CASCADE`, чтобы включить поведение по умолчанию, когда объект можно удалить, только если от него не зависят никакие другие.

Примечание. Стандарт SQL требует явного указания `RESTRICT` или `CASCADE` в команде `DROP`. Но это требование на самом деле не выполняется ни в одной СУБД, при этом одни системы по умолчанию подразумевают `RESTRICT`, а другие – `CASCADE`.

Если в команде `DROP` перечисляются несколько объектов, `CASCADE` требуется указывать, только когда есть зависимости вне заданной группы. Например, в команде `DROP TABLE tab1, tab2` при наличии внешнего ключа, ссылающегося на `tab1` из `tab2`, можно не указывать `CASCADE`, чтобы она выполнилась успешно.

Для пользовательских функций Синергия-БД отслеживает зависимости, связанные с внешне видимыми свойствами функции, такими как типы аргументов и результата, но не зависимости, которые могут быть выявлены только при анализе тела функции. В качестве примера рассмотрите следующий сценарий:

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow',  
                             'green', 'blue', 'purple');
```

```
CREATE TABLE my_colors (color rainbow, note text);
```

```
CREATE FUNCTION get_color_note (rainbow) RETURNS text AS  
  'SELECT note FROM my_colors WHERE color = $1'  
LANGUAGE SQL;
```

Синергия-БД будет понимать, что функция `get_color_note` зависит от типа `rainbow`: при удалении типа будет принудительно удалена функция, так как тип её

аргумента оказывается неопределённым. Но Синергия-БД не будет учитывать зависимость `get_color_note` от таблицы `my_colors` и не удалит функцию при удалении таблицы. Но у этого подхода есть не только минус, но и плюс. В случае отсутствия таблицы эта функция останется рабочей в некотором смысле: хотя при попытке выполнить её возникнет ошибка, но при создании новой таблицы с тем же именем функция снова будет работать.

4. МОДИФИКАЦИЯ ДАННЫХ

В предыдущей главе мы обсуждали, как создавать таблицы и другие структуры для хранения данных. Теперь пришло время заполнить таблицы данными. В этой главе мы расскажем, как добавлять, изменять и удалять данные из таблиц. А из следующей главы вы наконец узнаете, как извлекать нужные вам данные из базы данных.

4.1. Добавление данных

Сразу после создания таблицы она не содержит никаких данных. Поэтому, чтобы она была полезна, в нее прежде всего нужно добавить данные. По сути данные добавляются в таблицу по одной строке. И хотя вы конечно можете добавить в таблицу несколько строк, добавить в нее меньше, чем строку, невозможно. Даже если вы указываете значения только некоторых колонок, создается полная строка.

Чтобы создать строку, вы будете использовать команду `INSERT`. В этой команде необходимо указать имя таблицы и значения колонок. Например, рассмотрим таблицу товаров из главы 3:

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric  
);
```

Добавить в нее строку можно было бы так:

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

Значения данных перечисляются в порядке колонок в таблице и разделяются запятыми. Обычно в качестве значений указываются константы, но это могут быть и скалярные выражения.

Показанная выше запись имеет один недостаток – вам необходимо знать порядок колонок в таблице. Чтобы избежать этого, можно перечислить колонки явно. Например, следующие две команды дадут тот же результат, что и показанная

выше:

```
INSERT INTO products (product_no, name, price) VALUES  
(1, 'Cheese', 9.99);
```

```
INSERT INTO products (name, price, product_no) VALUES  
( 'Cheese', 9.99, 1);
```

Многие считают, что лучше всегда явно указывать имена колонок.

Если значения определяются не для всех колонок, лишние колонки можно опустить. В таком случае эти колонки получают значения по умолчанию, например:

```
INSERT INTO products (product_no, name) VALUES (1,  
'Cheese');
```

```
INSERT INTO products VALUES (1, 'Cheese');
```

Вторая форма является расширением Синергия-БД. Она заполняет колонки слева по числу переданных значений, а все остальные колонки принимают значения по умолчанию.

Для ясности можно также явно указать значения по умолчанию для отдельных колонок или всей строки:

```
INSERT INTO products (product_no, name, price) VALUES  
(1, 'Cheese', DEFAULT);
```

```
INSERT INTO products DEFAULT VALUES;
```

Одна команда может вставить сразу несколько строк:

```
INSERT INTO products (product_no, name, price) VALUES  
    (1, 'Cheese', 9.99),  
    (2, 'Bread', 1.99),  
    (3, 'Milk', 2.99);
```

Также возможно вставить результат запроса (который может не содержать строк либо содержать одну или несколько):

```
INSERT INTO products (product_no, name, price)  
    SELECT product_no, name, price FROM new_products  
    WHERE release_date = 'today';
```

Это позволяет использовать все возможности механизма запросов SQL для вычисления вставляемых строк.

Примечание. Когда нужно добавить сразу множество строк, возможно будет лучше использовать команду COPY. Она не такая гибкая, как INSERT, но гораздо

эффективнее.

4.2. Изменение данных

Модификация данных, уже сохраненных в БД, называется изменением. Изменить можно все строки таблицы, либо подмножество всех строк, либо только избранные строки. Каждую колонку при этом можно изменять независимо от других.

Для изменения данных в существующих строках используется команда UPDATE. Ей требуется следующая информация:

- имя таблицы и изменяемой колонки;
- новое значение колонки;
- критерий отбора изменяемых строк.

Если вы помните, в главе 3 говорилось, что в SQL в принципе нет уникального идентификатора строк. Таким образом, не всегда возможно явно указать на строку, которую требуется изменить. Поэтому необходимо указать условия, каким должны соответствовать требуемая строка. Только если в таблице есть первичный ключ (вне зависимости от того, объявляли вы его или нет), можно однозначно адресовать отдельные строки, определив условие по первичному ключу. Этим пользуются графические инструменты для работы с базой данных, дающие возможность редактировать данные по строкам.

Например, следующая команда увеличивает цену всех товаров, имевших до этого цену 5, до 10:

```
UPDATE products SET price = 10 WHERE price = 5;
```

В результате может измениться ноль, одна или множество строк. И если этому запросу не будет удовлетворять ни одна строка, это не будет ошибкой.

Давайте рассмотрим эту команду подробнее. Она начинается с ключевого слова UPDATE, за которым идет имя таблицы. Как обычно, имя таблицы может быть записано в полной форме, в противном случае она будет найдена по пути. Затем идет ключевое слово SET, за которым следует имя колонки, знак равенства и новое значение колонки. Этим значением может быть любое скалярное выражение, а не только константа. Например, если вы захотите поднять цену всех товаров на 10%,

это можно сделать так:

```
UPDATE products SET price = price * 1.10;
```

Как видно из этого примера, выражение нового значения может ссылаться на существующие значения колонок в строке. Мы также опустили в нем предложение WHERE. Это означает, что будут изменены все строки в таблице. Если же это предложение присутствует, изменяются только строки, которые соответствуют условию WHERE. Заметьте, что хотя знак равенства в предложении SET обозначает операцию присваивания, а такой же знак в предложении WHERE используется для сравнения, это не приводит к неоднозначности. И конечно, в условии WHERE не обязательно должна быть проверка равенства, а могут применяться и другие операторы. Необходимо только, чтобы это выражение возвращало логический результат.

В команде UPDATE можно изменить значения сразу нескольких колонок, перечислив их в предложении SET, например:

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

4.3. Удаление данных

Мы рассказали о том, как добавлять данные в таблицы и как изменять их. Теперь вам осталось узнать, как удалить данные, которые оказались не нужны. Так же, как добавлять данные можно только целыми строками, удалять их можно только по строкам. В предыдущем разделе мы отметили, что в SQL нет возможности напрямую адресовать отдельные строки, так что удалить избранные строки можно, только сформулировав для них подходящие условия. Но если в таблице есть первичный ключ, с его помощью можно однозначно выделить определенную строку. При этом можно так же удалить группы строк, соответствующие условию, либо сразу все строки таблицы.

Для удаления строк используется команда DELETE; её синтаксис очень похож на синтаксис команды UPDATE. Например, удалить все строки из таблицы с товарами, имеющими цену 10, можно так:

```
DELETE FROM products WHERE price = 10;
```

Если вы напишете просто:

```
DELETE FROM products;
```

будут удалены все строки таблицы! Будьте осторожны!

4.4. Возврат данных из изменённых строк

Иногда бывает полезно получать данные из модифицируемых строк в процессе их обработки. Это возможно с использованием предложения RETURNING, которое можно задать для команд INSERT, UPDATE и DELETE. Применение RETURNING позволяет обойтись без дополнительного запроса к базе для сбора данных и это особенно ценно, когда как-то иначе трудно получить изменённые строки надёжным образом.

В предложении RETURNING допускается то же содержимое, что и в выходном списке команды SELECT. Оно может содержать имена столбцов целевой таблицы команды или значения выражений с этими столбцами. Также часто применяется краткая запись RETURNING *, выбирающая все столбцы целевой таблицы по порядку.

В команде INSERT данные, выдаваемые в RETURNING, образуются из строки в том виде, в каком она была вставлена. Это не очень полезно при простом добавлении, так как в результате будут получены те же данные, что были переданы клиентом. Но это может быть очень удобно при использовании вычисляемых значений по умолчанию. Например, если в таблице есть столбец serial, в котором генерируются уникальные идентификаторы, команда RETURNING может вернуть идентификатор, назначенный новой строке:

```
CREATE TABLE users (firstname text, lastname text, id
serial primary key);
INSERT INTO users (firstname, lastname) VALUES ('Joe',
'Cool') RETURNING id;
```

Предложение RETURNING также очень полезно с INSERT ... SELECT.

В команде UPDATE данные, выдаваемые в RETURNING, образуются новым содержимым изменённой строки, например:

```
UPDATE products SET price = price * 1.10
```



```
WHERE price <= 99.99
```

```
RETURNING name, price AS new_price;
```

В команде DELETE данные, выдаваемые в RETURNING, образуются содержимым удалённой строки, например:

```
DELETE FROM products
```

```
WHERE obsoletion_date = 'today'
```

```
RETURNING *;
```

Если для целевой таблицы заданы триггеры, в RETURNING выдаются данные из строки, изменённой триггерами. Таким образом, RETURNING часто применяется и для того, чтобы проверить содержимое столбцов, изменяемых триггерами.

5. ЗАПРОСЫ

В предыдущих главах рассказывалось, как создать таблицы, как заполнить их данными и как изменить эти данные. Теперь мы обсудим, как получить данные из базы данных.

5.1. Обзор

Процесс или команда получения данных из базы данных называется запросом. В SQL запросы формулируются с помощью команды SELECT. В общем виде команда SELECT записывается так:

```
[WITH запросы_with] SELECT список_выборки  
FROM табличное_выражение [определение_сортировки]
```

В следующих разделах подробно описываются список выборки, табличное выражение и определение сортировки. Запросы WITH являются расширенной возможностью Синергия-БД и будут рассмотрены в последнюю очередь.

Простой запрос выглядит так:

```
SELECT * FROM table1;
```

Если предположить, что в базе данных есть таблица `table1`, эта команда получит все строки с содержимым всех колонок из `table1`. Метод выдачи результата определяет клиентское приложение. Например, программа `psql` выведет на экране ASCII-таблицу, хотя клиентские библиотеки позволяют извлекать отдельные значения из результата запроса. Здесь список выборки задан как `*`, это означает, что запрос должен вернуть все колонки табличного выражения. В списке выборки можно также указать подмножество доступных колонок или составить выражения с этими колонками. Например, если в `table1` есть колонки `a`, `b` и `c` (и возможно, другие), вы можете выполнить следующий запрос:

```
SELECT a, b + c FROM table1;
```

В предположении, что колонки `b` и `c` имеют числовой тип данных. Подробнее это описано в разделе 5.3.

`FROM table1` – это простейший тип табличного выражения, в котором просто читается одна таблица. Вообще табличные выражения могут быть сложными

конструкциями из базовых таблиц, соединений и подзапросов. А можно и вовсе опустить табличное выражение и использовать команду SELECT как калькулятор:

```
SELECT 3 * 4;
```

В этом может быть больше смысла, когда выражения в списке выборки возвращают меняющиеся результаты. Например, можно вызвать функцию так:

```
SELECT random();
```

5.2. Табличные выражения

Табличное выражение вычисляет таблицу. Это выражение содержит предложение FROM, за которым могут следовать предложения WHERE, GROUP BY и HAVING. Тривиальные табличные выражения просто ссылаются на физическую таблицу, её называют также базовой, но в более сложных выражениях такие таблицы можно преобразовывать и комбинировать самыми разными способами.

Необязательные предложения WHERE, GROUP BY и HAVING в табличном выражении определяют последовательность преобразований, осуществляемых с данными таблицы, полученной в предложении FROM. В результате этих преобразований образуется виртуальная таблица, строки которой передаются списку выборки, вычисляющему выходные строки запроса.

5.2.1. Предложение FROM

Предложение FROM образует таблицу из одной или нескольких ссылок на таблицы, разделенных запятыми.

```
FROM табличная_ссылка [, табличная_ссылка [, ...]]
```

Здесь табличной ссылкой может быть имя таблицы (возможно, с именем схемы), производная таблица, например подзапрос, соединение таблиц или сложная комбинация этих вариантов. Если в предложении FROM перечисляются несколько ссылок, для них применяется перекрестное соединение (то есть декартово произведение). Список FROM преобразуется в промежуточную виртуальную таблицу, которая может пройти через преобразования WHERE, GROUP BY и HAVING, и в итоге определит результат табличного выражения.

Когда в табличной ссылке указывается таблица, являющаяся родительской в

иерархии наследования, в результате будут получены строки не только этой таблицы, но и всех её дочерних таблиц. Чтобы выбрать строки только одной родительской таблицы, перед её именем нужно добавить ключевое слово `ONLY`. Учтите, что при этом будут получены только колонки указанной таблицы – дополнительные колонки дочерних таблиц не попадут в результат.

Если же вы не добавляете `ONLY` перед именем таблицы, вы можете дописать после него `*`, тем самым указав, что должны обрабатываться и все дочерние таблицы. Практических причин использовать этот синтаксис больше нет, так как поиск в дочерних таблицах теперь производится по умолчанию. Однако эта запись поддерживается для совместимости со старыми версиями.

5.2.1.1. Соединенные таблицы

Соединенная таблица – это таблица, полученная из двух других (реальных или производных от них) таблиц в соответствии с правилами соединения конкретного типа. Общий синтаксис описания соединенной таблицы:

`T1 тип_соединения T2 [ условие_соединения ]`

Соединения любых типов могут вкладываются друг в друга или объединяться: и `T1`, и `T2` могут быть результатами соединения. Для однозначного определения порядка соединений предложения `JOIN` можно заключать в скобки. Если скобки отсутствуют, предложения `JOIN` обрабатываются слева направо.

5.2.1.2. Типы соединений

Перекрестное соединение:

`T1 CROSS JOIN T2`

Соединенную таблицу образуют все возможные сочетания строк из `T1` и `T2` (т.е. их декартово произведение), а набор её колонок объединяет в себе колонки `T1` со следующими за ними колонками `T2`. Если таблицы содержат `N` и `M` строк, соединенная таблица будет содержать `N * M` строк.

`FROM T1 CROSS JOIN T2` эквивалентно `FROM T1 INNER JOIN T2 ON TRUE`. Эта запись также эквивалентна `FROM T1, T2`.

Примечание. Последняя запись не полностью эквивалентна первым при указании более чем двух таблиц, так как `JOIN` связывает таблицы сильнее, чем запятая. Например, `FROM T1 CROSS JOIN T2 INNER JOIN T3 ON условие` не

равнозначно `FROM T1, T2 INNER JOIN T3 ON условие`, так как условие может ссылаться на T1 в первом случае, но не во втором.

5.2.1.3. Соединения с сопоставлениями строк

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
    ON логическое_выражение
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
    USING ( список колонок соединения )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] }
JOIN T2
```

Слова `INNER` и `OUTER` необязательны во всех формах. По умолчанию подразумевается `INNER` (внутреннее соединение), а при указании `LEFT`, `RIGHT` и `FULL` – внешнее соединение.

Условие соединения указывается в предложении `ON` или `USING`, либо неявно задается ключевым словом `NATURAL`. Это условие определяет, какие строки двух исходных таблиц считаются «соответствующими» друг другу (это подробно рассматривается ниже).

Возможные типы соединений с сопоставлениями строк:

- `INNER JOIN`

Для каждой строки R1 из T1 в результирующей таблице содержится строка для каждой строки в T2, удовлетворяющей условию соединения с R1.

- `LEFT OUTER JOIN`

Сначала выполняется внутреннее соединение (`INNER JOIN`). Затем в результат добавляются все строки из T1, которым не соответствуют никакие строки в T2, а вместо значений колонок T2 вставляются `NULL`. Таким образом, в результирующей таблице всегда будет минимум одна строка для каждой строки из T1.

- `RIGHT OUTER JOIN`

Сначала выполняется внутреннее соединение (`INNER JOIN`). Затем в результат добавляются все строки из T2, которым не соответствуют никакие строки в T1, а вместо значений колонок T1 вставляются `NULL`. Это соединение является обратным к левому (`LEFT JOIN`): в результирующей

таблице всегда будет минимум одна строка для каждой строки из T2.

- FULL OUTER JOIN

Сначала выполняется внутреннее соединение. Затем в результат добавляются все строки из T1, которым не соответствуют никакие строки в T2, а вместо значений колонок T2 вставляются NULL. И наконец, в результат включаются все строки из T2, которым не соответствуют никакие строки в T1, а вместо значений колонок T1 вставляются NULL.

Предложение ON определяет наиболее общую форму условия соединения: в нем указываются выражения логического типа, подобные тем, что используются в предложении WHERE. Пара строк из T1 и T2 соответствуют друг другу, если выражение ON возвращает для них true.

USING – это сокращенная запись условия, полезная в ситуации, когда с обеих сторон соединения колонки имеют одинаковые имена. Она принимает список общих имен колонок через запятую и формирует условие соединения с равенством этих колонок. Например, запись соединения T1 и T2 с USING (a, b) формирует условие ON T1.a = T2.a AND T1.b = T2.b.

Более того, при выводе JOIN USING исключаются избыточные колонки: обе сопоставленных колонки выводить не нужно, так как они содержат одинаковые значения. Тогда как JOIN ON выдает все колонки из T1, а за ними все колонки из T2, JOIN USING выводит одну колонку для каждой пары (в указанном порядке), за ними все оставшиеся колонки из T1 и, наконец, все оставшиеся колонки T2.

Наконец, NATURAL – сокращенная форма USING: она образует список USING из всех имен колонок, существующих в обеих входных таблицах. Как и с USING, эти колонки оказываются в выходной таблице в единственном экземпляре. Если колонок с одинаковыми именами не находится, NATURAL действует как CROSS JOIN.

Примечание. Предложение USING разумно защищено от изменений в соединяемых отношениях, так как оно связывает только явно перечисленные колонки. NATURAL считается более рискованным, так как при любом изменении схемы в одном или другом отношении, когда появляются колонки с совпадающими именами, при соединении будут связываться и эти новые колонки.

Для наглядности предположим, что у нас есть таблицы t1:

num	name
1	a
2	b
3	c

и t2:

num	value
1	xxx
3	yyy
5	zzz

С ними для разных типов соединений мы получим следующие результаты:

=> SELECT * FROM t1 CROSS JOIN t2;

num	name	num	value
1	a	1	xxx
1	a	3	yyy
1	a	5	zzz
2	b	1	xxx
2	b	3	yyy
2	b	5	zzz
3	c	1	xxx
3	c	3	yyy
3	c	5	zzz

(9 rows)

=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;

num	name	num	value
1	a	1	xxx
3	c	3	yyy

(2 rows)

```
=> SELECT * FROM t1 INNER JOIN t2 USING (num);
```

```
  num | name | value
-----+-----+-----
    1 | a    | xxx
    3 | c    | yyy
(2 rows)
```

```
=> SELECT * FROM t1 NATURAL INNER JOIN t2;
```

```
  num | name | value
-----+-----+-----
    1 | a    | xxx
    3 | c    | yyy
(2 rows)
```

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
```

```
  num | name | num | value
-----+-----+-----+-----
    1 | a    |    1 | xxx
    2 | b    |     | 
    3 | c    |    3 | yyy
(3 rows)
```

```
=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
```

```
  num | name | value
-----+-----+-----
    1 | a    | xxx
    2 | b    | 
    3 | c    | yyy
(3 rows)
```

```
=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
```

```
  num | name | num | value
```



```

-----+-----+-----+-----
  1 | a      |    1 | xxx
  3 | c      |    3 | yyy
    |        |    5 | zzz
(3 rows)

```

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;

```

num | name | num | value
-----+-----+-----+-----
  1 | a      |    1 | xxx
  2 | b      |     | 
  3 | c      |    3 | yyy
    |        |    5 | zzz
(4 rows)

```

Условие соединения в предложении ON может также содержать выражения, не связанные непосредственно с соединением. Это может быть полезно в некоторых запросах, но не следует использовать это необдуманно. Рассмотрите следующий запрос:

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';

```

num | name | num | value
-----+-----+-----+-----
  1 | a      |    1 | xxx
  2 | b      |     | 
  3 | c      |     | 
(3 rows)

```

Заметьте, что если поместить ограничение в предложение WHERE, вы получите другой результат:

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value = 'xxx';

```

num | name | num | value
-----+-----+-----+-----
  1 | a      |    1 | xxx

```

(1 row)

Это связано с тем, что ограничение, помещенное в предложение ON, обрабатывается до операции соединения, тогда как ограничение в WHERE – после. Это не имеет значения при внутренних соединениях, но важно при внешних.

5.2.1.4. Псевдонимы таблиц и колонок

Таблицам и ссылкам на сложные таблицы в запросе можно дать временное имя, по которому к ним можно будет обращаться в рамках запроса. Такое имя называется псевдонимом таблицы.

Определить псевдоним таблицы можно, написав

```
FROM табличная_ссылка AS псевдоним
```

или

```
FROM табличная_ссылка псевдоним
```

Ключевое слово AS является необязательным. Вместо псевдоним здесь может быть любой идентификатор.

Псевдонимы часто применяются для назначения коротких идентификаторов длинным именам таблиц с целью улучшения читаемости запросов, например:

```
SELECT * FROM «очень_длинное_имя_таблицы» s JOIN  
«другое_длинное_имя» a  
ON s.id = a.num;
```

Псевдоним становится новым именем таблицы в рамках текущего запроса, т.е. после назначения псевдонима использовать исходное имя таблицы в другом месте запроса нельзя. Таким образом, следующий запрос недопустим:

```
SELECT * FROM my_table AS m WHERE my_table.a > 5;      --  
неправильно
```

Хотя в основном псевдонимы используются для удобства, они бывают необходимы, когда таблица соединяется сама с собой, например:

```
SELECT * FROM people AS mother JOIN people AS child  
ON mother.id = child.mother_id;
```

Кроме того, псевдонимы обязательно нужно назначать подзапросам (см. подраздел 5.2.1.3).

В случае неоднозначности определения псевдонимов можно использовать скобки. В следующем примере первый оператор назначает псевдоним b второму

экземпляру `my_table`, а второй оператор назначает псевдоним результату соединения:

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...  
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS  
b ...
```

В другой форме назначения псевдонима временные имена даются не только таблицам, но и её колонкам:

```
FROM табличная_ссылка [AS] псевдоним ( колонка1 [,  
колонка2 [, ...]] )
```

Если псевдонимов колонок оказывается меньше, чем фактически колонок в таблице, остальные колонки сохраняют свои исходные имена. Эта запись особенно полезна для замкнутых соединений или подзапросов.

Когда псевдоним применяется к результату JOIN, он скрывает оригинальные имена таблиц внутри JOIN. Например, это допустимый SQL-запрос:

```
SELECT a.* FROM my_table AS a JOIN your_table AS b  
ON ...
```

а запрос:

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b  
ON ...) AS c
```

ошибочный, так как псевдоним таблицы `a` не виден снаружи определения псевдонима `c`.

5.2.1.5. Подзапросы

Подзапросы, образующие таблицы, должны заключаться в скобки и им обязательно должны назначаться псевдонимы (как описано в подразделе 5.2.1.2), например:

```
FROM (SELECT * FROM table1) AS псевдоним
```

Этот пример равносильна записи `FROM table1 AS псевдоним`. Более интересные ситуации, которые нельзя свести к простому соединению, возникают, когда в подзапросе используются агрегирующие функции или группировка.

Подзапросом может также быть список VALUES:

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'),  
('joe', 'blow'))
```

```
AS names(first, last)
```

Такому подзапросу тоже требуется псевдоним. Назначать псевдонимы колонкам списка VALUES не требуется, но вообще это хороший прием. Подробнее это описано в разделе 5.7.

5.2.1.6. Табличные функции

Табличные функции – это функции, выдающие набор строк, содержащих либо базовые типы данных (скалярных типов), либо составные типы (табличные строки). Они применяются в запросах как таблицы, представления или подзапросы в предложении FROM. Колонки, возвращенные табличными функциями, можно включить в выражения SELECT, JOIN или WHERE так же, как колонки таблиц, представлений или подзапросов.

Табличные функции можно также скомбинировать, используя запись ROWS FROM. Результаты функций будут возвращены в параллельных колонках; число строк в этом случае будет наибольшим из результатов всех функций, а результаты функций с меньшим количеством строк будут дополнены значениями NULL.

```
вызов_функции [WITH ORDINALITY] [[AS] псевдоним_таблицы  
[ (псевдоним_колонки [, ...]) ]]  
ROWS FROM( вызов_функции [, ...] ) [WITH ORDINALITY]  
[[AS] псевдоним_таблицы [ (псевдоним_колонки [, ...]) ]]
```

Если указано предложение WITH ORDINALITY, к колонкам результатов функций будет добавлена ещё одна, с типом bigint. В этой колонке нумеруются строки результирующего набора, начиная с 1. Это обобщение стандартного SQL-синтаксиса UNNEST ... WITH ORDINALITY. По умолчанию, эта колонка называется ordinality, но ей можно присвоить и другое имя с помощью указания AS.

Специальную табличную функцию UNNEST можно вызвать с любым числом параметров-массивов, а возвращает она соответствующее число колонок, как если бы UNNEST вызывалась для каждого параметра в отдельности, а результаты объединялись с помощью конструкции ROWS FROM.

```
UNNEST( выражение_массива [, ...] ) [WITH ORDINALITY]  
[[AS] псевдоним_таблицы [ (псевдоним_колонки [, ...]) ]]
```

Если псевдоним_таблицы не указан, в качестве имени таблицы используется

имя функции; в случае с конструкцией `ROWS FROM()` – имя первой функции.

Если псевдонимы колонок не указаны, то для функции, возвращающей базовый тип данных, именем колонки будет имя функции. Для функций, возвращающих составной тип, имена результирующих колонок определяются индивидуальными атрибутами типа.

Примеры:

```
CREATE TABLE foo (fooid int, foosubid int, fooname
text);
```

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM getfoo(1) AS t1;
```

```
SELECT * FROM foo
    WHERE foosubid IN (
                        SELECT foosubid
                        FROM getfoo(foo.fooid) z
                        WHERE z.fooid = foo.fooid
                    );
```

```
CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);
```

```
SELECT * FROM vw_getfoo;
```

В некоторых случаях бывает удобно определить табличную функцию, возвращающую различные наборы колонок при разных вариантах вызова. Для этого нужно указать, что она возвращает псевдотип `record`. Используя такую функцию, ожидаемую структуру строк нужно описать в самом запросе, чтобы система знала, как разобрать запрос и составить его план. Записывается это так:

```
вызов_функции [AS] псевдоним (определение_колонки
[, ...])
вызов_функции AS [псевдоним] (определение_колонки
```

```
[, ...])  
ROWS FROM( ... вызов_функции AS (определение_колонки  
[, ...]) [, ...] )
```

Без ROWS FROM() список определения_колонок заменяет список псевдонимов, который можно также добавить в предложении FROM; имена в определениях колонок служат псевдонимами. С ROWS FROM() список определения_колонок можно добавить к каждой функции отдельно, либо в случае с одной функцией и без предложения WITH ORDINALITY, список определения_колонок можно записать вместо списка с псевдонимами колонок после ROWS FROM().

Взгляните на этот пример:

```
SELECT *  
    FROM dblink('dbname=mydb', 'SELECT proname, prosrc  
    FROM pg_proc')  
    AS t1(proname name, prosrc text)  
    WHERE proname LIKE 'bytea%';
```

Здесь функция dblink (из модуля dblink) выполняет удаленный запрос. Она объявлена как функция, возвращающая тип record, так как он подойдет для запроса любого типа. В этом случае фактический набор колонок функции необходимо описать в вызывающем её запросе, чтобы анализатор запроса знал, например, как преобразовать *.

5.2.1.7. Подзапросы LATERAL

Перед подзапросами в предложении FROM можно добавить ключевое слово LATERAL. Это позволит ссылаться в них на колонки предшествующих элементов списка FROM. Без LATERAL каждый подзапрос выполняется независимо и поэтому не может обращаться к другим элементам FROM.

Перед табличными функциями в предложении FROM также можно указать LATERAL, но для них это ключевое слово необязательно; в аргументах функций в любом случае можно обращаться к колонкам в предыдущих элементах FROM.

Элемент LATERAL может находиться на верхнем уровне списка FROM или в дереве JOIN. В последнем случае он может также ссылаться на любые элементы в

левой части JOIN, справа от которого он находится.

Когда элемент FROM содержит ссылки LATERAL, запрос выполняется следующим образом: сначала для строки элемента FROM с целевыми колонками, или набора строк из нескольких элементов FROM, содержащих целевые колонки, вычисляется элемент LATERAL со значениями этих колонок. Затем результирующие строки обычным образом соединяются со строками, из которых они были вычислены. Эта процедура повторяется для всех строк исходных таблиц.

LATERAL можно использовать так:

```
SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE
bar.id = foo.bar_id) ss;
```

Здесь это не очень полезно, так как тот же результат можно получить более простым и привычным способом:

```
SELECT * FROM foo, bar WHERE bar.id = foo.bar_id;
```

Применять LATERAL имеет смысл в основном, когда для вычисления соединяемых строк необходимо обратиться к колонкам других таблиц. В частности, это полезно, когда нужно передать значение функции, возвращающей набор данных. Например, если предположить, что vertices (polygon) возвращает набор вершин многоугольника, близкие вершины многоугольников из таблицы polygons можно получить так:

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1, polygons p2,
     LATERAL vertices(p1.poly) v1,
     LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

Этот запрос можно записать и так:

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1 CROSS JOIN LATERAL vertices(p1.poly)
v1,
     polygons p2 CROSS JOIN LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

или переформулировать другими способами. Как уже упоминалось, в данном

примере ключевое слово `LATERAL` не требуется, но мы добавили его для ясности.

Особенно полезно бывает использовать `LEFT JOIN` с подзапросом `LATERAL`, чтобы исходные строки оказывались в результате, даже если подзапрос `LATERAL` не возвращает строк. Например, если функция `get_product_names()` выдает названия продуктов, выпущенных определенным производителем, но о продукции некоторых производителей информации нет, мы можем найти, каких именно, примерно так:

```
SELECT m.name
FROM manufacturers m LEFT JOIN LATERAL
get_product_names(m.id) pname ON true
WHERE pname IS NULL;
```

5.2.2. Предложение `WHERE`

Предложение `WHERE` записывается так:

`WHERE условие_ограничения`

где `условие_ограничения` – любое выражение значения (см. раздел 2.2), выдающее результат типа `boolean`.

После обработки предложения `FROM` каждая строка полученной виртуальной таблицы проходит проверку по условию ограничения. Если результат условия равен `true`, эта строка остается в выходной таблице, а иначе (если результат равен `false` или `NULL`) отбрасывается. В условии ограничения, как правило, задействуется минимум одна колонка из таблицы, полученной на выходе `FROM`. Хотя строго говоря, это не требуется, но в противном случае предложение `WHERE` будет бессмысленным.

Примечание. Условие для внутреннего соединения можно записать как в предложении `WHERE`, так и в предложении `JOIN`. Например, это выражение:

```
FROM a, b WHERE a.id = b.id AND b.val > 5
```

равнозначно этому:

```
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
```

и возможно, даже этому:

```
FROM a NATURAL JOIN b WHERE b.val > 5
```

Какой вариант выбрать, в основном дело вкуса и стиля. Вариант с `JOIN` внутри предложения `FROM`, возможно, не лучший с точки зрения совместимости с другими СУБД, хотя он и описан в стандарте SQL. Но для внешних соединений других

вариантов нет: их можно записывать только во FROM. Предложения ON и USING во внешних соединениях не равнозначны условию WHERE, так как они могут добавлять строки (для входных строк без соответствия), а также удалять их из конечного результата.

Несколько примеров запросов с WHERE:

```
SELECT ... FROM fdt WHERE c1 > 5
```

```
SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE  
c2 = fdt.c1 + 10)
```

```
SELECT ... FROM fdt WHERE c1 BETWEEN  
(SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100
```

```
SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2  
WHERE c2 > fdt.c1)
```

fdt – название таблицы, порожденной в предложении FROM. Строки, которые не соответствуют условию WHERE, исключаются из fdt. Обратите внимание, как в качестве выражений значения используются скалярные подзапросы. Как и любые другие запросы, подзапросы могут содержать сложные табличные выражения. Заметьте также, что fdt используется в подзапросах. Дополнение имени c1 в виде fdt.c1 необходимо только, если в порожденной таблице в подзапросе также оказывается колонка c1. Полное имя придает ясность даже там, где без него можно обойтись. Этот пример показывает, как область именования колонок внешнего запроса распространяется на все вложенные в него внутренние запросы.

5.2.3. Предложения GROUP BY и HAVING

Строки порожденной входной таблицы, прошедшие фильтр WHERE, можно сгруппировать с помощью предложения GROUP BY, а затем оставить в результате только нужные группы строк, используя предложение HAVING.

```
SELECT список_выборки
      FROM ...
      [WHERE ...]
      GROUP BY группирующая_колонка [,
      группирующая_колонка] ...
```

Предложение GROUP BY группирует строки таблицы, объединяя их в одну группу при совпадении значений во всех перечисленных колонках. Порядок, в котором указаны колонки, не имеет значения. В результате наборы строк с одинаковыми значениями преобразуются в отдельные строки, представляющие все строки группы. Это может быть полезно для устранения избыточности выходных данных и/или для вычисления агрегатных функций, примененных к этим группам, например:

```
=> SELECT * FROM test1;
```

```
  x | y
----+---
  a | 3
  c | 2
  b | 5
  a | 1
(4 rows)
```

```
=> SELECT x FROM test1 GROUP BY x;
```

```
  x
---
  a
  b
  c
(3 rows)
```

Во втором запросе мы не могли написать SELECT * FROM test1 GROUP BY x, так как для колонки y нет единого значения, связанного с каждой группой. Однако колонки, по которым выполняется группировка, можно использовать в списке выборки, так как они имеют единственное значение в каждой группе.

Вообще говоря, в группированной таблице колонки, не включенные в список GROUP BY, можно использовать только в агрегатных выражениях. Пример такого агрегатного выражения:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
```

```

x | sum
---+-----
a |    4
b |    5
c |    2
(3 rows)
```

Здесь sum – агрегатная функция, вычисляющая единственное значение для всей группы. Подробную информацию о существующих агрегатных функциях можно найти в разделе 1.21 (Часть 2. Руководство оператора).

Примечание: Группировка без агрегатных выражений по сути выдает набор различающихся значений колонок. Этот же результат можно получить с помощью предложения DISTINCT.

Взгляните на следующий пример: в нем вычисляется общая сумма продаж по каждому продукту (а не общая сумма по всем продуктам):

```

SELECT product_id, p.name, (sum(s.units) * p.price) AS
sales
FROM products p LEFT JOIN sales s USING (product_id)
GROUP BY product_id, p.name, p.price;
```

В этом примере колонки product_id, p.name и p.price должны присутствовать в списке GROUP BY, так как они используются в списке выборки. Колонка s.units может отсутствовать в списке GROUP BY, так как она используется только в агрегатном выражении (sum(...)), вычисляющем сумму продаж. Для каждого продукта этот запрос возвращает строку с итоговой суммой по всем продажам данного продукта.

Если бы в таблице products по колонке product_id был создан первичный ключ, тогда в данном примере было бы достаточно сгруппировать строки по product_id, так как название и цена продукта функционально зависят от кода продукта и можно однозначно определить, какое название и цену возвращать для

каждой группы по ID.

В стандарте SQL GROUP BY может группировать только по колонкам исходной таблицы, но расширение Синергия-БД позволяет использовать в GROUP BY колонки из списка выборки. Также возможна группировка по выражениям, а не просто именам колонок.

Если таблица была сгруппирована с помощью GROUP BY, но интерес представляют только некоторые группы, отфильтровать их можно с помощью предложения HAVING, действующего подобно WHERE. Записывается это так:

```
SELECT список_выборки FROM ... [WHERE ...] GROUP BY ...
      HAVING логическое_выражение
```

В предложении HAVING могут использоваться и группирующие выражения, и выражения, не участвующие в группировке (в этом случае это должны быть агрегирующие функции).

Пример.

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING
sum(y) > 3;
```

```
  x | sum
----+-----
  a |    4
  b |    5
(2 rows)
```

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x <
'c';
```

```
  x | sum
----+-----
  a |    4
  b |    5
(2 rows)
```

И ещё один более реалистичный пример:

```
SELECT product_id, p.name, (sum(s.units) * (p.price -
p.cost)) AS profit
```

```
FROM products p LEFT JOIN sales s USING (product_id)
WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
GROUP BY product_id, p.name, p.price, p.cost
HAVING sum(p.price * s.units) > 5000;
```

В данном примере предложение WHERE выбирает строки по колонке, не включенной в группировку (выражение истинно только для продаж за последние четыре недели), тогда как предложение HAVING отфильтровывает группы с общей суммой продаж больше 5000. Заметьте, что агрегатные выражения не обязательно должны быть одинаковыми во всех частях запроса.

Если в запросе есть вызовы агрегатных функций, но нет предложения GROUP BY, строки все равно будут группироваться: в результате окажется одна строка группы (или возможно, ни одной строки, если эта строка будет отброшена предложением HAVING). Это справедливо и для запросов, которые содержат только предложение HAVING, но не содержат вызовы агрегатных функций и предложение GROUP BY.

5.2.4. GROUPING SETS, CUBE и ROLLUP

Более сложные, чем описанные выше, операции группировки возможны с концепцией наборов группирования. Данные, выбранные предложениями FROM и WHERE, группируются отдельно для каждого заданного набора группирования, затем для каждой группы вычисляются агрегатные функции как для простых предложений GROUP BY, и в конце возвращаются результаты, например:

```
=> SELECT * FROM items_sold;
  brand | size | sales
-----+-----+-----
  Foo   | L    |   10
  Foo   | M    |   20
  Bar   | M    |   15
  Bar   | L    |    5
(4 rows)
```

```
=> SELECT brand, size, sum(sales) FROM items_sold
      GROUP BY GROUPING SETS ((brand), (size), ());
```

brand	size	sum
-----+-----+-----		
Foo		30
Bar		20
	L	15
	M	35
		50

(5 rows)

В каждом внутреннем списке GROUPING SETS могут задаваться ноль или более столбцов или выражений, которые воспринимаются так же, как если бы они были непосредственно записаны в предложении GROUP BY. Пустой набор группировки означает, что все строки сводятся к одной группе (которая выводится, даже если входных строк нет), как описано выше для агрегатных функций без предложения GROUP BY.

Ссылки на группирующие столбцы или выражения заменяются в результирующих строках значениями NULL для тех группирующих наборов, в которых эти столбцы отсутствуют. Чтобы можно было понять, результатом какого группирования стала конкретная выходная строка, предназначена функция GROUPING().

Для указания двух распространённых видов наборов группирования предусмотрена краткая запись. Предложение формы:

```
ROLLUP ( e1, e2, e3, ... )
```

представляет заданный список выражений и всех префиксов списка, включая пустой список; то есть оно равнозначно записи:

```
GROUPING SETS (
    ( e1, e2, e3, ... ),
    ...
    ( e1, e2 ),
    ( e1 ),
```

()

)

Оно часто применяется для анализа иерархических данных, например, для суммирования зарплаты по отделам, подразделениям и компании в целом.

Предложение формы:

CUBE (e1, e2, ...)

представляет заданный список и все его возможные подмножества (степень множества). Таким образом, запись:

CUBE (a, b, c)

равнозначна:

```
GROUPING SETS (
    ( a, b, c ),
    ( a, b      ),
    ( a,      c ),
    ( a          ),
    (      b, c ),
    (      b      ),
    (          c ),
    (              )
)
```

)

Элементами предложений CUBE и ROLLUP могут быть либо отдельные выражения, либо вложенные списки элементов в скобках. Вложенные списки обрабатываются как атомарные единицы, с которыми формируются отдельные наборы группирования, например:

CUBE ((a, b), (c, d)) равнозначно

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b          ),
    (          c, d ),
    (              )
)
```

)

и

```
ROLLUP ( a, (b, c), d ) равнозначно
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b, c      ),
    ( a              ),
    (                )
)
```

Конструкции CUBE и ROLLUP могут применяться либо непосредственно в предложении GROUP BY, либо вкладываться внутрь предложения GROUPING SETS. Если одно предложение GROUPING SETS вкладывается внутрь другого, результат будет таким же, как если бы все элементы внутреннего предложения были записаны непосредственно во внешнем.

Если в одном предложении GROUP BY задаётся несколько элементов группирования, окончательный список наборов группирования образуется как прямое произведение этих элементов, например:

```
GROUP BY a, CUBE (b, c), GROUPING SETS ((d), (e))
```

равнозначно:

```
GROUP BY GROUPING SETS (
    (a, b, c, d), (a, b, c, e),
    (a, b, d),   (a, b, e),
    (a, c, d),   (a, c, e),
    (a, d),      (a, e)
)
```

Примечание. Конструкция (a, b) обычно воспринимается в выражениях как конструктор строки. Однако в предложении GROUP BY на верхнем уровне выражений запись (a, b) воспринимается как список выражений, как описано выше. Если вам по какой-либо причине нужен именно конструктор строки в выражении группирования, используйте запись ROW (a, b).

5.2.5. Обработка оконных функций

Если запрос содержит оконные функции, эти функции вычисляются после каждой группировки, агрегатных выражений и фильтрации HAVING. Другими

словами, если в запросе есть агрегатные функции, предложения GROUP BY или HAVING, оконные функции видят не исходные строки, полученные из FROM/WHERE, а сгруппированные.

Когда используются несколько оконных функций, все оконные функции, имеющие в своих определениях синтаксически равнозначные предложения PARTITION BY и ORDER BY, гарантированно обрабатывают данные за один проход. Таким образом, они увидят один порядок сортировки, даже если ORDER BY не определяет порядок однозначно. Однако относительно функций с разными формулировками PARTITION BY и ORDER BY никаких гарантий не дается. В таких случаях между проходами вычислений оконных функций обычно требуется дополнительный этап сортировки и эта сортировка может не сохранять порядок строк, равнозначный с точки зрения ORDER BY.

В настоящее время оконные функции всегда требуют предварительно отсортированных данных, так что результат запроса будет отсортирован согласно тому или иному предложению PARTITION BY/ORDER BY оконных функций. Однако полагаться на это не следует. Если вы хотите, чтобы результаты сортировались определенным образом, явно добавьте предложение ORDER BY на верхнем уровне запроса.

5.3. Списки выборки

Как говорилось в предыдущем разделе, табличное выражение в SELECT создает промежуточную виртуальную таблицу, возможно объединяя таблицы, представления, группируя и исключая лишние строки и т.д. Полученная таблица передается для обработки в список выборки. Этот список выбирает, какие колонки промежуточной таблицы должны выводиться в результате и как именно.

5.3.1. Элементы списка выборки

Простейший список выборки образует элемент *, который выбирает все колонки из полученного табличного выражения. Список выборки также может содержать список выражений значения через запятую (как определено в разделе 2.2).

Например, это может быть список имен колонок:

```
SELECT a, b, c FROM ...
```

Имена колонок *a*, *b* и *c* представляют либо фактические имена колонок таблиц, перечисленных в предложении *FROM*, либо их псевдонимы, определенные как описано в подразделе 5.2.1.2. Пространство имен в списке выборки то же, что и в предложении *WHERE*, если не используется группировка. В противном случае оно совпадает с пространством имен предложения *HAVING*.

Если колонка с заданным именем есть в нескольких таблицах, необходимо также указать имя таблицы, например так:

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

Обращаясь к нескольким таблицам, бывает удобно получить сразу все колонки одной из таблиц:

```
SELECT tbl1.*, tbl2.a FROM ...
```

См. также подраздел 5.2.2.

Если в списке выборки используется обычное выражение значения, по сути при этом в возвращаемую таблицу добавляется новая виртуальная колонка. Выражение значения вычисляется один раз для каждой строки результата со значениями колонок в данной строке. Хотя выражения в списке выборки не обязательно должны обращаться к колонкам табличного выражения из предложения *FROM*; они могут содержать, например и простые арифметические выражения.

5.3.2. Метки колонок

Элементам в списке выборки можно назначить имена для последующей обработки, например, для указания в предложении *ORDER BY* или для вывода в клиентском приложении, например:

```
SELECT a AS value, b + c AS sum FROM ...
```

Если выходное имя колонки не определено (с помощью *AS*), система назначает имя сама. Для простых ссылок на колонки этим именем становится имя целевой колонки, а для вызовов функций это имя функции. Для сложных выражений система генерирует некоторое подходящее имя.

Слово AS можно опустить, но только если имя новой колонки не является ключевым словом Синергия-БД. Во избежание случайного совпадения имени с ключевым словом это имя можно заключить в кавычки. Например, VALUE – ключевое слово, поэтому такой вариант не будет работать:

```
SELECT a value, b + c AS sum FROM ...
```

а такой будет:

```
SELECT a «value», b + c AS sum FROM ...
```

Для предотвращения конфликта с ключевыми словами, которые могут появиться в будущем, рекомендуется всегда писать AS или заключать метки выходных колонок в кавычки.

Примечание. Именованые выходных колонок отличается от того, что происходит в предложении FROM (см. подраздел 5.2.1.2). Одну колонку можно переименовать дважды, но на выходе окажется имя, назначенное в списке выборки.

5.3.3. DISTINCT

После обработки списка выборки в результирующей таблице можно дополнительно исключить дублирующиеся строки. Для этого сразу после SELECT добавляется ключевое слово DISTINCT:

```
SELECT DISTINCT список_выборки ...
```

Чтобы явно включить поведение по умолчанию, когда возвращаются все строки, вместо DISTINCT можно указать ключевое слово ALL.

Две строки считаются разными, если они содержат различные значения минимум в одной колонке. При этом значения NULL полагаются равными.

Кроме того, можно явно определить, какие строки будут считаться различными, следующим образом:

```
SELECT DISTINCT ON (выражение [, выражение ...])  
список_выборки ...
```

Здесь выражение – обычное выражение значения, вычисляемое для всех строк. Строки, для которых перечисленные выражения дают один результат, считаются дублирующимися и возвращается только первая строка из такого набора. Заметьте, что «первая строка» набора может быть любой, если только запрос не включает сортировку, гарантирующую однозначный порядок строк, поступающих в фильтр

DISTINCT. Обработка DISTINCT ON производится после сортировки ORDER BY.

Предложение DISTINCT ON не описано в стандарте SQL и иногда его применение считается плохим стилем из-за возможной неопределенности в результатах. При разумном использовании GROUP BY и подзапросов во FROM можно обойтись без этой конструкции, но часто она бывает удобнее.

5.4. Сочетание запросов

Результаты двух запросов можно обработать, используя операции над множествами: объединение, пересечение и вычитание. Эти операции записываются соответственно так:

```
запрос1 UNION [ALL] запрос2  
запрос1 INTERSECT [ALL] запрос2  
запрос1 EXCEPT [ALL] запрос2
```

Здесь запрос1 и запрос2 – это запросы, в которых могут использоваться все возможности, рассмотренные до этого. Операции над множествами тоже можно вкладывать и соединять, например:

```
запрос1 UNION запрос2 UNION запрос3
```

Этот сложный запрос выполняется так:

```
(запрос1 UNION запрос2) UNION запрос3
```

UNION по сути добавляет результаты второго запроса к результатам первого (хотя никакой порядок возвращаемых строк при этом не гарантируется). Более того, эта операция убирает дублирующиеся строки из результата так же, как это делает DISTINCT, если только не указано UNION ALL.

INTERSECT возвращает все строки, содержащиеся в результате и первого, и второго запроса. Дублирующиеся строки отфильтровываются, если не указано ALL.

EXCEPT возвращает все строки, которые есть в результате первого запроса, но отсутствуют в результате второго. Иногда это называют разницей двух запросов. И здесь дублирующиеся строки отфильтровываются, если не указано ALL.

Чтобы можно было вычислить объединение, пересечение или разницу результатов двух запросов, эти запросы должны быть «совместимыми для

объединения», что означает, что они должны иметь одинаковое число колонок и соответствующие колонки должны быть совместимых типов.

5.5. Сортировка строк

После того как запрос выдал таблицу результатов (после обработки списка выборки), её можно отсортировать. Если сортировка не задана, строки возвращаются в неопределенном порядке. Фактический порядок строк в этом случае будет зависеть от плана соединения и сканирования, а также от порядка данных на диске, поэтому полагаться на него нельзя. Определенный порядок выводимых строк гарантируется, только если этап сортировки задан явно.

Порядок сортировки определяет предложение ORDER BY:

```
SELECT список_выборки
      FROM табличное_выражение
      ORDER BY выражение_сортировки1 [ASC | DESC] [NULLS
{ FIRST | LAST }]
          [, выражение_сортировки2 [ASC | DESC]
[NULLS { FIRST | LAST }] ...]
```

Выражениями сортировки могут быть любые выражения, допустимые в списке выборки запроса, например:

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

Когда указывается несколько выражений, последующие значения позволяют отсортировать строки, в которых совпали все предыдущие значения. Каждое выражение можно дополнить ключевыми словами ASC или DESC, которые выбирают сортировку соответственно по возрастанию или убыванию. По умолчанию принят порядок по возрастанию (ASC). При сортировке по возрастанию сначала идут меньшие значения, где понятие «меньше» определяется оператором <. Подобным образом, сортировка по возрастанию определяется оператором >.

Для определения места значений NULL можно использовать указания NULLS FIRST и NULLS LAST, которые помещают значения NULL соответственно до или после значений не NULL. По умолчанию значения NULL считаются больше любых других, то есть подразумевается NULLS FIRST для порядка DESC и NULLS LAST в

противном случае.

Заметьте, что порядки сортировки определяются независимо для каждой колонки. Например, `ORDER BY x, y DESC` означает `ORDER BY x ASC, y DESC`, и это не то же самое, что `ORDER BY x DESC, y DESC`.

Здесь выражение_сортировки может быть меткой колонки или номером выводимой колонки, как в данном примере:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;  
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

Оба эти запроса сортируют результат по первой колонке. Заметьте, что имя выводимой колонки должно оставаться само по себе, его нельзя использовать в выражении. Например, это ошибка:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;  
-- неправильно
```

Это ограничение позволяет уменьшить неоднозначность. Тем не менее неоднозначность возможна, когда в `ORDER BY` указано простое имя, но оно соответствует и имени выходной колонки, и колонке из табличного выражения. В этом случае используется выходная колонка. Эта ситуация может возникнуть, только когда с помощью `AS` выходной колонке назначается то же имя, что имеет колонка в другой таблице.

`ORDER BY` можно применить к результату комбинации `UNION`, `INTERSECT` и `EXCEPT`, но в этом случае возможна сортировка только по номерам или именам колонок, но не по выражениям.

5.6. LIMIT и OFFSET

Указания `LIMIT` и `OFFSET` позволяют получить только часть строк из тех, что выдал остальной запрос:

```
SELECT список_выборки  
FROM табличное_выражение  
[ ORDER BY ... ]  
[ LIMIT { число | ALL } ] [ OFFSET число ]
```

Если указывается число `LIMIT`, в результате возвращается не больше

заданного числа строк (меньше может быть, если сам запрос выдал меньшее количество строк). `LIMIT ALL` равносильно отсутствию указания `LIMIT`.

`OFFSET` указывает пропустить указанное число строк, прежде чем начать выдавать строки. `OFFSET 0` равносильно отсутствию указания `OFFSET`, а `LIMIT NULL` – отсутствию `LIMIT`. Если указано и `OFFSET`, и `LIMIT`, сначала система пропускает `OFFSET` строк, а затем начинает подсчитывать строки для ограничения `LIMIT`.

Применяя `LIMIT`, важно использовать также предложение `ORDER BY`, чтобы строки результата выдавались в определенном порядке. Иначе будут возвращаться непредсказуемые подмножества строк. Вы можете запросить строки с десятой по двадцатую, но какой порядок вы имеете в виду? Порядок будет неизвестен, если не добавить `ORDER BY`.

Оптимизатор запроса учитывает ограничение `LIMIT`, строя планы выполнения запросов, поэтому вероятнее всего планы (а значит и порядок строк) будут меняться при разных `LIMIT` и `OFFSET`. Таким образом, различные значения `LIMIT/OFFSET`, выбирающие разные подмножества результатов запроса, приведут к несогласованности результатов, если не установить предсказуемую сортировку с помощью `ORDER BY`. Это не ошибка, а неизбежное следствие того, что SQL не гарантирует вывод результатов запроса в некотором порядке, если порядок не определен явно предложением `ORDER BY`.

Строки, пропускаемые согласно предложению `OFFSET`, тем не менее должны вычисляться на сервере. Таким образом, при больших значениях `OFFSET` работает неэффективно.

5.7. Списки `VALUES`

Предложение `VALUES` позволяет создать «постоянную таблицу», которую можно использовать в запросе, не создавая и не наполняя таблицу в БД. Синтаксис предложения:

```
VALUES ( выражение [, ...] ) [, ...]
```

Для каждого списка выражений в скобках создается строка таблицы. Все

списки должны иметь одинаковое число элементов (т.е. число колонок в таблице) и соответствующие элементы во всех списках должны иметь совместимые типы данных. Фактический тип данных колонок результата определяется по тем же правилам, что и для UNION, например:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

вернет таблицу из двух колонок и трех строк. Это равносильно такому запросу:

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

По умолчанию Синергия-БД назначает колонкам таблицы VALUES имена column1, column2 и т.д. Имена колонок не определены в стандарте SQL и в другой СУБД они могут быть другими, поэтому обычно лучше переопределить имена списком псевдонимов, например так:

```
=> SELECT * FROM (VALUES (1, 'one'), (2, 'two'), (3,
'three')) AS t (num,letter);
 num | letter
-----+-----
    1 | one
    2 | two
    3 | three
(3 rows)
```

Синтаксически список VALUES с набором выражений равнозначен:

```
SELECT список_выборки FROM табличное_выражение
```

и допускается везде, где допустим SELECT. Например, вы можете использовать его в составе UNION или добавить к нему определение_сортировки (ORDER BY, LIMIT и/или OFFSET). VALUES чаще всего используется как источник данных для команды INSERT, а также как подзапрос.

За дополнительными сведениями обратитесь к справке VALUES.

5.8. Запросы WITH (общие табличные выражения)

WITH предоставляет способ записывать дополнительные операторы для применения в больших запросах. Эти операторы, которые также называют общими табличными выражениями (Common Table Expressions, CTE), можно представить как определения временных таблиц, существующих только для одного запроса. Дополнительным оператором в предложении WITH может быть SELECT, INSERT, UPDATE или DELETE, а само предложение WITH присоединяется к основному оператору, которым также может быть SELECT, INSERT, UPDATE или DELETE.

5.8.1. SELECT в WITH

Основное предназначение SELECT в предложении WITH заключается в разбиении сложных запросов на простые части. Например, запрос:

```
WITH regional_sales AS (  
    SELECT region, SUM(amount) AS total_sales  
    FROM orders  
    GROUP BY region  
) , top_regions AS (  
    SELECT region  
    FROM regional_sales  
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM  
regional_sales)  
)  
SELECT region,  
    product,  
    SUM(quantity) AS product_units,  
    SUM(amount) AS product_sales  
FROM orders  
WHERE region IN (SELECT region FROM top_regions)  
GROUP BY region, product;
```

выводит итоги по продажам только для передовых регионов. Предложение WITH определяет два дополнительных оператора regional_sales и top_regions так, что результат regional_sales используется в

`top_regions`, а результат `top_regions` используется в основном запросе `SELECT`. Этот пример можно было бы переписать без `WITH`, но тогда нам понадобятся два уровня вложенных подзапросов `SELECT`. Показанным выше способом это можно сделать немного проще.

Необязательное указание `RECURSIVE` превращает `WITH` из простого синтаксического удобства в средство реализации того, что невозможно в стандартном SQL. Используя `RECURSIVE`, запрос `WITH` может обращаться к собственному результату. Очень простой пример, суммирующий числа от 1 до 100:

```
WITH RECURSIVE t(n) AS (  
    VALUES (1)  
    UNION ALL  
    SELECT n+1 FROM t WHERE n < 100  
)  
SELECT sum(n) FROM t;
```

В общем виде рекурсивный запрос `WITH` всегда записывается как не рекурсивная часть, потом `UNION` (или `UNION ALL`), а затем рекурсивная часть, где только в рекурсивной части можно обратиться к результату запроса. Такой запрос выполняется следующим образом:

Вычисление рекурсивного запроса

1) Вычисляется не рекурсивная часть. Для `UNION` (но не `UNION ALL`) отбрасываются дублирующие строки. Все оставшиеся строки включаются в результат рекурсивного запроса и также помещаются во временную рабочую таблицу.

2) Пока рабочая таблица не пуста, повторяются следующие действия:

- вычисляется рекурсивная часть так, что рекурсивная ссылка на сам запрос обращается к текущему содержимому рабочей таблицы. Для `UNION` (но не `UNION ALL`) отбрасываются дублирующие строки и строки, дублирующие ранее полученные. Все оставшиеся строки включаются в результат рекурсивного запроса и также помещаются во временную промежуточную таблицу;

- содержимое рабочей таблицы заменяется содержимым промежуточной таблицы, а затем промежуточная таблица очищается.

Примечание. Строго говоря, этот процесс является итерационным, а не рекурсивным, но комитетом по стандартам SQL был выбран термин `RECURSIVE`.

В показанном выше примере в рабочей таблице на каждом этапе содержится всего одна строка и в ней последовательно накапливаются значения от 1 до 100. На сотом шаге, благодаря условию `WHERE`, не возвращается ничего, так что вычисление запроса завершается.

Рекурсивные запросы обычно применяются для работы с иерархическими или древовидными структурами данных. В качестве полезного примера можно привести запрос, находящий все непосредственные и косвенные составные части продукта, используя только таблицу с прямыми связями:

```
WITH RECURSIVE included_parts(sub_part, part, quantity)
AS (
    SELECT sub_part, part, quantity FROM parts WHERE
    part = 'our_product'
    UNION ALL
    SELECT p.sub_part, p.part, p.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

Работая с рекурсивными запросами, важно обеспечить, чтобы рекурсивная часть запроса в конце концов не выдала никаких кортежей (строк), в противном случае цикл будет бесконечным. Иногда для этого достаточно применять `UNION` вместо `UNION ALL`, так как при этом будут отбрасываться строки, которые уже есть в результате. Однако часто в цикле выдаются строки, не совпадающие полностью с предыдущими: в таких случаях может иметь смысл проверить одно или несколько полей, чтобы определить, не была ли текущая точка достигнута раньше. Стандартный способ решения подобных задач — вычислить массив с уже

обработанными значениями. Например, рассмотрите следующий запрос, просматривающий таблицу graph по полю link:

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
    SELECT g.id, g.link, g.data, 1
    FROM graph g
    UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1
    FROM graph g, search_graph sg
    WHERE g.id = sg.link
)
SELECT * FROM search_graph;
```

Этот запрос заикнется, если связи link содержат циклы. Так как нам нужно получать в результате «depth», одно лишь изменение UNION ALL на UNION не позволит избежать заикливания. Вместо этого мы должны как-то определить, что уже достигали текущей строки, пройдя некоторый путь. Для этого мы добавляем две колонки path и cycle и получаем запрос, защищенный от заикливания:

```
WITH RECURSIVE search_graph(id, link, data, depth, path,
cycle) AS (
    SELECT g.id, g.link, g.data, 1,
    ARRAY[g.id],
    false
    FROM graph g
    UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
    path || g.id,
    g.id = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

Помимо предотвращения циклов, значения массива часто бывают полезны сами по себе для представления «пути», приведшего к определенной строке.

В общем случае, когда для выявления цикла нужно проверять несколько полей, следует использовать массив строк. Например, если нужно сравнить поля f1 и f2:

```
WITH RECURSIVE search_graph(id, link, data, depth, path,
cycle) AS (
    SELECT g.id, g.link, g.data, 1,
        ARRAY[ROW(g.f1, g.f2)],
        false
    FROM graph g
    UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
        path || ROW(g.f1, g.f2),
        ROW(g.f1, g.f2) = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

Примечания:

1. Часто для распознавания цикла достаточно одного поля и тогда ROW() можно опустить. При этом будет использоваться не массив данных составного типа, а простой массив, что более эффективно.

2. Этот алгоритм рекурсивного вычисления запроса выдает в результате узлы, упорядоченные по пути погружения. Чтобы получить результаты, отсортированные по глубине, можно добавить во внешний запрос ORDER BY по колонке «path», полученной, как показано выше.

Для тестирования запросов, которые могут заикливаться, есть хороший прием – добавить LIMIT в родительский запрос. Например, следующий запрос заиклится, если не добавить предложение LIMIT:

```
WITH RECURSIVE t(n) AS (
    SELECT 1
    UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

Но в данном случае этого не происходит, так как в Синергия-БД запрос WITH

выдает столько строк, сколько фактически принимает родительский запрос. В производственной среде использовать этот прием не рекомендуется, так как другие системы могут вести себя по-другому. Кроме того, это не будет работать, если внешний запрос сортирует результаты рекурсивного запроса или соединяет их с другой таблицей, так как в подобных случаях внешний запрос обычно все равно выбирает результат запроса `WITH` полностью.

Запросы `WITH` имеют полезное свойство – они вычисляются только раз для всего родительского запроса, даже если этот запрос или соседние запросы `WITH` обращаются к ним неоднократно. Таким образом, сложные вычисления, результаты которых нужны в нескольких местах, можно выносить в запросы `WITH` в целях оптимизации. Кроме того, такие запросы позволяют избежать нежелательных вычислений функций с побочными эффектами. Однако есть и обратная сторона – оптимизатор не может распространить ограничения родительского запроса на запрос `WITH` так, как он делает это для обычного подзапроса. Запрос `WITH` обычно выполняется буквально и возвращает все строки, включая те, что потом может отбросить родительский запрос. Но как было сказано выше, вычисление может остановиться раньше, если в ссылке на этот запрос затребуется только ограниченное число строк.

Примеры выше показывают только предложение `WITH` с `SELECT`, но таким же образом его можно использовать с командами `INSERT`, `UPDATE` и `DELETE`. В каждом случае он по сути создает временную таблицу, к которой можно обратиться в основной команде.

5.8.2. Изменение данных в `WITH`

В предложении `WITH` можно также использовать операторы, изменяющие данные (`INSERT`, `UPDATE` или `DELETE`). Это позволяет выполнять в одном запросе сразу несколько разных операций, например:

```
WITH moved_rows AS (  
    DELETE FROM products  
    WHERE
```

```
«date» >= '2010-10-01' AND
«date» < '2010-11-01'
RETURNING *
)
INSERT INTO products_log
SELECT * FROM moved_rows;
```

Этот запрос фактически перемещает строки из `products` в `products_log`. Оператор `DELETE` в `WITH` удаляет указанные строки из `products` и возвращает их содержимое в предложении `RETURNING`; а затем главный запрос читает это содержимое и вставляет в таблицу `products_log`.

Следует заметить, что предложение `WITH` в данном случае присоединяется к оператору `INSERT`, а не к `SELECT`, вложенному в `INSERT`. Это необходимо, так как `WITH` может содержать операторы, изменяющие данные, только на верхнем уровне запроса. Однако при этом применяются обычные правила видимости `WITH`, так что к результату `WITH` можно обратиться и из вложенного оператора `SELECT`.

Операторы, изменяющие данные, в `WITH` обычно дополняются предложением `RETURNING`, как показано в этом примере. Важно понимать, что временная таблица, которую можно будет использовать в остальном запросе, создается из результата `RETURNING`, а не целевой таблицы оператора. Если оператор, изменяющий данные в `WITH`, не дополнен предложением `RETURNING`, временная таблица не создается и обращаться к ней в остальном запросе нельзя. Однако такой запрос все равно будет выполнен. Например, допустим следующий не очень практичный запрос:

```
WITH t AS (
    DELETE FROM foo
)
DELETE FROM bar;
```

Он удалит все строки из таблиц `foo` и `bar`. При этом число задействованных строк, которое получит клиент, будет подсчитываться только по строкам, удаленным из `bar`.

Рекурсивные ссылки в операторах, изменяющих данные, не допускаются. В

некоторых случаях это ограничение можно обойти, обратившись к конечному результату рекурсивного WITH, например так:

```
WITH RECURSIVE included_parts(sub_part, part) AS (  
    SELECT sub_part, part FROM parts WHERE part =  
    'our_product'  
    UNION ALL  
    SELECT p.sub_part, p.part  
    FROM included_parts pr, parts p  
    WHERE p.part = pr.sub_part  
)  
DELETE FROM parts  
    WHERE part IN (SELECT part FROM included_parts);
```

Этот запрос удаляет все непосредственные и косвенные составные части продукта.

Операторы, изменяющие данные в WITH, выполняются только один раз и всегда полностью, вне зависимости от того, принимает ли их результат основной запрос. Заметьте, что это отличается от поведения SELECT в WITH: как говорилось в предыдущем разделе, SELECT выполняется только до тех пор, пока его результаты востребованы основным запросом.

Вложенные операторы в WITH выполняются одновременно друг с другом и с основным запросом. Таким образом, порядок, в котором операторы в WITH будут фактически изменять данные, непредсказуем. Все эти операторы выполняются с одним снимком данных, так что они не могут «видеть», как каждый из них меняет целевые таблицы. Это уменьшает эффект непредсказуемости фактического порядка изменения строк и означает, что RETURNING – единственный вариант передачи изменений от вложенных операторов WITH основному запросу. Например, в данном случае:

```
WITH t AS (  
    UPDATE products SET price = price * 1.05  
    RETURNING *  
)
```



```
SELECT * FROM products;
```

внешний оператор SELECT выдаст цены, которые были до действия UPDATE, тогда как в запросе:

```
WITH t AS (  
    UPDATE products SET price = price * 1.05  
    RETURNING *)  
SELECT * FROM t;
```

внешний SELECT выдаст измененные данные.

Неоднократное изменение одной и той же строки в рамках одного оператора не поддерживается. Иметь место будет только одно из нескольких изменений и надежно определить, какое именно, часто довольно сложно (а иногда и вовсе невозможно). Это так же касается случая, когда строка удаляется и изменяется в том же операторе: в результате может быть выполнено только обновление. Поэтому в общем случае следует избегать подобного наложения операций. В частности, избегайте подзапросов WITH, которые могут повлиять на строки, изменяемые основным оператором или операторами, вложенные в него. Результат действия таких запросов будет непредсказуемым.

В настоящее время, для оператора, изменяющего данные в WITH, в качестве целевой нельзя использовать таблицу, для которой определено условное правило или правило ALSO или INSTEAD, если оно состоит из нескольких операторов.

Примечание. На деле Синергия-БД определяет порядок сортировки для ASC и DESC по классу оператора В-дерева по умолчанию для типа данных выражения. Обычно типы данных создаются так, что этому порядку соответствуют операторы < и >, но возможно разработать собственный тип данных, который будет вести себя по-другому.

6. ТИПЫ ДАННЫХ

Синергия-БД предоставляет пользователям богатый ассортимент встроенных типов данных. Кроме того, пользователи могут создавать свои типы в Синергия-БД, используя команду CREATE TYPE.

Таблица 6.1 содержит все встроенные типы данных общего пользования. Многие из альтернативных имен, приведенных в колонке «Псевдонимы», используются внутри Синергия-БД по историческим причинам. В этот список не включены некоторые устаревшие типы и типы для внутреннего применения.

Таблица 6.1 – Типы данных

Имя	Псевдонимы	Описание
bigint	int8	знаковое целое из 8 байт
bigserial	serial8	восьмибайтное целое с автоувеличением
bit [(n)]		битовая строка фиксированной длины
bit varying [(n)]	varbit [(n)]	битовая строка переменной длины
boolean	bool	логическое значение (true/false)
box		прямоугольник в плоскости
bytea		двоичные данные («массив байт»)
character [(n)]	char [(n)]	символьная строка фиксированной длины
character varying [(n)]	varchar [(n)]	символьная строка переменной длины
cidr		сетевой адрес IPv4 или IPv6
circle		круг в плоскости
date		календарная дата (год, месяц, день)
double precision	float8	число двойной точности с плавающей точкой (8 байт)
inet		адрес узла IPv4 или IPv6
integer	int, int4	знаковое четырехбайтное целое
interval [поля] [(p)]		интервал времени
json		текстовые данные JSON
jsonb		двоичные данные JSON, разобранные
line		прямая в плоскости
lseg		отрезок в плоскости
macaddr		MAC-адрес
macaddr8		Адрес MAC (Media Access Control) (в формате EUI-64)
money		денежная сумма
numeric [(p, s)]	decimal [(p, s)]	вещественное число заданной точности
path		геометрический путь в плоскости
pg_lsn		Последовательный номер в журнале PostgreSQL
point		геометрическая точка в плоскости
polygon		замкнутый геометрический путь в плоскости
real	float4	число одинарной точности с плавающей точкой (4 байта)
smallint	int2	знаковое двухбайтное целое
smallserial	serial2	двухбайтное целое с автоувеличением
serial	serial4	четырёхбайтное целое с автоувеличением
text		символьная строка переменной длины
time [(p)] [without time zone]		время суток (без часового пояса)

Окончание таблицы 6.1

Имя	Псевдонимы	Описание
time [(p)] with time zone	timetz	время суток с учетом часового пояса
timestamp [(p)] [without time zone]		дата и время (без часового пояса)
timestamp [(p)] with time zone	timestampz	дата и время с учетом часового пояса
tsquery		запрос текстового поиска
tsvector		документ для текстового поиска
txid_snapshot		снимок идентификатора транзакций
uuid		универсальный уникальный идентификатор
xml		XML-данные

В стандарте SQL описаны следующие типы (или их имена): bigint, bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (с часовым поясом и без), timestamp (с часовым поясом и без), xml.

Каждый тип данных имеет внутреннее представление, скрытое функциями ввода и вывода. При этом многие встроенные типы стандартны и имеют очевидные внешние форматы. Однако есть типы, уникальные для Синергия-БД, например геометрические пути, и есть типы, которые могут иметь разные форматы, например, дата и время. Некоторые функции ввода и вывода не являются в точности обратными друг к другу, то есть результат функции вывода может не совпадать со входным значением из-за потери точности.

6.1. Числовые типы

Числовые типы включают двух-, четырех- и восьмибайтные целые, четырех- и восьмибайтные числа с плавающей точкой, а также десятичные числа с задаваемой точностью. Все эти типы перечислены в таблице 6.2.

Таблица 6.2 – Числовые типы

Имя	Размер	Описание	Диапазон
smallint	2 байта	целое в небольшом диапазоне	-32768 .. +32767
integer	4 байта	типичный выбор для целых чисел	-2147483648 .. +2147483647
bigint	8 байт	целое в большом диапазоне	-9223372036854775808 .. 9223372036854775807
decimal	переменный	вещественное число с указанной точностью	до 131072 цифр до десятичной точки и до 16383 – после
numeric	переменный	вещественное число с указанной точностью	до 131072 цифр до десятичной точки и до 16383 – после
real	4 байта	вещественное число с переменной точностью	точность в пределах 6 десятичных цифр

Окончание таблицы 6.2

Имя	Размер	Описание	Диапазон
double precision	8 байт	вещественное число с переменной точностью	точность в пределах 15 десятичных цифр
smallserial	2 байта	небольшое целое с автоувеличением	1 .. 32767
serial	4 байта	целое с автоувеличением	1 .. 2147483647
bigserial	8 байт	большое целое с автоувеличением	1 .. 9223372036854775807

Синтаксис констант числовых типов описан в подразделе 2.1.2. Для этих типов определен полный набор соответствующих арифметических операторов и функций. Подробнее эти типы описаны в следующих разделах.

6.1.1. Целочисленные типы

Типы `smallint`, `integer` и `bigint` хранят целые числа, то есть числа без дробной части, имеющие разные допустимые диапазоны. Попытка сохранить значение, выходящее за рамки диапазона, приведет к ошибке.

Чаще всего используется тип `integer`, как наиболее сбалансированный выбор ширины диапазона, размера и быстродействия. Тип `smallint` обычно применяется, только когда крайне важно уменьшить размер данных на диске. Тип `bigint` предназначен для тех случаев, когда числа не умещаются в диапазон типа `integer`.

В SQL определены только типы `integer` (или `int`), `smallint` и `bigint`. Имена типов `int2`, `int4` и `int8` выходят за рамки стандарта, хотя могут работать и в некоторых других СУБД.

6.1.2. Числа с произвольной точностью

Тип `numeric` позволяет хранить числа с очень большим количеством цифр и выполнять вычисления точно. Он рекомендуется для хранения денежных сумм и других величин, где важна точность. Вычисления с типом `numeric` дают точные результаты, где это возможно, например, при сложении, вычитании и умножении. Однако арифметические операции со значениями `numeric` выполняются гораздо медленнее, чем с целыми числами или с типами с плавающей точкой, описанными в следующем разделе.

Ниже мы используем следующие термины: масштаб значения `numeric` определяет количество десятичных цифр в дробной части, справа от десятичной

точки, а точность – общее количество значимых цифр в числе, т.е. количество цифр по обе стороны десятичной точки. Например, число 23.5141 имеет точность 6 и масштаб 4. Целочисленные значения можно считать числами с масштабом 0.

Для колонки типа `numeric` можно настроить и максимальную точность, и максимальный масштаб. Колонка типа `numeric` объявляется следующим образом:

```
NUMERIC (точность, масштаб)
```

Точность должна быть положительной, а масштаб положительным или равным нулю. Альтернативный вариант

```
NUMERIC (точность)
```

устанавливает масштаб 0. Форма:

```
NUMERIC
```

без указания точности и масштаба создает колонку, в которой можно сохранять числовые значения любой точности и масштаба в пределах, поддерживаемых системой. В колонке этого типа входные значения не будут приводиться к какому-либо масштабу, тогда как в колонках `numeric` с явно заданным масштабом значения подгоняются под этот масштаб. Стандарт SQL утверждает, что по умолчанию должен устанавливаться масштаб 0, т.е. значения должны приводиться к целым числам. Однако мы считаем это не очень полезным. Если для вас важна переносимость, всегда указывайте точность и масштаб явно.

Примечание. Максимально допустимая точность, которую можно указать в объявлении типа, равна 1000; если же использовать `NUMERIC` без указания точности, действуют ограничения, описанные в таблице 6.2.

Если масштаб значения, которое нужно сохранить, превышает объявленный масштаб колонки, система округлит его до заданного количества цифр после точки. Если же после этого количество цифр слева в сумме с масштабом превысит объявленную точность, произойдет ошибка.

Числовые значения физически хранятся без каких-либо дополняющих нулей слева или справа. Таким образом, объявляемые точность и масштаб колонки определяют максимальный, а не фиксированный размер хранения. В этом смысле тип `numeric` больше похож на тип `varchar(n)`, чем на `char(n)`. Действительный размер хранения такого значения складывается из двух байт для каждой группы из четырех

цифр и дополнительных трех-восьми байт.

Помимо обычных чисел тип `numeric` позволяет сохранить специальное значение `NaN`, что означает «not-a-number» (не число). Любая операция с `NaN` выдает в результате тоже `NaN`. Записывая это значение в виде константы в команде SQL, его нужно заключать в апострофы, например так: `UPDATE table SET x = 'NaN'`. Регистр символов в строке `NaN` не важен.

Примечание. В большинстве реализаций «не-число» (`NaN`) считается не равным любому другому значению (в том числе и самому `NaN`). Чтобы значения `numeric` можно было сортировать и использовать в древовидных индексах, Синергия-БД считает, что значения `NaN` равны друг другу и при этом больше любых числовых значений (не `NaN`).

Типы `decimal` и `numeric` равнозначны. Оба эти типа описаны в стандарте SQL.

При округлении значений тип `numeric` выдаёт число, большее по модулю, тогда как (на большинстве платформ) типы `real` и `double precision` выдают ближайшее чётное число, например:

```
SELECT x,
       round(x::numeric) AS num_round,
       round(x::double precision) AS dbl_round
FROM generate_series(-3.5, 3.5, 1) as x;
```

x		num_round		dbl_round
-3.5		-4		-4
-2.5		-3		-2
-1.5		-2		-2
-0.5		-1		-0
0.5		1		0
1.5		2		2
2.5		3		2
3.5		4		4

(8 rows)

6.1.3. Типы с плавающей точкой

Типы данных `real` и `double precision` хранят приближенные числовые значения

с переменной точностью. На практике эти типы обычно реализуют Стандарт IEEE 754 для двоичной арифметики с плавающей точкой (с одинарной и двойной точностью соответственно), в той мере, в какой его поддерживают процессор, операционная система и компилятор.

Неточность здесь выражается в том, что некоторые значения, которые нельзя преобразовать во внутренний формат, сохраняются приближенно, так что полученное значение может несколько отличаться от записанного. Управление подобными ошибками и их распространение в процессе вычислений является предметом изучения целого раздела математики и компьютерной науки, и здесь не рассматривается. Мы отметим только следующее:

- если вам нужна точность при хранении и вычислениях (например, для денежных сумм), используйте вместо этого тип `numeric`;
- если вы хотите выполнять с этими типами сложные вычисления, имеющие большую важность, тщательно изучите реализацию операций в вашей среде и особенно поведение в крайних случаях (бесконечность, антипереполнение);
- проверка равенства двух чисел с плавающей точкой может не всегда давать ожидаемый результат.

На большинстве платформ тип `real` может сохранить значения в пределах от $1E-37$ до $1E+37$ с точностью не меньше 6 десятичных цифр. Тип `double precision` предлагает диапазон значений от $1E-307$ до $1E+308$ и точностью не меньше 15 цифр. Попытка сохранить слишком большие или слишком маленькие значения приведет к ошибке. Если точность вводимого числа слишком велика, оно будет округлено. При попытке сохранить число, близкое к 0, но непредставимое как отличное от 0, произойдет ошибка антипереполнения.

Примечание. Параметр `extra_float_digits` определяет количество дополнительных значащих цифр при преобразовании значения с плавающей точкой в текст для вывода. Со значением по умолчанию (0) вывод будет одинаковым на всех платформах, поддерживаемых Синергия-БД.

В дополнение к обычным числовым значениям типы с плавающей точкой могут содержать следующие специальные значения:

`Infinity`
`-Infinity`

NaN

Они представляют особые значения, описанные в IEEE 754, соответственно: «бесконечность», «минус бесконечность» и «не число». (Но компьютерах, где арифметика с плавающей точкой не соответствует стандарту IEEE 754, эти значения, вероятно, не будут работать должным образом.) Записывая эти значения в виде констант в команде SQL, их нужно заключать в апострофы, например так: `UPDATE table SET x = 'Infinity'`. Регистр символов в этих строках не важен.

Примечание. Согласно IEEE754, NaN не должно считаться равным любому другому значению с плавающей точкой (в том числе и самому NaN). Чтобы значения с плавающей точкой можно было сортировать и использовать в древовидных индексах, Синергия-БД считает, что значения NaN равны друг другу, и при этом больше любых числовых значений (не NaN).

Синергия-БД также поддерживает форматы `float` и `float(p)`, оговоренные в стандарте SQL, для указания неточных числовых типов. Здесь `p` определяет минимально допустимую точность в двоичных цифрах. Синергия-БД воспринимает запись от `float(1)` до `float(24)` как выбор типа `real`, а запись от `float(25)` до `float(53)` как выбор типа `double precision`. Значения `p` вне допустимого диапазона вызывают ошибку. Если `float` указывается без точности, подразумевается тип `double precision`.

Примечание. Предположение, что типы `real` и `double precision` имеют в мантиссе 24 и 53 бита соответственно, справедливо для всех реализаций плавающей точки по стандарту IEEE. На платформах, не поддерживающих IEEE, размер мантиссы может несколько отличаться, но для простоты диапазоны `p` везде считаются одинаковыми.

6.1.4. Последовательные типы

В этом разделе описывается специфичный для Синергия-БД способ создания столбца с автоувеличением. Другой способ, соответствующий стандарту SQL, заключается в использовании столбцов идентификации и рассматривается в описании `CREATE TABLE`.

Типы данных `smallserial`, `serial` и `bigserial` не являются настоящими типами, а представляют собой просто удобное средство для создания колонок с уникальными идентификаторами (подобное свойству `AUTO_INCREMENT` в некоторых СУБД). В текущей реализации запись:

```
CREATE TABLE имя_таблицы (
```



```
имя_колонки SERIAL  
);
```

равнозначна следующим командам:

```
CREATE SEQUENCE имя_таблицы_имя_колонки_seq AS integer;  
CREATE TABLE имя_таблицы (  
    имя_колонки integer NOT NULL DEFAULT  
    nextval('имя_таблицы_имя_колонки_seq')  
);  
  
ALTER SEQUENCE имя_таблицы_имя_колонки_seq OWNED BY  
имя_таблицы.имя_колонки;
```

То есть при определении такого типа создается целочисленная колонка со значением по умолчанию, извлекаемым из генератора последовательности. Чтобы в колонку нельзя было вставить NULL, в её определение добавляется ограничение NOT NULL. Во многих случаях также имеет смысл добавить для этой колонки ограничения UNIQUE или PRIMARY KEY для защиты от ошибочного добавления дублирующихся значений, но автоматически это не происходит. Последняя команда определяет, что последовательность «принадлежит» колонке, так что она будет удалена при удалении колонки или таблицы.

Примечание. Так как типы smallserial, serial и bigserial реализованы через последовательности, в числовом ряду значений колонки могут образовываться пропуски (или «дыры»), даже если никакие строки не удалялись. Значение, выделенное из последовательности, считается «задействованным», даже если строку с этим значением не удалось вставить в таблицу. Это может произойти, например, при откате транзакции, добавляющей данные.

Чтобы вставить в колонку serial следующее значение последовательности, ей нужно присвоить значение по умолчанию. Это можно сделать, либо исключив её из списка колонок в операторе INSERT, либо с помощью ключевого слова DEFAULT.

Имена типов serial и serial4 равнозначны: они создают колонки integer. Так же являются синонимами имена bigserial и serial8, но они создают колонки bigint. Тип bigserial следует использовать, если за все время жизни таблицы планируется использовать больше чем 231 значений. И наконец, синонимами являются имена типов smallserial и serial2, но они создают колонку smallint.

Последовательность, созданная для колонки serial, автоматически удаляется

при удалении связанной колонки. Последовательность можно удалить и отдельно от колонки, но при этом также будет удалено определение значения по умолчанию.

6.2. Денежные типы

Тип `money` хранит денежную сумму с фиксированной дробной частью; см. таблицу 6.3. Точность дробной части определяется на уровне базы данных параметром `lc_monetary`. Для диапазона, показанного в таблице, предполагается, что число содержит два знака после запятой. Входные данные могут быть записаны по-разному, в том числе в виде целых и дробных чисел, а также в виде строки в денежном формате, например `'$1,000.00'`. Выводятся эти значения обычно в денежном формате, зависящем от региональных настроек.

Таблица 6.3 – Денежные типы

Имя	Размер	Описание	Диапазон
<code>money</code>	8 байт	денежная сумма	-92233720368547758.08 .. +92233720368547758.07

Так как выводимые значения этого типа зависят от региональных настроек, попытка загрузить данные типа `money` в базу данных с другим параметром `lc_monetary` может быть неудачной. Во избежание подобных проблем, прежде чем восстанавливать копию в новую базу данных, убедитесь в том, что параметр `lc_monetary` в этой базе данных имеет то же значение, что и в исходной.

Значения типов `numeric`, `int` и `bigint` можно привести к типу `money`. Преобразования типов `real` и `double precision` так же возможны через тип `numeric`, например:

```
SELECT '12.34'::float8::numeric::money;
```

Однако использовать числа с плавающей точкой для денежных сумм не рекомендуется из-за возможных ошибок округления.

Значение `money` можно привести к типу `numeric` без потери точности. Преобразование в другие типы может быть неточным и также должно выполняться в два этапа:

```
SELECT '52093.89'::money::numeric::float8;
```

При делении значения типа `money` на целое число выполняется отбрасывание дробной части и получается целое, ближайшее к нулю. Чтобы получить результат с

округлением, выполните деление значения с плавающей точкой или приведите значение типа money к numeric до деления, а затем приведите результат к типу money. Последний вариант предпочтительнее, так как исключает риск потери точности. Когда значение money делится на другое значение money, результатом будет значение типа double precision (то есть просто число, не денежная величина); денежные единицы измерения при делении сокращаются.

6.3. Символьные типы

Таблица 6.4 – Символьные типы

Имя	Описание
<code>character varying(n)</code> , <code>varchar(n)</code>	строка ограниченной переменной длины
<code>character(n)</code> , <code>char(n)</code>	строка фиксированной длины, дополненная пробелами
<code>text</code>	строка неограниченной переменной длины

В таблице 6.4 перечислены символьные типы общего назначения, доступные в Синергия-БД.

SQL определяет два основных символьных типа: `character varying(n)` и `character(n)`, где `n` – положительное число. Оба эти типа могут хранить текстовые строки длиной до `n` символов (не байт). Попытка сохранить в колонке такого типа более длинную строку приведет к ошибке, если только все лишние символы не являются пробелами (тогда они будут усечены до максимально допустимой длины). Это несколько странное исключение продиктовано стандартом SQL. Если длина сохраняемой строки оказывается меньше объявленной, значения типа `character` будут дополняться пробелами; а тип `character varying` просто сохранит короткую строку.

При попытке явно привести значение к типу `character varying(n)` или `character(n)`, часть строки, выходящая за границу в `n` символов, удаляется, не вызывая ошибки. Это также продиктовано стандартом SQL.

Записи `varchar(n)` и `char(n)` являются синонимами `character varying(n)` и `character(n)`, соответственно. Записи `character` без указания длины соответствует `character(1)`. Если же длина не указывается для `character varying`, этот тип будет принимать строки любого размера. Это поведение является расширением Синергия-БД.

Помимо этого, Синергия-БД предлагает тип `text`, в котором можно хранить

строки произвольной длины. Хотя тип `text` не описан в стандарте SQL, его поддерживают и некоторые другие СУБД SQL.

Значения типа `character` физически дополняются пробелами до `n` символов и хранятся, а затем отображаются в таком виде. Однако при сравнении двух значений типа `character` дополняющие пробелы считаются незначащими и игнорируются. С правилами сортировки, где пробельные символы являются значащими, это поведение может приводить к неожиданным результатам, например `SELECT 'a'::CHAR(2) collate «C» < 'a\n'::CHAR(2)` вернет `true` (условие будет истинным), хотя в локали `C` символ пробела считается больше символа новой строки. При преобразовании значения `character` к другому символьному типу дополняющие пробелы отбрасываются. Заметьте, что эти пробелы несут смысловую нагрузку в типах `character varying` и `text`, а также для проверок по шаблонам `LIKE` и регулярным выражениям.

Для хранения короткой строки (до 126 байт) требуется дополнительный 1 байт плюс размер самой строки, включая дополняющие пробелы для типа `character`. Для строк длиннее требуется не 1, а 4 дополнительных байта. Система может автоматически сжимать длинные строки, так что физический размер на диске может быть меньше. Очень длинные текстовые строки переносятся в отдельные таблицы, чтобы они не замедляли работу с другими колонками. В любом случае, максимально возможный размер строки составляет около 1 ГБ. Допустимое значение `n` в объявлении типа данных меньше этого числа. Это объясняется тем, что в зависимости от кодировки каждый символ может занимать несколько байт. Если вы желаете сохранять строки без определенного предела длины, используйте типы `text` или `character varying` без указания длины, а не задавайте какое-либо большое максимальное значение.

Примечание. По быстродействию эти три типа практически не отличаются друг от друга, не считая большего размера хранения для типа с дополняющими пробелами и нескольких машинных операций для проверки длины при сохранении строк в колонке с ограниченной длиной. Хотя в некоторых СУБД тип `character(n)` работает быстрее других, в Синергия-БД это не так; на деле `character(n)` обычно оказывается медленнее остальных типов из-за большего размера данных и более медленной сортировки.. В большинстве случаев вместо него лучше применять `text`

или character varying.

Подробнее синтаксис текстовых строк описан в подразделе 2.1.2.1. Кодировка, используемая для хранения текстовых строк, определяется набором символов, выбранным для базы данных.

Пример использование символьных типов:

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- (1)
```

a		char_length
-----+		-----
ok		2

```
CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good ');
INSERT INTO test2 VALUES ('too long');
ОШИБКА: значение не помещается в тип character
varying(5)
```

```
INSERT INTO test2 VALUES ('too long'::varchar(5)); --
явное усечение
```

```
SELECT b, char_length(b) FROM test2;
```

b		char_length
-----+		-----
ok		2
good		5
too l		5

В Синергия-БД есть ещё два символьных типа фиксированной длины, приведенные в таблице 6.5. Тип name создан только для хранения идентификаторов во внутренних системных таблицах и не предназначен для обычного применения пользователями. В настоящее время его длина составляет 64 байта (63 ASCII-символа плюс конечный знак), но в исходном коде С она задается константой NAMEDATALEN. Эта константа определяется во время компиляции (и её можно менять в особых случаях), а кроме того, максимальная длина по умолчанию может

быть увеличена в следующих версиях. Тип «char» (обратите внимание на кавычки) отличается от char(1) тем, что он фактически хранится в одном байте. Он используется во внутренних системных таблицах для простых перечислений.

Таблица 6.5 – Специальные символьные типы

Имя	Размер	Описание
«char»	1 байт	внутренний однобайтный тип
name	64 байта	внутренний тип для имен объектов

6.4. Двоичные типы данных

Для хранения двоичных данных предназначен тип `bytea`; см. таблицу 6.6.

Таблица 6.6 – Двоичные типы данных

Имя	Размер	Описание
<code>bytea</code>	1 или 4 байта плюс сама двоичная строка	двоичная строка переменной длины

Двоичные строки представляют собой последовательность октетов (байт) и имеют два отличия от текстовых строк. Во-первых, в двоичных строках можно хранить байты с кодом 0 и другими «непечатаемыми» значениями (обычно это значения вне диапазона 32..126). В текстовых строках нельзя сохранять нулевые байты, а также значения и последовательности значений, не соответствующие выбранной кодировке базы данных. Во-вторых, в операциях с двоичными строками обрабатываются байты в чистом виде, тогда как текстовые строки обрабатываются в зависимости от языковых настроек. То есть, двоичные строки больше подходят для данных, которые программист видит как «просто байты», а символьные строки – для хранения текста.

Тип `bytea` поддерживает два формата ввода и вывода: традиционный для Синергия-БД формат «спецпоследовательностей» и «шестнадцатеричный». Входные данные принимаются в обоих форматах, а формат выходных данных зависит от параметра конфигурации `bytea_output`; по умолчанию выбран шестнадцатеричный.

Стандарт SQL определяет другой тип двоичных данных, `BLOB` (BINARY LARGE OBJECT, большой двоичный объект). Его входной формат отличается от форматов `bytea`, но функции и операторы в основном те же.

6.4.1. Шестнадцатеричный формат `bytea`

В «шестнадцатеричном» формате двоичные данные кодируются двумя

шестнадцатеричными цифрами на байт, при этом первая цифра соответствует старшим 4 битам. К полученной строке добавляется префикс \x (чтобы она отличалась от формата спецпоследовательности). В некоторых контекстах обратную косую черту нужно выделить, продублировав ее, в тех же случаях это нужно сделать для формата спецпоследовательности; подробнее это описано ниже. Шестнадцатеричные цифры могут быть в любом регистре, а между парами цифр допускаются пробельные символы (но не внутри пары и не в начале последовательности \x). Этот формат совместим со множеством внешних приложений и протоколов, к тому же обычно преобразуется быстрее, поэтому предпочтительнее использовать его.

Пример.

```
SELECT E'\\xDEADBEEF';
```

6.4.2. Формат спецпоследовательностей bytea

Формат «спецпоследовательностей» традиционно использовался в Синергия-БД для значений типа bytea. В нем двоичная строка представляется в виде последовательности ASCII-символов, а байты, непредставимые в виде ASCII-символов, передаются в виде спецпоследовательностей. Этот формат может быть удобен, если с точки зрения приложения представление байт в виде символов имеет смысл. Но на практике это обычно создает путаницу, так как двоичные и символьные строки могут выглядеть одинаково, а кроме того выбранный механизм спецпоследовательностей довольно неуклюж. Поэтому в новых приложениях этот формат обычно не стоит использовать.

Передавая значения bytea в формате спецпоследовательности, байты с определенными значениями необходимо записывать специальным образом, хотя так можно записывать и все значения. В общем виде для этого значение байта нужно преобразовать в трехзначное восьмеричное число и добавить перед ним обратную косую черту (и продублировать ее, если значение записывается в текстовой спецстроке). Саму обратную косую черту (символ с кодом 92) можно записать в виде двух таких символов. В таблице 6.7 перечислены символы, которые нужно записывать спецпоследовательностями, и приведены альтернативные варианты

записи там, где они возможны.

Таблица 6.7 – Спецпоследовательности записи значений bytea

Десятичное значение байта	Описание	Спецпоследовательность ввода	Пример	Выводимое представление
0	нулевой байт	'\000'	SELECT '\000'::bytea;	\x00
39	апостроф	'''' или '\047'	SELECT ''''::bytea;	\x27
92	обратная косая черта	'\\' или '\134'	SELECT '\\'\::bytea;	\x5c
от 0 до 31 и от 127 до 255	«непечатаемые» байты	E'\xxx' (восьмеричное значение)	SELECT '\001'::bytea;	\x01

Требования экранирования непечатаемых символов определяются языковыми стандартами. Иногда такие символы могут восприниматься и без спецпоследовательностей.

Апострофы должны дублироваться, потому что это обязательно для любой текстовой строки в команде SQL. При общем разборе текстовой строки внешние апострофы убираются, а каждая пара внутренних сводится к одному символу. Таким образом, функция ввода bytea видит всего один апостроф, который она обрабатывает как обычный символ в данных. Дублировать же обратную косую черту при вводе bytea не требуется: этот символ считается особым и меняет поведение функции ввода.

В некоторых контекстах обратная косая черта должна дублироваться (относительно примеров выше), так как при общем разборе строковых констант пара таких символов будет сведена к одному.

Данные Bytea по умолчанию выводятся в шестнадцатеричном формате (hex). Если поменять значение bytea_output на escape, «непечатаемые» байты представляются в виде соответствующих трёхзначных восьмеричных значений, которые предваряются одной обратной косой чертой. Большинство «печатаемых» байтов представляются обычными символами из клиентского набора символов, например:

```
SET bytea_output = 'escape';
```

```
SELECT 'abc \153\154\155 \052\251\124'::bytea;
bytea
```

abc klm *\251T

Таблица 6.8 – Спецпоследовательности выходных значений bytea

Десятичное значение байта	Описание	Спецпоследовательность вывода	Пример	Выводимый результат
92	обратная косая черта	\\	SELECT '<134':::bytea;	\\
от 0 до 31 и от 127 до 255	«непечатаемые» байты	\xxx (значение байта)	SELECT '\001':::bytea;	\001
от 32 до 126	«печатаемые» байты	представление из клиентского набора символов	SELECT '\176':::bytea;	~

В зависимости от применяемой клиентской библиотеки Синергия-БД, для преобразования значений bytea в спецстроки и обратно могут потребоваться дополнительные действия. Например, если приложение сохраняет в строках символы перевода строк, возможно их также нужно будет представить спецпоследовательностями.

6.5. Типы даты/времени

Синергия-БД поддерживает полный набор типов даты и времени SQL, показанный в таблице 6.9. Все даты считаются по Григорианскому календарю, даже для времени до его введения.

Таблица 6.9 – Типы даты/времени

Имя	Размер	Описание	Наименьшее значение	Наибольшее значение	Точность
timestamp [(p)] [without time zone]	8 байт	дата и время (без часового пояса)	4713 до н. э.	294276 н. э.	1 микросекунда
timestamp [(p)] with time zone	8 байт	дата и время (с часовым поясом)	4713 до н. э.	294276 н. э.	1 микросекунда
date	4 байта	дата (без времени суток)	4713 до н. э.	5874897 н. э.	1 день
time [(p)] [without time zone]	8 байт	время суток (без даты)	00:00:00	24:00:00	1 микросекунда

Окончание таблицы 6.9

Имя	Размер	Описание	Наименьшее значение	Наибольшее значение	Точность
time [(p)] with time zone	12 байт	время дня (без даты), с часовым поясом	00:00:00+1459	24:00:00-1459	1 микросекунда
interval [поля] [(p)]	16 байт	временной интервал	-178000000 лет	178000000 лет	1 микросекунда
Примечание. Стандарт SQL требует, чтобы тип timestamp подразумевал timestamp without time zone (время без часового пояса), и Синергия-БД следует этому. Для краткости timestamp with time zone можно записать как timestamptz; это расширение Синергия-БД.					

Типы time, timestamp и interval принимают необязательное значение точности p, определяющее, сколько знаков после запятой должно сохраняться в секундах. По умолчанию точность не ограничивается. Допустимые значения p лежат в интервале от 0 до 6.

Тип interval дополнительно позволяет ограничить набор сохраняемых полей следующими фразами:

YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
YEAR TO MONTH
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND

Заметьте, что если указаны и поля, и точность p, указание полей должно включать SECOND, так как точность применима только к секундам.

Тип time with time zone определен стандартом SQL, но в его определении

описаны свойства сомнительной ценности. В большинстве случаев сочетание типов date, time, timestamp without time zone и timestamp with time zone удовлетворяет все потребности в функционале дат/времени, возникающие в приложениях.

Типы abstime и reltime имеют меньшую точность и предназначены для внутреннего использования. Эти типы не рекомендуется использоваться в обычных приложениях; их может не быть в будущих версиях.

6.5.1. Ввод даты/времени

Значения даты и времени принимаются практически в любом разумном формате, включая ISO 8601, SQL-совместимый, традиционный формат POSTGRES и другие. В некоторых форматах порядок даты, месяца и года во вводимой дате неоднозначен и поэтому поддерживается явное определение формата. Для этого предназначен параметр DateStyle. Когда он имеет значение MDY, выбирается интерпретация месяц-день-год, значению DMY соответствует день-месяц-год, а YMD – год-месяц-день.

Синергия-БД обрабатывает вводимые значения даты/времени более гибко, чем того требует стандарт SQL.

Помните, что любые вводимые значения даты и времени нужно заключать в апострофы, как текстовые строки. За дополнительной информацией обратитесь к подразделу 2.1.2.7. SQL предусматривает следующий синтаксис:

тип [(p)] 'значение'

Здесь p – необязательное указание точности, определяющее число знаков после точки в секундах. Точность может быть определена для типов time, timestamp и interval в интервале от 0 до 6. Если в определении константы точность не указана, она считается равной точности значения в строке (но не больше 6 цифр).

6.5.1.1. Даты

В таблице 6.10 приведены некоторые допустимые значения типа date.

Таблица 6.10 – Вводимые даты

Пример	Описание
1999-01-08	ISO 8601; 8 января в любом режиме (рекомендуемый формат)
January 8, 1999	воспринимается однозначно в любом режиме datestyle

Окончание таблицы 6.10

Пример	Описание
1/8/1999	8 января в режиме MDY и 1 августа в режиме DMY
1/18/1999	18 января в режиме MDY; недопустимая дата в других режимах
01/02/03	2 января 2003 г. в режиме MDY; 1 февраля 2003 г. в режиме DMY и 3 февраля 2001 г. в режиме YMD
1999-Jan-08	8 января в любом режиме
Jan-08-1999	8 января в любом режиме
08-Jan-1999	8 января в любом режиме
99-Jan-08	8 января в режиме YMD; ошибка в других режимах
08-Jan-99	8 января; ошибка в режиме YMD
Jan-08-99	8 января; ошибка в режиме YMD
19990108	ISO 8601; 8 января 1999 в любом режиме
990108	ISO 8601; 8 января 1999 в любом режиме
1999.008	год и день года
J2451187	дата по юлианскому календарю
January 8, 99 BC	99 до н.э.

6.5.1.2. Время

Для хранения времени суток без даты предназначены типы time [(p)] without time zone and time [(p)] with time zone. Тип time без уточнения эквивалентен типу time without time zone.

Допустимые вводимые значения этих типов состоят из записи времени суток и необязательного указания часового пояса (см. таблицу 6.11 и таблицу 6.12). Если в значении для типа time without time zone указывается часовой пояс, он просто игнорируется. Так же будет игнорироваться дата, если её указать, за исключением случаев, когда в указанном часовом поясе принят переход на летнее время, например America/New_York. В данном случае указать дату необходимо, чтобы система могла определить, применяется ли обычное или летнее время. Соответствующее смещение часового пояса записывается в значении time with time zone.

Таблица 6.11 – Вводимое время

Пример	Описание
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	то же, что и 04:05; AM не меняет значение времени
04:05 PM	то же, что и 16:05; часы должны быть <= 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	часовой пояс задается аббревиатурой
2003-04-12 04:05:06 America/New_York	часовой пояс задается полным названием

Таблица 6.12 – Вводимый часовой пояс

Пример	Описание
PST	аббревиатура (Pacific Standard Time, Стандартное тихоокеанское время)
America/New_York	полное название часового пояса
PST8PDT	указание часового пояса в стиле POSIX
-8:00	смещение часового пояса PST по ISO-8601
-800	смещение часового пояса PST по ISO-8601
-8	смещение часового пояса PST по ISO-8601
zulu	принятое у военных сокращение UTC
z	краткая форма zulu

Подробнее узнать о том, как указывается часовой пояс, можно в подразделе 6.5.3.

6.5.1.3. Даты и время

Допустимые значения типов timestamp состоят из записи даты и времени, после которого может указываться часовой пояс и необязательное уточнение AD или BC, определяющее эпоху до нашей эры и нашу эру соответственно. AD/BC можно указать и перед часовым поясом, но предпочтительнее первый вариант. Таким образом:

1999-01-08 04:05:06

и

1999-01-08 04:05:06 -8:00

допустимые варианты, соответствующие стандарту ISO 8601. В дополнение к этому поддерживается распространенный формат:

January 8 04:05:06 1999 PST

Стандарт SQL различает константы типов timestamp without time zone и timestamp with time zone по знаку «+» или «-» и смещению часового пояса, добавленному после времени. Следовательно, согласно стандарту, записи:

TIMESTAMP '2004-10-19 10:23:54'

должен соответствовать тип timestamp without time zone, а

TIMESTAMP '2004-10-19 10:23:54+02'

тип timestamp with time zone. Синергия-БД никогда не анализирует содержимое текстовой строки, чтобы определить тип значения, и поэтому обе записи будут обработаны как значения типа timestamp without time zone. Чтобы текстовая

константа обрабатывалась как timestamp with time zone, укажите этот тип явно:

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

В константе типа timestamp without time zone Синергия-БД просто игнорирует часовой пояс. То есть результирующее значение вычисляется только из полей даты/времени и не подстраивается под указанный часовой пояс.

Значения timestamp with time zone внутри всегда хранятся в UTC (Universal Coordinated Time, Всемирное скоординированное время или время по Гринвичу, GMT). Вводимое значение, в котором явно указан часовой пояс, переводится в UTC с учетом смещения данного часового пояса. Если во входной строке не указан часовой пояс, подразумевается часовой пояс, заданный системным параметром TimeZone и время так же пересчитывается в UTC со смещением timezone.

Когда значение timestamp with time zone выводится, оно всегда преобразуется из UTC в текущий часовой пояс timezone и отображается как локальное время. Чтобы получить время для другого часового пояса, нужно либо изменить timezone, либо воспользоваться конструкцией AT TIME ZONE.

В преобразованиях между timestamp without time zone и timestamp with time zone обычно предполагается, что значение timestamp without time zone содержит местное время (для часового пояса timezone). Другой часовой пояс для преобразования можно задать с помощью AT TIME ZONE.

6.5.1.4. Специальные значения

Синергия-БД для удобства поддерживает несколько специальных значений даты/времени, перечисленных в таблице 6.13. Значения infinity и -infinity имеют особое представление в системе и они отображаются в том же виде, тогда как другие варианты при чтении преобразуются в значения даты/времени. В частности, now и подобные строки преобразуются в актуальные значения времени в момент чтения. Чтобы использовать эти значения в качестве констант в командах SQL, их нужно заключать в апострофы.

Таблица 6.13 – Специальные значения даты/времени

Вводимая строка	Допустимые типы	Описание
epoch	date, timestamp	1970-01-01 00:00:00+00 (точка отсчета времени в Unix)
infinity	date, timestamp	время после максимальной допустимой даты

Окончание таблицы 6.13

Вводимая строка	Допустимые типы	Описание
-infinity	date, timestamp	время до минимальной допустимой даты
now	date, time, timestamp	время начала текущей транзакции
today	date, timestamp	время начала текущих суток
tomorrow	date, timestamp	время начала следующих суток
yesterday	date, timestamp	время начала предыдущих суток
allballs	time	00:00:00.00 UTC

Для получения текущей даты/времени соответствующего типа можно также использовать следующие SQL-совместимые функции: CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME и LOCALTIMESTAMP. Последние четыре функции принимают необязательное указание точности. Заметьте, что во входных строках эти SQL-функции не распознаются.

6.5.2. Вывод даты/времени

В качестве выходного формата типов даты/времени можно использовать один из четырех стилей: ISO 8601, SQL (Ingres), традиционный формат POSTGRES (формат date в Unix) или German. По умолчанию выбран формат ISO. Стандарт SQL требует, чтобы использовался именно ISO 8601. Другой формат называется «SQL» исключительно по историческим причинам. Примеры всех стилей вывода перечислены в таблице 6.14. Вообще со значениями типов date и time выводилась бы только часть даты или времени из показанных примеров, но со стилем POSTGRES значение даты без времени выводится в формате ISO.

Таблица 6.14 – Стили вывода даты/время

Стиль	Описание	Пример
ISO	ISO 8601, стандарт SQL	1997-12-17 07:37:16-08
SQL	традиционный стиль	12/17/1997 07:37:16.00 PST
Postgres	изначальный стиль	Wed Dec 17 07:37:16 1997 PST
German	региональный стиль	17.12.1997 07:37:16.00 PST

Примечание. ISO 8601 указывает, что дата должна отделяться от времени буквой Т в верхнем регистре. Синергия-БД принимает этот формат при вводе, но при выводе вставляет вместо Т пробел, как показано выше. Это сделано для улучшения читаемости и для совместимости с RFC 3339 и другими СУБД.

В стилях SQL и POSTGRES день выводится перед месяцем, если установлен порядок DMY, а в противном случае месяц выводится перед днем. Как этот параметр также влияет на интерпретацию входных значений, описано в подразделе 6.5.1.

Соответствующие примеры показаны в таблице 6.15.

Таблица 6.15 – Соглашения о порядке компонентов даты

Параметр datestyle	Порядок при вводе	Пример вывода
SQL, DMY	день/месяц/год	17/12/1997 15:37:16.00 CET
SQL, MDY	месяц/день/год	12/17/1997 07:37:16.00 PST
Postgres, DMY	день/месяц/год	Wed 17 Dec 07:37:16 1997 PST

Стиль даты/времени пользователь может выбрать с помощью команды `SET datestyle`, параметра `DateStyle` в файле конфигурации `postgresql.conf` или переменной окружения `PGDATESTYLE` на сервере или клиенте.

Для большей гибкости при форматировании выводимой даты/времени можно использовать функцию `to_char`.

6.5.3. Часовые пояса

Часовые пояса и правила их применения определяются, как вы знаете, не только по географическим, но и по политическим соображениям. Часовые пояса во всем мире были более-менее стандартизированы в начале прошлого века, но они продолжают претерпевать изменения, в частности это касается перехода на летнее время. Для расчета времени в прошлом Синергия-БД получает исторические сведения о правилах часовых поясов из распространенной базы данных IANA (Olson). Для будущего времени предполагается, что в заданном часовом поясе будут продолжать действовать последние принятые правила.

Синергия-БД стремится к совместимости со стандартом SQL в наиболее типичных случаях. Однако стандарт SQL допускает некоторые странности при смешивании типов даты и времени. Две очевидные проблемы:

1. Хотя для типа `date` часовой пояс указать нельзя, это можно сделать для типа `time`. В реальности это не очень полезно, так как без даты нельзя точно определить смещение при переходе на летнее время.
2. По умолчанию часовой пояс задается постоянным смещением от UTC. Это также не позволяет учесть летнее время при арифметических операций с датами, пересекающими границы летнего времени.

Поэтому мы советуем использовать часовой пояс с типами, включающими и

время, и дату. Мы не рекомендуем использовать тип `time with time zone` (хотя Синергия-БД поддерживает его для старых приложений и совместимости со стандартом SQL). Для типов, включающих только дату или только время, в Синергия-БД предполагается местный часовой пояс.

Все значения даты и времени с часовым поясом представляются внутри в UTC, а при передаче клиентскому приложению они переводятся в местное время, при этом часовой пояс по умолчанию определяется параметром конфигурации `TimeZone`.

Синергия-БД позволяет задать часовой пояс тремя способами:

1. Полное название часового пояса, например `America/New_York`. Все допустимые названия перечислены в представлении `pg_timezone_names`. Определения часовых поясов Синергия-БД берет из широко распространенной базы IANA, так что имена часовых поясов Синергия-БД будут воспринимать и многие другие приложения.
2. Аббревиатура часового пояса, например `PST`. Такое определение просто задает смещение от UTC, в отличие от полных названий поясов, которые кроме того подразумевают и правила перехода на летнее время. Распознаваемые аббревиатуры перечислены в представлении `pg_timezone_abbrevs`. Аббревиатуры можно использовать во вводимых значениях даты/времени и в операторе `AT TIME ZONE`, но не в параметрах конфигурации `TimeZone` и `log_timezone`.
3. В дополнение к полным названиям и аббревиатурам часовых поясов Синергия-БД принимает указания часового пояса в стиле POSIX: `STD` смещение или `STD` смещение `DST`, где `STD` – аббревиатура пояса, смещение – разница с UTC, а `DST` – необязательное дополнение, обозначающее летнее время и добавляющее к смещению ещё один час. Например, если бы обозначение `EST5EDT` не являлось названием часового пояса, оно все равно было бы воспринято и функционально соответствовало бы поясу североамериканского восточного времени. При такой записи аббревиатура пояса может быть набором букв или обычной строкой, заключенной в угловые

скобки ($\langle \rangle$). Когда представлено название пояса с летним временем, при пересчете времени будут использоваться правила перехода на летнее время, описанные в базе данных IANA в файле `posixrules`. В стандартной установке Синергия-БД файл `posixrules` соответствует описанию `US/Eastern`, так что для указаний часовых поясов в стиле POSIX применяются правила перехода на летнее время, принятые в США. При необходимости это можно изменить, заменив файл `posixrules`.

Вкратце, различие между аббревиатурами и полными названиями заключается в следующем: аббревиатуры представляют определенный сдвиг от UTC, а полное название подразумевает ещё и местное правило по переходу на летнее время, то есть, возможно, два сдвига от UTC. Например, `2014-06-04 12:00 America/New_York` представляет полдень по местному времени в Нью-Йорк, что для данного дня было бы летним восточным временем (EDT или UTC-4). Так что `2014-06-04 12:00 EDT` обозначает тот же момент времени. Но `2014-06-04 12:00 EST` задает стандартное восточное время (UTC-5), не зависящее от того, действовало ли летнее время в этот день.

Мало того, в некоторых юрисдикциях одна и та же аббревиатура часового пояса означала разные сдвиги UTC в разное время; например, аббревиатура московского времени MSK несколько лет означала UTC+3, а затем стала означать UTC+4. Синергия-БД обрабатывает такие аббревиатуры в соответствии с их значениями на заданную дату, но, как и с примером выше EST, это не обязательно будет соответствовать местному гражданскому времени в этот день.

При этом следует использовать возможность указания часового пояса в стиле POSIX с осторожностью, так как при этом могут быть приняты заведомо неверные данные, потому что разумность аббревиатуры никак не проверяется. Например, команда `SET TIMEZONE TO FOOBAR0` будет работать и система примет эту довольно оригинальную аббревиатуру для UTC. Также следует учитывать, что в названиях часовых поясов POSIX положительные смещения соответствуют сдвигу к западу Гринвича. Во всех остальных формах Синергия-БД следует соглашению ISO-8601, по которому положительным смещениям соответствует сдвиг к востоку от

Гринвича.

Независимо от формы, регистр в названиях и аббревиатурах часовых поясов не важен.

Ни названия, ни аббревиатуры часовых поясов, не зашиты в самом сервере; они считываются из файлов конфигурации, находящихся в путях `.../share/timezone/` и `.../share/timezonesets/` относительно каталога установки.

Параметр конфигурации `TimeZone` можно установить в `postgresql.conf` или любым другим стандартным способом. Часовой пояс может быть также определен следующими специальными способами:

1. Часовой пояс для текущего сеанса можно установить с помощью SQL-команды `SET TIME ZONE`. Это альтернативная запись команды `SET TIMEZONE TO`, более соответствующая SQL-стандарту.
2. Если установлена переменная окружения `PGTZ`, клиенты `libpq` используют её значение, выполняя при подключении к серверу команду `SET TIME ZONE`.

6.5.4. Ввод интервалов

Значения типа `interval` могут быть записаны в следующей расширенной форме:

```
[@] количество единица [количество единица...]  
[направление]
```

где количество – это число (возможно, со знаком); единица – одно из значений: `microsecond`, `millisecond`, `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, `millennium` (которые обозначают соответственно микросекунды, миллисекунды, секунды, минуты, часы, дни, недели, месяцы, годы, десятилетия, века и тысячелетия), либо эти же слова во множественном числе, либо их сокращения; направление может принимать значение `ago` (назад) или быть пустым. Знак `@` является необязательным. Все заданные величины различных единиц суммируются вместе с учетом знака чисел. Указание `ago` меняет знак всех полей на противоположный. Этот синтаксис также используется при выводе интервала, если параметр `IntervalStyle` имеет значение `postgres_verbose`.

Количества дней, часов, минут и секунд можно определить, не указывая явно соответствующие единицы. Например, запись `'1 12:59:10'` равнозначна `'1 day 12 hours`

59 min 10 sec'. Сочетание года и месяца также можно записать через минус; например '200-10' означает то, же что и '200 years 10 months'. На самом деле только эти краткие формы разрешены стандартом SQL и они используются при выводе, когда IntervalStyle имеет значение sql_standard.

Интервалы можно также записывать в виде, определенном в ISO 8601, либо в «формате с кодами», описанном в разделе 4.4.3.2 этого стандарта, либо в «альтернативном формате», описанном в разделе 4.4.3.3. Формат с кодами выглядит так:

`P количество единица [количество единица ...] [T
[количество единица ...]]`

Строка должна начинаться с символа P и может включать также T перед временем суток. Допустимые коды единиц перечислены в таблице 6.16. Коды единиц можно опустить или указать в любом порядке, но компоненты времени суток должны идти после символа T. В частности, значение кода M зависит от того, располагается ли он до или после T.

Таблица 6.16 – Коды единиц временных интервалов ISO 8601

Код	Значение
Y	годы
M	месяцы (в дате)
W	недели
D	дни
H	часы
M	минуты (во времени)
S	секунды

В альтернативном формате:

`P [ год-месяц-день ] [ T часы:минуты:секунды ]`

строка должна начинаться с P, а T разделяет компоненты даты и времени.

Значения выражаются числами так же, как и в датах ISO 8601.

При записи интервальной константы с указанием полей или присвоении колонке типа interval строки с полями, интерпретация непомеченных величин зависит от полей. Например, INTERVAL '1' YEAR воспринимается как 1 год, а INTERVAL '1' – как 1 секунда. Кроме того, значения «справа» от меньшего значащего поля, заданного в определении полей, просто отбрасываются. Например, в записи INTERVAL '1 day 2:03:04' HOUR TO MINUTE будут отброшены

секунды, но не день.

Согласно стандарту SQL, все компоненты значения interval должны быть одного знака, и ведущий минус применяется ко всем компонентам; например, минус в записи '-1 2:03:04' применяется и к дню, и к часам/минутам/секундам. Синергия-БД позволяет задавать для разных компонентов разные знаки и традиционно обрабатывает знак каждого компонента в текстовом представлении отдельно от других, так что в данном случае часы/минуты/секунды будут считаться положительными. Если параметр IntervalStyle имеет значение sql_standard, ведущий знак применяется ко всем компонентам (но только если они не содержат знаки явно). В противном случае действуют традиционные правила Синергия-БД. Во избежание неоднозначности рекомендуется добавлять знак к каждому компоненту с отрицательным значением.

В расширенном формате ввода и в некоторых полях более компактных форматов значения компонентов могут иметь дробные части, например '1.5 week' или '01:02:03.45'. Такое значение при сохранении пересчитывается в соответствующее число месяцев, дней и секунд. Когда при этом остаётся дробная часть в месяцах или в днях, она переносится в младший компонент с допущением, что 1 месяц = 30 дней, а 1 день = 24 часа. Например, значение '1.5 month' будет преобразовано в 1 месяц и 15 дней. В виде дробного числа хранятся и выводятся только секунды.

В таблице 6.17 показано несколько примеров допустимых вводимых значений типа interval.

Таблица 6.17 – Ввод интервалов

Пример	Описание
1-2	Стандартный формат SQL: 1 год и 2 месяца
3 4:05:06	Стандартный формат SQL: 3 дня 4 часа 5 минут 6 секунд
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	Традиционный формат Postgres: 1 год 2 месяца 3 дня 4 часа 5 минут 6 секунд
P1Y2M3DT4H5M6S	«Формат с кодами» ISO 8601: то же значение, что и выше
P0001-02-03T04:05:06	«Альтернативный формат» ISO 8601: то же значение, что и выше

Тип interval представлен внутри в виде отдельных значений месяцев, дней и секунд. Это объясняется тем, что число дней в месяце может быть разным, а в сутках может быть и 23, и 25 часов в дни перехода на летнее/зимнее время. Значения месяцев и дней представлены целыми числами, а число секунд может быть

дробным. Так как интервалы обычно получаются из строковых констант или при вычитании типов `timestamp`, этот способ хранения эффективен в большинстве случаев, но может давать неожиданные результаты:

```
SELECT EXTRACT(hours from '80 minutes'::interval);
date_part
-----
1
```

```
SELECT EXTRACT(days from '80 hours'::interval);
date_part
-----
0
```

Для корректировки числа дней и часов, когда они выходят за обычные границы, есть специальные функции `justify_days` и `justify_hours`.

6.5.5. Вывод интервалов

Формат вывода типа `interval` может определяться одним из четырех стилей: `sql_standard`, `postgres`, `postgres_verbose` и `iso_8601`. Выбрать нужный стиль позволяет команда `SET intervalstyle` (по умолчанию выбран `postgres`). Примеры форматов разных стилей показаны в таблице 6.18.

Стиль `sql_standard` выдает результат, соответствующий стандарту SQL, если значение интервала удовлетворяет ограничениям стандарта (и содержит либо только год и месяц, либо только день и время, и при этом все его компоненты одного знака). В противном случае выводится год-месяц, за которым идет дата-время, а в компоненты для однозначности явно добавляются знаки.

Таблица 6.18 – Примеры стилей вывода интервалов

Стиль	Интервал год-месяц	Интервал день- время	Смешанный интервал
<code>sql_standard</code>	1-2	3 4:05:06	-1-2 +3 -4:05:06
<code>postgres</code>	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
<code>postgres_verbose</code>	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
<code>iso_8601</code>	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

6.6. Логический тип

В Синергия-БД есть стандартный SQL-тип `boolean`; см. таблицу 6.19. Тип `boolean` может иметь следующие состояния: «true», «false» и третье состояние, «unknown», которое представляется SQL-значением `NULL`.

Таблица 6.19 – Логический тип данных

Имя	Размер	Описание
<code>boolean</code>	1 байт	состояние: истина или ложь

Логические константы могут представляться в SQL-запросах следующими ключевыми словами SQL: `TRUE`, `FALSE` и `NULL`.

Функция ввода данных типа `boolean` воспринимает следующие строковые представления состояния «true»:

```
true
yes
on
1
```

и следующие представления состояния «false»:

```
false
no
off
0
```

Также воспринимаются уникальные префиксы этих строк, например `t` или `n`. Регистр символов не имеет значения, а пробельные символы в начале и в конце строки игнорируются.

Функция вывода данных типа `boolean` всегда выдаёт `t` или `f`.

Пример использования типа `boolean`:

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
a |      b
---+-----
```

```
t | sic est
f | non est
```

```
SELECT * FROM test1 WHERE a;
```

```
a |      b
---+-----
t | sic est
```

Ключевые слова TRUE и FALSE являются предпочтительными (соответствующими стандарту SQL) для записи логических констант в SQL-запросах. Но вы также можете использовать строковые представления, которые допускает синтаксис строковых констант, например, 'yes'::boolean.

Заметьте, что при анализе запроса TRUE и FALSE автоматически считаются значениями типа boolean, но для NULL это не так, потому что ему может соответствовать любой тип. Поэтому в некоторых контекстах может потребоваться привести NULL к типу boolean явно, например так: NULL::boolean. С другой стороны, приведение строковой константы к логическому типу можно опустить в тех контекстах, где анализатор запроса может понять, что буквальное значение должно иметь тип boolean.

6.7. Типы перечислений

Типы перечислений (enum) определяют статический упорядоченный набор значений, так же как и типы enum, существующие в ряде языков программирования. В качестве перечисления можно привести дни недели или набор состояний.

6.7.1. Объявление перечислений

Тип перечислений создаются с помощью команды CREATE TYPE, например так:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Созданные типы enum можно использовать в определениях таблиц и функций, как и любые другие:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```



```
CREATE TABLE person (  
    name text,  
    current_mood mood  
);  
INSERT INTO person VALUES ('Moe', 'happy');  
SELECT * FROM person WHERE current_mood = 'happy';  
    name | current_mood  
-----+-----  
    Moe  | happy  
(1 row)
```

6.7.2. Порядок

Порядок значений в перечислении определяется последовательностью, в которой были указаны значения при создании типа. Перечисления поддерживаются всеми стандартными операторами сравнения и связанными агрегатными функциями, например:

```
INSERT INTO person VALUES ('Larry', 'sad');  
INSERT INTO person VALUES ('Curly', 'ok');  
SELECT * FROM person WHERE current_mood > 'sad';  
    name | current_mood  
-----+-----  
    Moe  | happy  
    Curly | ok  
(2 rows)
```

```
SELECT * FROM person WHERE current_mood > 'sad' ORDER BY  
current_mood;  
    name | current_mood  
-----+-----  
    Curly | ok  
    Moe  | happy  
(2 rows)
```

```
SELECT name
```

```
FROM person
WHERE current_mood = (SELECT MIN(current_mood) FROM
person);
    name
-----
    Larry
(1 row)
```

6.7.3. Безопасность типа

Все типы перечислений считаются уникальными и поэтому значения разных типов нельзя сравнивать. Взгляните на этот пример:

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy',
'ecstatic');
CREATE TABLE holidays (
    num_weeks integer,
    happiness happiness
);
INSERT INTO holidays(num_weeks,happiness) VALUES (4,
'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6,
'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8,
'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2,
'sad');
ОШИБКА: неверное значение для перечисления happiness:
«sad»
SELECT person.name, holidays.num_weeks FROM person,
holidays
```

```
    WHERE person.current_mood = holidays.happiness;
```

ОШИБКА: оператор не существует: mood = happiness

Если вам действительно нужно сделать что-то подобное, вы можете либо реализовать собственный оператор, либо явно преобразовать типы в запросе:

```
SELECT person.name, holidays.num_weeks FROM person,
holidays
    WHERE person.current_mood::text =
```

```
holidays.happiness::text;
  name | num_weeks
-----+-----
  Moe  |           4
(1 row)
```

6.7.4. Тонкости реализации

В метках значений регистр имеет значение, т. е. 'happy' и 'HAPPY' – не одно и то же. Также в метках имеют значение пробелы.

Хотя типы-перечисления предназначены прежде всего для статических наборов значений, имеется возможность добавлять новые значения в существующий тип-перечисление и переименовывать значения. Однако удалять существующие значения из перечисления, а также изменять их порядок, нельзя – для получения нужного результата придётся удалить и воссоздать это перечисление.

Значение enum занимает на диске 4 байта. Длина текстовой метки значения ограничена параметром компиляции NAMEDATALEN; в стандартных сборках Синергия-БД он ограничивает длину 63 байтами.

В метках значений регистр имеет значение, т.е. 'happy' и 'HAPPY' – не одно и то же. Также в метках имеют значение пробелы.

Сопоставления внутренних значений enum с текстовыми метками хранятся в системном каталоге pg_enum. Он может быть полезен в ряде случаев.

6.8. Геометрические типы

Геометрические типы данных представляют объекты в двумерном пространстве. Все существующие в Синергия-БД геометрические типы перечислены в таблице 6.20.

Таблица 6.20 – Геометрические типы

Имя	Размер	Описание	Представление
point	16 байт	Точка на плоскости	(x,y)
line	32 байта	Бесконечная прямая	{A,B,C}
lseg	32 байта	Отрезок	((x1,y1),(x2,y2))
box	32 байта	Прямоугольник	((x1,y1),(x2,y2))
path	16+16n байт	Закрытый путь (подобный многоугольнику)	((x1,y1),...)
path	16+16n байт	Открытый путь	[(x1,y1),...]

Окончание таблицы 6.20

Имя	Размер	Описание	Представление
polygon	40+16n байт	Многоугольник (подобный закрытому пути)	((x1,y1),...)
circle	24 байта	Окружность	<(x,y),r> (центр окружности и радиус)

Для выполнения различных геометрических операций, в частности масштабирования, вращения и определения пересечений, Синергия-БД предлагает богатый набор функций и операторов.

6.8.1. Точки

Точки – это основной элемент, на базе которого строятся все остальные геометрические типы. Значения типа point записываются в одном из двух форматов:

(x , y)

x , y

где x и y – координаты точки на плоскости, выраженные числами с плавающей точкой.

Выводятся точки в первом формате.

6.8.2. Прямые

Прямые представляются линейным уравнением $Ax + By + C = 0$, где A и B не равны 0. Значения типа line вводятся и выводятся в следующем виде:

{ A, B, C }

Кроме того, для ввода может использоваться любая из этих форм:

[(x1 , y1) , (x2 , y2)]

((x1 , y1) , (x2 , y2))

(x1 , y1) , (x2 , y2)

x1 , y1 , x2 , y2

где (x1,y1) и (x2,y2) – две различные точки на данной прямой.

6.8.3. Отрезки

Отрезок представляется парой точек, определяющих концы отрезка. Значения типа lseg записываются в одной из следующих форм:

[(x1 , y1) , (x2 , y2)]

((x1 , y1) , (x2 , y2))

$$\begin{aligned} & (x_1, y_1), (x_2, y_2) \\ & x_1, y_1, x_2, y_2 \end{aligned}$$

где (x_1, y_1) и (x_2, y_2) – концы отрезка.

Выводятся отрезки в первом формате.

6.8.4. Прямоугольники

Прямоугольник представляется двумя точками, находящимися в противоположных его углах. Значения типа `box` записываются в одной из следующих форм:

$$\begin{aligned} & ((x_1, y_1), (x_2, y_2)) \\ & (x_1, y_1), (x_2, y_2) \\ & x_1, y_1, x_2, y_2 \end{aligned}$$

где (x_1, y_1) и (x_2, y_2) – противоположные углы прямоугольника.

Выводятся прямоугольники во второй форме.

Во вводимом значении могут быть указаны любые два противоположных угла, но затем они будут упорядочены, так что внутри сохраняются правый верхний и левый нижний углы, в таком порядке.

6.8.5. Пути

Пути представляют собой списки соединенных точек. Пути могут быть закрытыми, когда подразумевается, что первая и последняя точка в списке соединены, или открытыми, в противном случае.

Значения типа `path` записываются в одной из следующих форм:

$$\begin{aligned} & [(x_1, y_1), \dots, (x_n, y_n)] \\ & ((x_1, y_1), \dots, (x_n, y_n)) \\ & (x_1, y_1), \dots, (x_n, y_n) \\ & (x_1, y_1, \dots, x_n, y_n) \\ & x_1, y_1, \dots, x_n, y_n \end{aligned}$$

где точки задают узлы сегментов, составляющих путь. Квадратные скобки `[]` указывают, что путь открытый, а круглые `()` – закрытый. Когда внешние скобки опускаются, как в показанных выше последних трех формах, считается, что путь закрытый.

Пути выводятся в первой или второй форме, в соответствии с типом.

6.8.6. Многоугольники

Многоугольники представляются списками точек (вершин). Многоугольники похожи на закрытые пути, но хранятся в другом виде и для работы с ними предназначен отдельный набор функций.

Значения типа `polygon` записываются в одной из следующих форм:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1 , ... , xn , yn )
x1 , y1 , ... , xn , yn
```

где точки задают узлы сегментов, образующих границу многоугольника.

Выводятся многоугольники в первом формате.

6.8.7. Окружности

Окружности задаются координатами центра и радиусом. Значения типа `circle` записываются в одном из следующих форматов:

```
< ( x , y ) , r >
( ( x , y ) , r )
( x , y ) , r
x , y , r
```

где (x,y) – центр окружности, а r – её радиус.

Выводятся окружности в первом формате.

6.9. Типы, описывающие сетевые адреса

Синергия-БД предлагает типы данных для хранения адресов IPv4, IPv6 и MAC, показанные в таблице 6.21. Для хранения сетевых адресов лучше использовать эти типы, а не простые текстовые строки, так как Синергия-БД проверяет вводимые значения данных типов и предоставляет специализированные операторы и функции для работы с ними.

Таблица 6.21 – Типы, описывающие сетевые адреса

Имя	Размер	Описание
<code>cidr</code>	7 или 19 байт	Сети IPv4 и IPv6
<code>inet</code>	7 или 19 байт	Узлы и сети IPv4 и IPv6
<code>macaddr</code>	6 байт	MAC-адреса
<code>macaddr8</code>	8 байт	MAC-адреса (в формате EUI-64)

При сортировке типов `inet` и `cidr`, адреса IPv4 всегда идут до адресов IPv6, в том числе адреса IPv4, включенные в IPv6 или сопоставленные с ними, например `::10.2.3.4` или `::ffff:10.4.3.2`.

6.9.1. `inet`

Тип `inet` содержит IPv4- или IPv6-адрес узла и может также содержать его подсеть, все в одном поле. Подсеть представляется числом бит, определяющих адрес сети в адресе узла (или «маску сети»). Если маска сети равна 32 для адреса IPv4, такое значение представляет не подсеть, а определенный узел. Адреса IPv6 имеют длину 128 бит, поэтому уникальный адрес узла задается с маской 128 бит. Заметьте, что когда нужно, чтобы принимались только адреса сетей, следует использовать тип `cidr`, а не `inet`.

Вводимые значения такого типа должны иметь формат IP-адрес/у, где IP-адрес – адрес IPv4 или IPv6, а у – число бит в маске сети. Если компонент /у отсутствует, маска сети считается равной 32 для IPv4 и 128 для IPv6, так что это значение будет представлять один узел. При выводе компонент /у опускается, если сетевой адрес определяет адрес одного узла.

6.9.2. `cidr`

Тип `cidr` содержит определение сети IPv4 или IPv6. Входные и выходные форматы соответствуют соглашениям CIDR (Classless Internet Domain Routing, Бесклассовая межсетевая адресация). Определение сети записывается в формате IP-адрес/у, где IP-адрес – адрес сети IPv4 или IPv6, а у – число бит в маске сети. Если у не указывается, это значение вычисляется по старой классовой схеме нумерации сетей, но при этом оно может быть увеличено, чтобы в него вошли все байты введенного адреса. Если в сетевом адресе справа от маски сети окажутся биты со значением 1, он будет считаться ошибочным.

В таблице 6.22 показаны несколько примеров адресов.

Таблица 6.22 – Примеры допустимых значений типа `cidr`

Вводимое значение <code>cidr</code>	Выводимое значение <code>cidr</code>	<code>abbrev(cidr)</code>
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

6.9.3. Различия `inet` и `cidr`

Существенным различием типов данных `inet` и `cidr` является то, что `inet` принимает значения с ненулевыми битами справа от маски сети, а `cidr` – нет. Например, значение 192.168.0.1/24 является допустимым для типа `inet`, но не для `cidr`.

Примечание. Если вас не устраивает выходной формат значений `inet` или `cidr`, попробуйте функции `host`, `text` и `abbrev`.

6.9.4. `macaddr`

Тип `macaddr` предназначен для хранения MAC-адреса, примером которого является адрес сетевой платы Ethernet (хотя MAC-адреса применяются и для других целей). Вводимые значения могут задаваться в следующих форматах:

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'0800-2b01-0203'
'08002b010203'
```

Все эти примеры определяют один и тот же адрес. Шестнадцатеричные цифры от `a` до `f` могут быть и в нижнем, и в верхнем регистре. Выводятся MAC-адреса

всегда в первой форме.

Стандарт IEEE 802-2001 считает канонической формой MAC-адресов вторую (с минусами), а в первой (с двоеточиями) предполагает обратный порядок бит, так что 08-00-2b-01-02-03 = 01:00:4D:08:04:0C. В настоящее время этому соглашению практически никто не следует, и уместно оно было только для устаревших сетевых протоколов (таких как Token Ring). Синергия-БД не меняет порядок бит и во всех принимаемых форматах подразумевается традиционный порядок LSB.

Последние пять входных форматов не описаны ни в каком стандарте.

6.9.5. macaddr8

Тип `macaddr8` хранит MAC-адреса в формате EUI-64, применяющиеся, например, для аппаратных адресов плат Ethernet (хотя MAC-адреса используются и для других целей). Этот тип может принять и 6-байтовые, и 8-байтовые адреса MAC и сохраняет их в 8 байтах. MAC-адреса, заданные в 6-байтовом формате, хранятся в формате 8 байт, а 4-ый и 5-ый байт содержат FF и FE, соответственно. Заметьте, что для IPv6 используется модифицированный формат EUI-64, в котором 7-ой бит должен быть установлен в 1 после преобразования из EUI-48. Для выполнения этого изменения предоставляется функция `macaddr8_set7bit`. Вообще говоря, этот тип принимает любые строки, состоящие из пар шестнадцатеричных цифр (выровненных по границам байт), которые могут согласованно разделяться одинаковыми символами ':', '-' или '.'. Шестнадцатеричных цифр должно быть либо 16 (8 байт), либо 12 (6 байт). Ведущие и замыкающие пробелы игнорируются. Ниже показаны примеры допустимых входных строк:

```
'08:00:2b:01:02:03:04:05'
'08-00-2b-01-02-03-04-05'
'08002b:0102030405'
'08002b-0102030405'
'0800.2b01.0203.0405'
'0800-2b01-0203-0405'
'08002b01:02030405'
```

```
'08002b0102030405'
```

Во всех этих примерах задаётся один и тот же адрес. Для цифр с a по f принимаются буквы и в верхнем, и в нижнем регистре. Вывод всегда представляется в первом из показанных форматов. Последние шесть входных форматов, показанных выше, не являются стандартизированными. Чтобы преобразовать традиционный 48-битный MAC-адрес в формате EUI-48 в модифицированный формат EUI-64 для включения в состав адреса IPv6 в качестве адреса узла, используйте функцию `macaddr8_set7bit`:

```
SELECT macaddr8_set7bit('08:00:2b:01:02:03');
```

```

macaddr8_set7bit
-----
0a:00:2b:ff:fe:01:02:03
(1 row)
```

6.10. Битовые строки

Битовые строки представляют собой последовательности из 1 и 0. Их можно использовать для хранения или отображения битовых масок. В SQL есть два битовых типа: `bit(n)` и `bit varying(n)`, где `n` – положительное целое число.

Длина значения типа `bit` должна в точности равняться `n`; при попытке сохранить данные длиннее или короче произойдет ошибка. Данные типа `bit varying` могут иметь переменную длину, но не превышающую `n`; строки большей длины не будут приняты. Запись `bit` без указания длины равнозначна записи `bit(1)`, тогда как `bit varying` без указания длины подразумевает строку неограниченной длины.

Примечание. При попытке привести значение битовой строки к типу `bit(n)`, оно будет усечено или дополнено нулями справа до длины ровно `n` бит, ошибки при этом не будет. Подобным образом, если явно привести значение битовой строки к типу `bit varying(n)`, она будет усечена справа, если её длина превышает `n` бит.

Синтаксис констант битовых строк описан в подразделе 2.1.2.5.

Пример использования битовых строк:

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
```

```
INSERT INTO test VALUES (B'10', B'101');
```

ОШИБКА: длина битовой строки (2) не соответствует типу bit(3)

```
INSERT INTO test VALUES (B'10'::bit(3), B'101');
```

```
SELECT * FROM test;
```

```

a   |   b
-----+-----
101 | 00
100 | 101
```

Для хранения битовой строки используется по 1 байту для каждой группы из 8 бит, плюс 5 или 8 байт дополнительно в зависимости от длины строки (но длинные строки могут быть сжаты или вынесены отдельно, как описано в разделе 6.3 применительно к символьным строкам).

6.11. Типы, предназначенные для текстового поиска

Синергия-БД предоставляет два типа данных для поддержки полнотекстового поиска. Текстовым поиском называется операция анализа набора документов с текстом на естественном языке, в результате которой находятся фрагменты, наиболее соответствующие запросу. Тип `tsvector` представляет документ в виде, оптимизированном для текстового поиска, а `tsquery` представляет запрос текстового поиска в подобном виде.

6.11.1. `tsvector`

Значение типа `tsvector` содержит отсортированный список неповторяющихся лексем, т.е. слов, нормализованных так, что все словоформы сводятся к одной. Сортировка и исключение повторяющихся слов производится автоматически при вводе значения, как показано в этом примере:

```
SELECT 'a fat cat sat on a mat and ate a fat
rat'::tsvector;
```

```

tsvector
-----
'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

Для представления в виде лексем пробелов или знаков препинания их нужно

заклучить в апострофы:

```
SELECT $$the lexeme '      ' contains spaces$$::tsvector;
tsvector
```

```
-----
'      ' 'contains' 'lexeme' 'spaces' 'the'
```

В данном и следующих примерах мы используем строку в долларах, чтобы не дублировать все апострофы в таких строках. При этом включаемый апостроф или обратную косую черту нужно продублировать:

```
SELECT $$the lexeme 'Joe''s' contains a
quote$$::tsvector;
tsvector
```

```
-----
'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

Также для лексем можно указать их целочисленные позиции:

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9
a:10 fat:11
rat:12'::tsvector;
tsvector
```

```
-----
-----
'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7
'on':5 'rat':12
'sat':4
```

Позиция обычно указывает положение исходного слова в документе. Информация о расположении слов затем может использоваться для оценки близости. Позиция может задаваться числом от 1 до 16383; большие значения просто заменяются на 16383. Если для одной лексемы дважды указывается одно положение, такое повторение отбрасывается.

Лексемам, для которых заданы позиции, также можно назначить вес, выраженный буквами A, B, C или D. Вес D подразумевается по умолчанию и поэтому он не показывается при выводе:

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
tsvector
```

```
-----  
'a':1A 'cat':5 'fat':2B,4C
```

Веса обычно применяются для отражения структуры документа, например для придания особого значения словам в заголовке по сравнению со словами в обычном тексте. Назначенным весам можно сопоставить числовые приоритеты в функциях ранжирования результатов.

Важно понимать, что тип `tsvector` сам по себе не выполняет нормализацию; предполагается, что в сохраняемом значении слова уже нормализованы приложением, например:

```
select 'The Fat Rats'::tsvector;  
      tsvector
```

```
-----  
'Fat' 'Rats' 'The'
```

Для большинства англоязычных приложений приведенные выше слова будут считаться ненормализованными, но для `tsvector` это не важно. Поэтому исходный документ обычно следует обработать функцией `to_tsvector`, нормализующей слова:

```
SELECT to_tsvector('english', 'The Fat Rats');  
      to_tsvector  
-----  
'fat':2 'rat':3
```

6.11.2. `tsquery`

Значение `tsquery` содержит искомые лексемы, объединяемые логическими операторами `&` (И), `|` (ИЛИ) и `!` (НЕ), а также оператором поиска фраз `<->` (ПРЕДШЕСТВУЕТ). Также допускается вариация оператора ПРЕДШЕСТВУЕТ вида `<N>`, где `N` – целочисленная константа, задающая расстояние между двумя искомыми лексемами. Запись оператора `<->` равнозначна `<1>`.

Для группировки операторов могут использоваться скобки. Без скобок эти операторы имеют разные приоритеты, в порядке убывания: `!` (НЕ), `<->` (ПРЕДШЕСТВУЕТ), `&` (И) и `|` (ИЛИ).

Несколько примеров:

```
SELECT 'fat & rat'::tsquery;  
      tsquery
```

```
-----  
'fat' & 'rat'
```

```
SELECT 'fat & (rat | cat)'::tsquery;  
      tsquery
```

```
-----  
'fat' & ( 'rat' | 'cat' )
```

```
SELECT 'fat & rat & ! cat'::tsquery;  
      tsquery
```

```
-----  
'fat' & 'rat' & !'cat'
```

Лексемам в `tsquery` можно дополнительно сопоставить буквы весов, при этом они будут соответствовать только тем лексемам в `tsvector`, которые имеют те же веса:

```
SELECT 'fat:ab & cat'::tsquery;  
      tsquery
```

```
-----  
'fat':AB & 'cat'
```

Кроме того, в лексемах `tsquery` можно использовать знак `*` для поиска по префиксу:

```
SELECT 'super:*'::tsquery;  
      tsquery
```

```
-----  
'super':*
```

Этот запрос найдет все слова в `tsvector`, начинающиеся с приставки «super».

Апострофы в лексемах этого типа можно использовать так же, как и в лексемах в `tsvector`; и так же, как и для типа `tsvector`, необходимая нормализация слова должна выполняться до приведения значения к типу `tsquery`. Для такой

нормализации удобно использовать функцию `to_tsquery`:

```
SELECT to_tsquery('Fat:ab & Cats');
      to_tsquery
-----
'fat':AB & 'cat'
```

Заметьте, что функция `to_tsquery` будет обрабатывать префиксы подобно другим словам, поэтому следующее сравнение возвращает true:

```
SELECT to_tsvector( 'postgraduate' ) @@
to_tsquery( 'postgres:*' );
?column?
-----
t
```

так как `postgres` преобразуется стеммером в `postgr`:

```
SELECT to_tsvector( 'postgraduate' ),
to_tsquery( 'postgres:*' );
      to_tsvector | to_tsquery
-----+-----
'postgradu':1 | 'postgr':*
```

и эта приставка находится в преобразованной форме слова `postgraduate`.

6.12. Тип UUID

Тип данных `uuid` сохраняет универсальные уникальные идентификаторы (Universally Unique Identifiers, UUID), определенные в RFC 4122, ISO/IEC 9834-8:2005 и связанных стандартах. В некоторых системах это называется GUID, глобальным уникальным идентификатором. Этот идентификатор представляет собой 128-битное значение, генерируемое специальным алгоритмом, практически гарантирующим, что этим же алгоритмом оно не будет получено больше нигде в мире. Таким образом, эти идентификаторы будут уникальными и в распределенных системах, а не только в единственной базе данных, как значения генераторов последовательностей.

UUID записывается в виде последовательности шестнадцатеричных цифр в нижнем регистре, разделенных знаками минуса на несколько групп, в таком порядке:

группа из 8 цифр, за ней три группы из 4 цифр и, наконец, группа из 12 цифр, что в сумме составляет 32 цифры и представляет 128 бит. Пример UUID в этом стандартном виде:

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

Синергия-БД также принимает альтернативные варианты: цифры в верхнем регистре, стандартную запись, заключенную в фигурные скобки, запись без минусов или с минусами, разделяющими любые группы из четырех цифр, например:

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
```

```
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
```

```
a0eebc999c0b4ef8bb6d6bb9bd380a11
```

```
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
```

```
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

Выводится значение этого типа всегда в стандартном виде.

В Синергия-БД встроены функции хранения и сравнения идентификаторов UUID, но нет внутренней функции генерирования UUID, потому что не существует какого-то единственного алгоритма, подходящего для всех приложений. Сгенерировать UUID можно с помощью дополнительного модуля `uuid-oss`, в котором реализованы несколько стандартных алгоритмов, а можно воспользоваться модулем `pgcrypto`, где тоже есть функция генерирования случайных UUID. Кроме того, можно сделать это в клиентском приложении или в другой библиотеке, подключенной на стороне сервера.

6.13. Тип XML

Тип `xml` предназначен для хранения XML-данных. Его преимущество по сравнению с обычным типом `text` в том, что он проверяет вводимые значения на допустимость по правилам XML и для работы с ним есть типобезопасные функции. Для использования этого типа дистрибутив должен быть скомпилирован в конфигурации `configure --with-libxml`.

Тип `xml` может сохранять правильно оформленные «документы», в соответствии со стандартом XML, а также фрагменты «содержимого», определяемые как менее ограниченные «узлы документа» в модели данных XQuery

и XPath. Другими словами, это означает, что во фрагментах содержимого может быть несколько элементов верхнего уровня или текстовых узлов. Является ли некоторое значение типа xml полным документом или фрагментом содержимого, позволяет определить выражение xml-значение IS DOCUMENT.

6.13.1. Создание XML-значений

Чтобы получить значение типа xml из текстовой строки, используйте функцию `xmlparse`:

```
XMLPARSE ( { DOCUMENT | CONTENT } value)
```

Примеры:

```
XMLPARSE (DOCUMENT '<?xml
version=«1.0»?><book><title>Manual</title>
<chapter>...</chapter></book>')
```

```
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

Хотя в стандарте SQL описан только один способ преобразования текстовых строк в XML-значения, специфический синтаксис Синергия-БД:

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

тоже допустим.

Тип xml не проверяет вводимые значения по схеме DTD (Document Type Declaration, Объявления типа документа), даже если в них присутствуют ссылка на DTD. В настоящее время в Синергия-БД также нет встроенной поддержки других разновидностей схем, например XML Schema.

Обратная операция, получение текстовой строки из xml, выполняется с помощью функции `xmlserialize`:

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } значение AS тип )
```

Здесь допустимый тип – `character`, `character varying` или `text` (или их псевдонимы). И в данном случае стандарт SQL предусматривает только один способ преобразования xml в тип текстовых строк, но Синергия-БД позволяет просто привести значение к нужному типу.

При преобразовании текстовой строки в тип xml или наоборот без использования функций `XMLPARSE` и `XMLSERIALIZE`, выбор режима `DOCUMENT`

или CONTENT определяется параметром конфигурации сеанса «XML option», установить который можно следующей стандартной командой:

```
SET XML OPTION { DOCUMENT | CONTENT };
```

или такой командой в духе Синергия-БД:

```
SET xmloption TO { DOCUMENT | CONTENT };
```

По умолчанию этот параметр имеет значение CONTENT, так что допускаются все формы XML-данных.

6.13.2. Обработка кодировки

Если на стороне сервера и клиента и в XML-данных используются разные кодировки символов, с этим могут возникать проблемы. Когда запросы передаются на сервер, а их результаты возвращаются клиенту в обычном текстовом режиме, Синергия-БД преобразует все передаваемые текстовые данные в кодировку для соответствующей стороны. В том числе это происходит и со строковыми представлениями XML-данных, подобными тем, что показаны в предыдущих примерах. Обычно это означает, что объявления кодировки, содержащиеся в XML-данных, могут не соответствовать действительности, когда текстовая строка преобразуется из одной кодировки в другую при передаче данных между клиентом и сервером, так как подобные включенные в данные объявления не будут изменены автоматически. Для решения этой проблемы объявления кодировки, содержащиеся в текстовых строках, вводимых в тип xml, просто игнорируются и предполагается, что XML-содержимое представлено в текущей кодировке сервера. Как следствие, для правильной обработки таких строк с XML-данными клиент должен передавать их в своей текущей кодировке. Для сервера не важно, будет ли клиент для этого преобразовывать документы в свою кодировку, или изменит ее, прежде чем передавать ему данные. При выводе значения типа xml не содержат объявления кодировки, а клиент должен предполагать, что все данные поступают в его текущей кодировке.

Если параметры запроса передаются на сервер и он возвращает результаты клиенту в двоичном режиме, кодировка символов не преобразуется, так что возникает другая ситуация. В этом случае объявление кодировки в XML

принимается во внимание, а если его нет, то предполагается, что данные закодированы в UTF-8 (это соответствует стандарту XML; заметьте, что Синергия-БД не поддерживает UTF-16). При выводе в данные будет добавлено объявление кодировки, выбранной на стороне клиента (но если это UTF-8, объявление будет опущено).

Само собой, XML-данные в Синергия-БД будут обрабатываться гораздо эффективнее, когда и в XML-данных, и на стороне клиента, и на стороне сервера используется одна кодировка. Так как внутри XML-данные представляются в UTF-8, оптимальный вариант, когда на сервере также выбрана кодировка UTF-8.

Примечание. Некоторые XML-функции способны работать исключительно с ASCII-данными, если кодировка сервера не UTF-8. В частности, это известная особенность функций `xmltable()` и `xpath()`.

6.13.3. Обращение к XML-значениям

Тип `xml` отличается от других тем, что для него не определены никакие операторы сравнения, так как четко определенного и универсального алгоритма сравнения XML-данных не существует. Одно из следствий этого — нельзя отфильтровать строки таблицы, сравнив колонку `xml` с искомым значением. Поэтому обычно XML-значения должны дополняться отдельным ключевым полем, например `ID`. Можно также сравнивать XML-значения, преобразовав их сначала в текстовые строки, но заметьте, что с учетом специфики XML-данных этот метод практически бесполезен.

Из-за отсутствия операторов сравнения для типа `xml`, для колонки этого типа также нельзя создать индекс. Поэтому когда требуется быстрый поиск в XML-данных, обойти это ограничение можно, приведя данные к типу текстовой строки и проиндексировав эти строки, либо проиндексировав выражение `XPath`. Конечно сам запрос при этом следует изменить, чтобы поиск выполнялся по индексированному выражению.

Для ускорения поиска в XML-данных также можно использовать функции полнотекстового поиска в Синергия-БД. Однако это требует определенной подготовки данных, что дистрибутив Синергия-БД пока не поддерживает.

6.14. Типы JSON

Типы JSON предназначены для хранения данных JSON (JavaScript Object Notation, Запись объекта JavaScript) согласно стандарту RFC 7159. Такие данные можно хранить и в типе text, но типы JSON лучше тем, что проверяют, соответствует ли вводимое значение формату JSON. Для работы с ним есть также несколько специальных функций и операторов.

Существуют два типа данных JSON: json и jsonb. Они принимают на вход почти одинаковые наборы значений, но основное их отличие в эффективности. Тип json сохраняет точную копию введенного текста, которую функции обработки должны разбирать заново при каждом выполнении, тогда как данные jsonb сохраняются в разобранном двоичном формате, что несколько замедляет ввод из-за преобразования, но значительно ускоряет обработку, не требуя многократного разбора текста. Кроме того, jsonb поддерживает индексацию, что тоже может быть очень полезно.

Так как тип json сохраняет точную копию введенного текста, он сохраняет семантически незначащие пробелы между элементами, а также порядок ключей в JSON-объектах. И если JSON-объект внутри содержит повторяющиеся ключи, этот тип сохранит все пары ключ/значение. Функции обработки будут считать действительной последнюю пару. Тип jsonb, напротив, не сохраняет пробелы, порядок ключей и значения с дублирующимися ключами. Если во входных данных оказываются дублирующиеся ключи, сохраняется только последнее значение.

Для большинства приложений предпочтительнее хранить данные JSON в типе jsonb (если нет особых противопоказаний, например важны прежние предположения о порядке ключей объектов).

Синергия-БД позволяет использовать только одну кодировку символов в базе данных, поэтому данные JSON не будут полностью соответствовать спецификации, если кодировка базы данных не UTF-8. При этом нельзя будет вставить символы, непредставимые в кодировке сервера, и наоборот, допустимыми будут символы, представимые в кодировке сервера, но не в UTF-8.

RFC 7159 разрешает включать в строки JSON спецпоследовательности

Unicode в виде `\uXXXX`. В функцию ввода для типа `json` эти спецпоследовательности допускаются вне зависимости от кодировки базы данных, и проверяется только правильность их синтаксиса (за `\u` должны следовать четыре шестнадцатеричных цифры). Однако, функция ввода для типа `jsonb` более строгая: она не допускает спецпоследовательности Unicode для не-ASCII символов (символов после U+007F), если кодировка базы данных не UTF8. Тип `jsonb` также не принимает `\u0000` (так как это значение не может быть представлено в типе `text` Синергия-БД), и требует, чтобы суррогатные пары Unicode использовались для представления символов вне основной многоязыковой плоскости (BMP) правильно. Корректные спецпоследовательности Unicode преобразуются для хранения в соответствующий символ ASCII или UTF8 (это подразумевает сворачивание суррогатных пар в один символ).

Примечание. Многие из функций обработки JSON преобразуют спецпоследовательности Unicode в обычные символы, поэтому могут выдавать подобные ошибки, даже если им на вход поступает тип `json`, а не `jsonb`. То, что функция ввода в тип `json` не производит этих проверок, можно считать историческим артефактом, хотя это и позволяет просто сохранять (но не обрабатывать) в JSON спецкоды Unicode в базе данных с кодировкой не UTF8. Вообще же, по возможности следует избегать смешения спецкодов Unicode в JSON с кодировкой базой данных не UTF8.

При преобразовании вводимого текста JSON в тип `jsonb`, примитивные типы, описанные в RFC 7159, по сути отображаются в собственные типы Синергия-БД как показано в таблице 6.23. Таким образом, к содержимому типа `jsonb` предъявляются некоторые дополнительные требования, продиктованные ограничениями представления нижележащего типа данных, которые не распространяются ни на тип `json`, ни на формат JSON вообще. В частности, тип `jsonb` не принимает числа, выходящие за диапазон типа данных Синергия-БД `numeric`, тогда как с `json` такого ограничения нет. Такие ограничения, накладываемые реализацией, допускаются согласно RFC 7159. Однако, на практике такие проблемы более вероятны в других реализациях, так как обычно примитивный тип JSON `number` представляется в виде числа с плавающей точкой двойной точности IEEE 754 (что RFC 7159 явно признает и допускает). При использовании JSON в качестве формата обмена данными с такими системами следует учитывать риски потери точности чисел, хранившихся в

Синергия-БД.

И напротив, как показано в таблице, есть некоторые ограничения в формате ввода примитивных типов JSON, не актуальные для соответствующих типов Синергия-БД.

Таблица 6.23 – Примитивные типы JSON и соответствующие им типы Синергия-БД

Примитивный тип JSON	Тип Синергия-БД	Примечания
string	text	\u0000 не допускается, как не ASCII символ, если кодировка базы данных не UTF8
number	numeric	Значения NaN и infinity не допускаются
boolean	boolean	Допускаются только варианты true и false (в нижнем регистре)
null	(нет)	NULL в SQL имеет другой смысл

6.14.1. Синтаксис вводимых и выводимых значений JSON

Синтаксис ввода/вывода типов данных JSON соответствует стандарту RFC 7159.

Примеры допустимых выражений с типом json (или jsonb):

-- Простое скалярное/примитивное значение

-- Простыми значениями могут быть числа, строки в кавычках, true, false или null

```
SELECT '5'::json;
```

-- Массив из нуля и более элементов (элементы могут быть разных типов)

```
SELECT '[1, 2, «foo», null]'::json;
```

-- Объект, содержащий пары ключей и значений

-- Заметьте, что ключи объектов – это всегда строки в кавычках

```
SELECT '{«bar»: «baz», «balance»: 7.77, «active»: false}'::json;
```

-- Массивы и объекты могут вкладываться произвольным образом

```
SELECT '{«foo»: [true, «bar»], «tags»: {«a»: 1, «b»: null}}'::json;
```

Как было сказано ранее, когда значение JSON вводится и затем выводится без

дополнительной обработки, тип `json` выводит тот же текст, что поступил на вход, а `jsonb` не сохраняет семантически незначащие детали, такие как пробелы.

Например, посмотрите на эти различия:

```
SELECT '{«bar»: «baz», «balance»: 7.77,
«active»:false}':::json;

json

-----

{«bar»: «baz», «balance»: 7.77, «active»:false}
(1 row)
```

```
SELECT '{«bar»: «baz», «balance»: 7.77,
«active»:false}':::jsonb;

jsonb

-----

{«bar»: «baz», «active»: false, «balance»: 7.77}
(1 row)
```

Первая семантически незначимая деталь, заслуживающая внимания: с `jsonb` числа выводятся по правилам нижележащего типа `numeric`. На практике это означает, что числа, заданные в записи с `E`, будут выведены без нее, например:

```
SELECT '{«reading»: 1.230e-5}':::json, '{«reading»:
1.230e-5}':::jsonb;

json          |          jsonb
-----+-----
{«reading»: 1.230e-5} | {«reading»: 0.00001230}
(1 row)
```

Однако, как видно из этого примера, `jsonb` сохраняет конечные нули дробного числа, хотя они и не имеют семантической значимости, в частности для проверки на равенство.

6.14.2. Эффективная организация документов JSON

Представлять данные в JSON можно гораздо более гибко, чем в традиционной реляционной модели данных, что очень привлекательно там, где нет жестких условий. И оба этих подхода вполне могут сосуществовать и дополнять друг друга в

одном приложении. Однако, даже для приложений, которым нужна максимальная гибкость, рекомендуется, чтобы документы JSON имели некоторую фиксированную структуру. Эта структура обычно не навязывается жестко (хотя можно декларативно диктовать некоторые бизнес-правила), но когда она предсказуема, становится гораздо проще писать запросы, которые извлекают полезные данные из набора «документов» (информации) в таблице.

Данные JSON, как и данные любых других типов, хранящиеся в таблицах, находятся под контролем механизма параллельного доступа. Хотя хранить большие документы вполне возможно, не забывайте, что при любом изменении устанавливается блокировка всей строки (на уровне строки). Поэтому для оптимизации блокировок транзакций, изменяющих данные, стоит ограничить размер документов JSON разумными пределами. В идеале каждый документ JSON должен собой представлять атомарный информационный блок, который, согласно бизнес-логике, нельзя разделить на меньшие, индивидуально изменяемые блоки.

6.14.3. Проверки на вхождение и существование jsonb

Проверка вхождения – важная особенность типа `jsonb`, не имеющая аналога для типа `json`. Эта проверка определяет, входит ли один документ `jsonb` в другой. В следующих примерах возвращается истинное значение (кроме упомянутых исключений):

```
-- Простые скалярные/примитивные значения включают  
только одно идентичное значение:
```

```
SELECT '«foo»'::jsonb @> '«foo»'::jsonb;
```

```
-- Массив с правой стороны входит в массив слева:
```

```
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;
```

```
-- Порядок элементов в массиве не важен, поэтому это  
условие тоже выполняется:
```

```
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;
```

```
-- А повторяющиеся элементы массива не имеют значения:
```



```
SELECT '[1, 2, 3]':::jsonb @> '[1, 2, 2]':::jsonb;
```

-- Объект с одной парой справа входит в объект слева:

```
SELECT '{«product»: «PostgreSQL», «version»: 9.4,  
«jsonb»:true}':::jsonb @> '{«version»:9.4}':::jsonb;
```

-- Массив справа не считается входящим в

-- массив слева, хотя в последний и вложен подобный массив:

```
SELECT '[1, 2, [1, 3]]':::jsonb @> '[1, 3]':::jsonb; --  
выдает false
```

-- Но если добавить уровень вложенности, проверка на
вхождение выполняется:

```
SELECT '[1, 2, [1, 3]]':::jsonb @> '[[1, 3]]':::jsonb;
```

-- Аналогично, это вхождением не считается:

```
SELECT '{«foo»: {«bar»: «baz»}}':::jsonb @> '{«bar»:  
«baz»}':::jsonb; -- выдает false
```

-- Ключ с пустым объектом на верхнем уровне входит в
объект с таким ключом:

```
SELECT '{"foo": {"bar": "baz"}}':::jsonb @> '{"foo":  
{}}':::jsonb;
```

Общий принцип этой проверки в том, что входящий объект должен соответствовать объекту, содержащему его, по структуре и данным, возможно, после исключения из содержащего объекта лишних элементов массива или пар ключ/значение. Но помните, что порядок элементов массива для проверки на вхождение не имеет значения, а повторяющиеся элементы массива считаются только один раз.

В качестве особого исключения для требования идентичности структур, массив может содержать примитивное значение:

-- В этот массив входит примитивное строковое значение:

```
SELECT '[«foo», «bar»]':::jsonb @> '«bar»':::jsonb;
```

```
-- Это исключение действует только в одну сторону --  
здесь вхождения нет:
```

```
SELECT '«bar»'::jsonb @> '«bar»'::jsonb; -- выдает  
false
```

Для типа `jsonb` введен также оператор существования, который является вариацией на тему вхождения: он проверяет, является ли строка (заданная в виде значения `text`) ключом объекта или элементом массива на верхнем уровне значения `jsonb`. В следующих примерах возвращается истинное значение (кроме упомянутых исключений):

```
-- Строка существует в качестве элемента массива:
```

```
SELECT '«foo», «bar», «baz»'::jsonb ? 'bar';
```

```
-- Строка существует в качестве ключа объекта:
```

```
SELECT '{«foo»: «bar»}'::jsonb ? 'foo';
```

```
-- Значения объектов не рассматриваются:
```

```
SELECT '{«foo»: «bar»}'::jsonb ? 'bar'; -- выдает false
```

```
-- Как и вхождение, существование определяется на  
верхнем уровне:
```

```
SELECT '{«foo»: {«bar»: «baz»}}'::jsonb ? 'bar'; --  
выдает false
```

```
-- Строка считается существующей, если она соответствует  
примитивной строке JSON:
```

```
SELECT '«foo»'::jsonb ? 'foo';
```

Объекты JSON для проверок на существование и вхождение со множеством ключей или элементов подходят больше, чем массивы, так как, в отличие от массивов, они внутри оптимизируются для поиска, и поиск элемента не будет линейным.

Вхождение в JSON проверяется с учётом вложенности, правильно написанный запрос может заменить явную выборку внутренних объектов. Например, предположим, что у нас есть столбец `doc`, содержащий объекты на верхнем уровне,

и большинство этих объектов содержит поля `tags` с массивами вложенных объектов. Данный запрос найдёт записи, в которых вложенные объекты содержат ключи `"term": "paris"` и `"term": "food"`, и при этом пропустит такие ключи, находящиеся вне массива `tags`:

```
SELECT doc->'site_name' FROM websites
-----

WHERE doc @> '{"tags": [{"term": "paris"},
{"term": "food"}] }';
```

Этого же результата можно добиться, например, так:

```
SELECT doc->'site_name' FROM websites
-----

WHERE doc->'tags' @> '[{"term": "paris"},
{"term": "food"}]';
```

Но данный подход менее гибкий и часто также менее эффективный.

С другой стороны, оператор существования JSON не учитывает вложенность: он будет искать заданный ключ или элемент массива только на верхнем уровне значения JSON.

6.14.4. Индексация `jsonb`

Для эффективного поиска ключей или пар ключ/значение в большом количестве документов `jsonb` можно успешно применять индексы GIN. Для этого предоставляются два «класса операторов» GIN, предлагающие выбор между производительностью и гибкостью.

Класс операторов GIN по умолчанию для `jsonb` поддерживает запросы с операторами существования ключа на верхнем уровне (`?`, `?&` и `?|`) и оператором существования пути/значения (`@>`). Пример создания индекса с этим классом операторов:

```
CREATE INDEX idxgin ON api USING GIN (jdoc);
```

Дополнительный класс операторов GIN `jsonb_path_ops` поддерживает индексацию только с оператором `@>`. Пример создания индекса с этим классом операторов:

```
CREATE INDEX idxginp ON api USING gin (jdoc
```

```
jsonb_path_ops);
```

Рассмотрим пример таблицы, в которой хранятся документы JSON, получаемые от сторонней веб-службы, с документированным определением схемы.

Типичный документ:

```
{
  «guid»: «9c36adc1-7fb5-4d5b-83b4-90356a46061a»,
  «name»: «Angela Barton»,
  «is_active»: true,
  «company»: «Magnaфone»,
  «address»: «178 Howard Place, Gulf, Washington,
702»,
  «registered»: «2009-11-07T08:53:22 +08:00»,
  «latitude»: 19.793713,
  «longitude»: 86.513373,
  «tags»: [
    «enim»,
    «aliquip»,
    «qui»
  ]
}
```

Мы сохраняем эти документы в таблице `api`, в колонке `jdoc` типа `jsonb`. Если по этой колонке создается GIN-индекс, он может применяться в подобных запросах:

```
-- Найти документы, в которых ключ «company» имеет
значение «Magnaфone»
```

```
----
```

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @>
'{«company»: «Magnaфone»}';
```

Однако, в следующих запросах он не будет использоваться, потому что, несмотря на то, что оператор `?` – индексируемой, он применяется не к индексированной колонке `jdoc` непосредственно:

```
-- Найти документы, в которых ключ «tags» содержит ключ
или элемент массива «qui»
```

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc ->
'tags' ? 'qui';
```

И все же, правильно применяя индексы выражений, в этом запросе можно задействовать индекс. Если запрос определенных элементов в ключе «tags» выполняется часто, вероятно стоит определить такой индекс:

```
CREATE INDEX idxgintags ON api USING gin ((jdoc ->
'tags'));
```

Теперь предложение WHERE jdoc -> 'tags' ? 'qui' будет выполняться как применение индексируемого оператора ? к индексируемому выражению jdoc -> 'tags'.

Ещё один подход к использованию проверок на существование:

-- Найти документы, в которых ключ «tags» содержит элемент массива «qui»

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @>
'{'tags': ['qui']}';
```

Этот запрос может задействовать простой GIN-индекс по колонке jdoc. Но заметьте, что такой индекс будет хранить копии всех ключей и значений в поле jdoc, тогда как индекс выражения из предыдущего примера хранит только данные внутри объекта с ключом tags. Хотя подход с простым индексом гораздо более гибкий (так как он поддерживает запросы по любому ключу), индексы конкретных выражений скорее всего будут меньше и быстрее, чем простые индексы.

Класс операторов jsonb_path_ops поддерживает только запросы с оператором @>, но зато он значительно производительнее класса по умолчанию jsonb_ops. Индекс jsonb_path_ops обычно гораздо меньше индекса jsonb_ops для тех же данных и более точен при поиске, особенно, если запросы обращаются к ключам, часто встречающимся в данных. Таким образом, с ним операции поиска выполняются гораздо лучше, чем с классом операторов по умолчанию.

Техническое различие между GIN-индексами jsonb_ops и jsonb_path_ops состоит в том, что для первых создаются независимые элементы индекса для каждого ключа/значения в данных, тогда как для вторых создаются

элементы только для значений. По сути, каждый элемент индекса `jsonb_path_ops` представляет собой хеш значения и ключа(ей), приводящего к нему; например, при индексации `{«foo»: {«bar»: «baz»}}` будет создан один элемент индекса с хешем, рассчитанным по всем трем значениям: `foo`, `bar` и `baz`. Таким образом, проверка на вхождение этой структуры будет использовать крайне точный поиск по индексу, но определить, является ли `foo` ключом, с помощью такого индекса нельзя. С другой стороны, индекс `jsonb_ops` создаст три отдельных элемента индекса, представляющих `foo`, `bar` и `baz` по отдельности; для выполнения проверки на вхождение будут проверены строки таблицы, содержащие все эти три значения. Хотя GIN-индексы позволяют вычислить AND довольно эффективно, такой поиск все же будет менее точным и более медленным, чем равнозначный поиск с `jsonb_path_ops`, особенно если любое одно из этих трех значений содержится в большом количестве строк.

Недостаток класса `jsonb_path_ops` заключается в том, что он не учитывает в индексе структуры JSON, не содержащие никаких значений `{«a»: {}}`. Для поиска по документам, содержащих такие структуры, потребуется выполнить полное сканирование индекса, что довольно долго, поэтому `jsonb_path_ops` не очень подходит для приложений, часто выполняющих такие запросы.

Тип `jsonb` также поддерживает индексы `btree` и `hash`. Они полезны, только если требуется проверять равенство JSON-документов в целом. Порядок сортировки `btree` для типа `jsonb` редко имеет большое значение, но для полноты он приводится ниже:

Объект > Массив > Логическое значение > Число > Строка > Null

Объект с n парами > Объект с n - 1 парами

Массив с n элементами > Массив с n - 1 элементами

Объекты с равным количеством пар сравниваются в таком порядке:

ключ-1, значение-1, ключ-2 ...

Заметьте, что ключи объектов сравниваются согласно порядку при хранении; в частности, из-за того, что короткие ключи хранятся перед длинными, результаты

могут оказаться несколько не интуитивными:

```
{ «aa»: 1, «c»: 1 } > { «b»: 1, «d»: 1 }
```

Массивы с равным числом элементов упорядочиваются аналогично:

элемент-1, элемент-2 ...

Примитивные значения JSON сравниваются по тем же правилам сравнения, что и лежащие типы данных Синергия-БД. Строки сравниваются с учетом порядка сортировки по умолчанию в текущей базе данных.

6.15. Трансформации

Для различных процедурных языков представлены дополнительные расширения, реализующие трансформации для типа `jsonb`.

Расширения для PL/Perl называются `jsonb_plperl` и `jsonb_plperlu`. Когда они используются, значения `jsonb` отображаются в соответствующие структуры Perl: массивы, хеши или скаляры.

Расширения для PL/Python называются `jsonb_plpythonu`, `jsonb_plpython2u` и `jsonb_plpython3u`. Когда они используются, значения `jsonb` отображаются в соответствующие структуры Python: массивы, хеши или скаляры.

6.16. Массивы

Синергия-БД позволяет определять колонки таблицы как многомерные массивы переменной длины. Элементами массивов могут быть любые встроенные или определенные пользователями типы, перечисления или составные типы.

6.16.1. Объявления типов массивов

Чтобы проиллюстрировать использование массивов, мы создадим такую таблицу:

```
CREATE TABLE sal_emp (  
    name                text,  
    pay_by_quarter      integer[],  
    schedule             text[][]  
);
```

Как показано, для объявления типа массива к названию типа элементов добавляются квадратные скобки ([]). Показанная выше команда создаст таблицу `sal_emp` с колонками типов `text` (`name`), одномерный массив с элементами `integer` (`pay_by_quarter`), представляющий квартальную зарплату работников и двухмерный массив с элементами `text` (`schedule`), представляющий недельный график работника.

Команда `CREATE TABLE` позволяет также указать точный размер массивов, например так:

```
CREATE TABLE tictactoe (  
    squares    integer[3][3]  
);
```

Однако текущая реализация игнорирует все указанные размеры, т.е. фактически размер массива остается неопределенным.

Текущая реализация также не ограничивает число размерностей. Все элементы массивов считаются одного типа, вне зависимости от его размера и числа размерностей. Поэтому явно указывать число элементов или размерностей в команде `CREATE TABLE` имеет смысл только для документирования, на механизм работы с массивом это не влияет.

Для объявления одномерных массивов можно применять альтернативную запись с ключевым словом `ARRAY`, соответствующую стандарту SQL. Колонку `pay_by_quarter` можно было бы определить так:

```
pay_by_quarter    integer ARRAY[4],
```

Или без указания размера массива:

```
pay_by_quarter    integer ARRAY,
```

Заметьте, что и в этом случае Синергия-БД не накладывает ограничения на фактический размер массива.

6.16.2. Ввод значения массива

Чтобы записать значение массива в виде буквальной константы, заключите значения элементов в фигурные скобки и разделите их запятыми. Если вам знаком C,

вы найдете, что это похоже на синтаксис инициализации структур в С. Вы можете заключить значение любого элемента в двойные кавычки, а если он содержит запятые или фигурные скобки, это обязательно нужно сделать. Таким образом, общий формат константы массива выглядит так:

```
'{ значение1 разделитель значение2 разделитель ... }'
```

где `разделитель` – символ, указанный в качестве разделителя в соответствующей записи в таблице `pg_type`. Для стандартных типов данных, существующих в дистрибутиве Синергия-БД, разделителем является запятая (`,`), за исключением лишь типа `box`, в котором разделитель – точка с запятой (`;`). Каждое значение здесь – это либо константа типа элемента массива, либо вложенный массив. Например, константа массива может быть такой:

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

Эта константа определяет двумерный массив `3x3`, состоящий из трех вложенных массивов целых чисел.

Чтобы присвоить элементу массива значение `NULL`, достаточно просто написать `NULL` (регистр символов при этом не имеет значения). Если же требуется добавить в массив строку, содержащую «`NULL`», это слово нужно заключить в двойные кавычки.

Такого рода константы массивов на самом деле представляют собой всего лишь частный случай констант, описанных в подразделе 2.1.2.7. Константа изначально воспринимается как строка и передается процедуре преобразования вводимого массива. При этом может потребоваться явно указать целевой тип.

Теперь мы можем показать несколько операторов `INSERT`:

```
INSERT INTO sal_emp
VALUES ('Bill',
       '{10000, 10000, 10000, 10000}',
       '{{«meeting», «lunch»}, {«training»,
«presentation»}}');
```

```
INSERT INTO sal_emp
VALUES ('Carol',
```

```
'{20000, 25000, 25000, 25000}',
'{{«breakfast», «consulting»}, {«meeting»,
«lunch»}}')';
```

Результат двух предыдущих команд:

```
SELECT * FROM sal_emp;
```

```
name |          pay_by_quarter          |          schedule
-----+-----+-----
```

```
Bill
```

```
|{10000,10000,10000,10000}|{{meeting,lunch},{training,pr
esentation}}
```

```
Carol|{20000,25000,25000,25000}|{{breakfast,consulting},
{meeting,lunch}}
```

```
(2 rows)
```

В многомерных массивов число элементов в каждой размерности должно быть одинаковым; в противном случае возникает ошибка, например:

```
INSERT INTO sal_emp
VALUES ('Bill',
'{10000, 10000, 10000, 10000}',
'{{«meeting», «lunch»}, {«meeting»}}');
```

ОШИБКА: для многомерных массивов должны задаваться выражения

с соответствующими размерностями

Также можно использовать синтаксис конструктора ARRAY:

```
INSERT INTO sal_emp
VALUES ('Bill',
ARRAY[10000, 10000, 10000, 10000],
ARRAY[['meeting', 'lunch'], ['training',
'presentation']]);
```

```
INSERT INTO sal_emp
VALUES ('Carol',
ARRAY[20000, 25000, 25000, 25000],
ARRAY[['breakfast', 'consulting'], ['meeting',
'lunch']]);
```

Заметьте, что элементы массива здесь – это простые SQL-константы или выражения; и поэтому, например строки будут заключаться в одинарные апострофы, а не в двойные, как в буквальной константе массива. Более подробно конструктор ARRAY обсуждается в подразделе 2.2.12.

6.16.3. Обращение к массивам

Добавив данные в таблицу, мы можем перейти к выборкам. Сначала мы покажем, как получить один элемент массива. Этот запрос получает имена сотрудников, зарплата которых изменилась во втором квартале:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <>
pay_by_quarter[2];
```

```
name
-----
Carol
(1 row)
```

Индексы элементов массива записываются в квадратных скобках. По умолчанию в Синергия-БД действует соглашение о нумерации элементов массива с 1, то есть в массиве из n элементов первым считается array[1], а последним – array[n].

Этот запрос выдает зарплату всех сотрудников в третьем квартале:

```
SELECT pay_by_quarter[3] FROM sal_emp;
```

```
pay_by_quarter
-----
10000
25000
(2 rows)
```

Мы также можем получать обычные прямоугольные срезы массива, то есть подмассивы. Срез массива обозначается как нижняя-граница:верхняя-граница для одной или нескольких размерностей. Например, этот запрос получает первые пункты в графике Билла в первые два дня недели:

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name =  
'Bill';
```

```
          schedule  
-----  
  {{meeting},{training}}  
(1 row)
```

Если одна из размерностей записана в виде среза, то есть содержит двоеточие, тогда срез распространяется на все размерности. Если при этом для размерности указывается только одно число (без двоеточия), в срез войдут элемент от 1 до заданного номера. Например, в этом примере [2] будет равнозначно [1:2]:

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name =  
'Bill';
```

```
          schedule  
-----  
  {{meeting,lunch},{training,presentation}}  
(1 row)
```

Во избежание путаницы с обращением к одному элементу, срезы лучше всегда записывать явно для всех измерений, например [1:2][1:1] вместо [2][1:1].

Значения нижняя-граница и/или верхняя-граница в указании среза можно опустить; опущенная граница заменяется нижним или верхним пределом индексов массива, например:

```
SELECT schedule[:2][2:] FROM sal_emp WHERE name =  
'Bill';
```

```
          schedule  
-----  
  {{lunch},{presentation}}  
(1 row)
```

```
SELECT schedule[:][1:1] FROM sal_emp WHERE name =  
'Bill';
```

```

schedule
-----
{{meeting},{training}}
(1 row)

```

Выражение обращения к элементу массива возвратит NULL, если сам массив или одно из выражений индексов элемента равны NULL. Значение NULL также возвращается, если индекс выходит за границы массива (это не считается ошибкой). Например, если `schedule` в настоящее время имеет размерности `[1:3][1:2]`, результатом обращения к `schedule[3][3]` будет NULL. Подобным образом, при обращении к элементу массива с неправильным числом индексов возвращается NULL, а не ошибка.

Аналогично, NULL возвращается при обращении к срезу массива, если сам массив или одно из выражений, определяющих индексы элементов, равны NULL. Однако, в других случаях, например, когда границы среза выходят за рамки массива, возвращается не NULL, а пустой массив (с размерностью 0). Так сложилось исторически, что в этом срезы отличаются от обращений к обычным элементам. Если запрошенный срез пересекает границы массива, тогда возвращается не NULL, а срез, сокращенный до области пересечения.

Текущие размеры значения массива можно получить с помощью функции `array_dims`:

```
SELECT array_dims(schedule) FROM sal_emp WHERE name =
'Carol';
```

```

array_dims
-----
[1:2][1:2]
(1 row)

```

`array_dims` выдает результат типа `text`, что удобно скорее для людей, чем для программ. Размеры массива также можно получить с помощью функций `array_upper` и `array_lower`, которые возвращают соответственно верхнюю и

нижнюю границу для указанной размерности:

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name  
= 'Carol';
```

```
array_upper  
-----  
2
```

(1 row)

array_length возвращает число элементов в указанной размерности

массива:

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name  
= 'Carol';
```

```
array_length  
-----  
2
```

(1 row)

cardinality возвращает общее число элементов массива по всем

измерениям. Фактически это число строк, которое вернет функция unnest:

```
SELECT cardinality(schedule) FROM sal_emp WHERE name =  
'Carol';
```

```
cardinality  
-----  
4
```

(1 row)

6.16.4. Изменение массивов

Значение массива можно заменить полностью так:

```
UPDATE sal_emp SET pay_by_quarter =  
'{25000,25000,27000,27000}'  
WHERE name = 'Carol';
```

или используя синтаксис ARRAY:

```
UPDATE sal_emp SET pay_by_quarter =
```

```
ARRAY[25000,25000,27000,27000]
```

```
WHERE name = 'Carol';
```

Также можно изменить один элемент массива:

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
```

```
WHERE name = 'Bill';
```

или срез:

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
```

```
WHERE name = 'Carol';
```

При этом в указании среза может быть опущена нижняя-граница и/или верхняя-граница, но только для массива, отличного от NULL, и имеющего ненулевую размерность (иначе неизвестно, какие граничные значения должны подставляться вместо опущенных).

Сохраненный массив можно расширить, определив значения ранее отсутствовавших в нем элементов. При этом все элементы, располагающиеся между существовавшими ранее и новыми, принимают значения NULL. Например, если массив `myarray` содержит 4 элемента, после присвоения значения элементу `myarray[6]` его длина будет равна 6, а `myarray[5]` будет содержать NULL. В настоящее время подобное расширение поддерживается только для одномерных, но не многомерных массивов.

Определяя элементы по индексам, можно создавать массивы, в которых нумерация элементов может начинаться не с 1. Например, можно присвоить значение выражению `myarray[-2:7]` и таким образом создать массив, в котором будут элементы с индексами от -2 до 7.

Значения массива также можно сконструировать с помощью оператора конкатенации, ||:

```
SELECT ARRAY[1,2] || ARRAY[3,4];
```

```
?column?
```

```
-----
```

```
{1,2,3,4}
```

```
(1 row)
```

```
SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
      ?column?
```

```
-----
{{5,6},{1,2},{3,4}}
```

```
(1 row)
```

Оператор конкатенации позволяет вставить один элемент в начало или в конец одномерного массива. Он также может принять два N-мерных массива или массивы размерностей N и N+1.

Когда в начало или конец одномерного массива вставляется один элемент, в образованном в результате массиве будет та же нижняя граница, что и в массиве-операнде, например:

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
      array_dims
```

```
-----
[0:2]
(1 row)
```

```
SELECT array_dims(ARRAY[1,2] || 3);
      array_dims
```

```
-----
[1:3]
(1 row)
```

Когда складываются два массива одинаковых размерностей, в результате сохраняется нижняя граница внешней размерности левого операнда. Выходной массив включает все элементы левого операнда, после которых добавляются все элементы правого, например:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
      array_dims
```

```
-----
[1:5]
(1 row)
```



```
SELECT array_dims(ARRAY[[1,2],[3,4]] ||  
ARRAY[[5,6],[7,8],[9,0]]);
```

```
array_dims  
-----  
[1:5][1:2]  
(1 row)
```

Когда к массиву размерности N+1 спереди или сзади добавляется N-мерный массив, он вставляется аналогично тому, как в массив вставляется элемент (это было описано выше). Любой N-мерный массив по сути является элементом во внешней размерности массива, имеющего размерность N+1, например:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
```

```
array_dims  
-----  
[1:3][1:2]  
(1 row)
```

Массив также может быть сконструирован с помощью функций `array_prepend`, `array_append` или `array_cat`. Первые две функции поддерживают только одномерные массивы, а `array_cat` поддерживает и многомерные. Несколько примеров:

```
SELECT array_prepend(1, ARRAY[2,3]);
```

```
array_prepend  
-----  
{1,2,3}  
(1 row)
```

```
SELECT array_append(ARRAY[1,2], 3);
```

```
array_append  
-----  
{1,2,3}  
(1 row)
```

```
SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
```

```
array_cat
```

```
{1,2,3,4}
(1 row)
```

```
SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
        array_cat
```

```
{{1,2},{3,4},{5,6}}
(1 row)
```

```
SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
        array_cat
```

```
{{5,6},{1,2},{3,4}}
```

В простых случаях оператор конкатенации, рассмотренный выше, предпочтительнее непосредственного вызова указанных функций. Однако, поскольку оператор конкатенации перегружен, чтобы подходить для всех трех случаев, возможны ситуации, когда использовании функции помогает избежать неоднозначности.

Пример.

```
SELECT ARRAY[1, 2] || '{3, 4}'; -- нетипизированный
литерал принимается за массив
```

```
?column?
```

```
{1,2,3,4}
```

```
SELECT ARRAY[1, 2] || '7'; -- и этот
тоже
```

```
ERROR:  malformed array literal: «7»
```

```
SELECT ARRAY[1, 2] || NULL; -- то же
самое и с NULL
```

```
?column?
```

```
{1,2}
(1 row)
```

```
SELECT array_append(ARRAY[1, 2], NULL);      -- а возможно
имелось в виду это
```

```
array_append
-----
{1,2,NULL}
```

В показанных примерах анализатор запроса видит целочисленный массив с одной стороны оператора конкатенации и константу неопределённого типа с другой. Согласно своим правилам разрешения типа констант, он полагает, что она имеет тот же тип, что и другой операнд – в данном случае целочисленный массив. Поэтому предполагается, что оператор конкатенации здесь представляет функцию `array_cat`, а не `array_append`. Если это решение оказывается неверным, его можно скорректировать, приведя константу к типу элемента массива; однако может быть лучше явно использовать функцию `array_append`.

6.16.5. Поиск значений в массивах

Чтобы найти значение в массиве, необходимо проверить все его элементы. Это можно сделать вручную, если вы знаете размер массива, например:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                                pay_by_quarter[2] = 10000 OR
                                pay_by_quarter[3] = 10000 OR
                                pay_by_quarter[4] = 10000;
```

Однако с большим массивами этот метод становится утомительным, и к тому же он не работает, когда размер массива неизвестен. Показанный выше запрос можно было переписать так:

```
SELECT * FROM sal_emp WHERE 10000 = ANY
(pay_by_quarter);
```

А так можно найти в таблице строки, в которых массивы содержат только значения, равные 10000:

```
SELECT * FROM sal_emp WHERE 10000 = ALL
(pay_by_quarter);
```

Кроме того, для обращения к элементам массива можно использовать функцию `generate_subscripts`. Например, так:

```
SELECT * FROM
    (SELECT pay_by_quarter,
        generate_subscripts(pay_by_quarter, 1) AS s
    FROM sal_emp) AS foo
WHERE pay_by_quarter[s] = 10000;
```

Также искать в массиве значения можно, используя оператор `&&`, который проверяет, перекрывается ли левый операнд с правым, например:

```
SELECT * FROM sal_emp WHERE pay_by_quarter &&
ARRAY[10000];
```

Вы также можете искать определённые значения в массиве, используя функции `array_position` и `array_positions`. Первая функция возвращает позицию первого вхождения значения в массив, а вторая – массив позиций всех его вхождений, например:

```
SELECT
array_position(ARRAY['sun','mon','tue','wed','thu','fri',
    'sat'], 'mon');
```

```
array_positions
-----
2
```

```
SELECT array_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1],
1);
```

```
array_positions
-----
{1,4,8}
```

Примечание. Массивы – это не множества; необходимость поиска определенных элементов в массиве может быть признаком неудачно сконструированной базы данных. Возможно, вместо массива лучше использовать отдельную таблицу, строки которой будут содержать данные элементов массива. Это может быть лучше и для поиска, и для работы с большим количеством элементов.

6.16.6. Синтаксис вводимых и выводимых значений массива

Внешнее текстовое представление значения массива состоит из записи

элементов, интерпретируемых по правилам ввода/вывода для типа элемента массива, и оформления структуры массива. Оформление состоит из фигурных скобок ({ и }), окружающих значение массива, и знаков-разделителей между его элементами. В качестве знака-разделителя обычно используется запятая (,), но это может быть и другой символ; он определяется параметром `typdelim` для типа элемента массива. Для стандартных типов данных, существующих в дистрибутиве Синергия-БД, разделителем является запятая (,), за исключением лишь типа `box`, в котором разделитель – точка с запятой (;). В многомерном массиве у каждой размерности (ряд, плоскость, куб и т.д.) есть свой уровень фигурных скобок, а соседние значения в фигурных скобках на одном уровне должны отделяться разделителями.

Функция вывода массива заключает значение элемента в кавычки, если это пустая строка или оно содержит фигурные скобки, знаки-разделители, кавычки, обратную косую черту, пробельный символ или это текст `NULL`. Кавычки и обратная косая черта, включенные в такие значения, преобразуются в спецпоследовательность с обратной косой чертой. Для числовых типов данных можно рассчитывать на то, что значения никогда не будут выводиться в кавычках, но для текстовых типов следует быть готовым к тому, что выводимое значение массива может содержать кавычки.

По умолчанию нижняя граница всех размерностей массива равна одному. Чтобы представить массивы с другими нижними границами, перед содержимым массива можно указать диапазоны индексов. Такое оформление массива будет содержать квадратные скобки ([]) вокруг нижней и верхней границ каждой размерности с двоеточием (:) между ними. За таким указанием размерности следует знак равно (=), например:

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
FROM (SELECT '[1:1][-2:-
1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;
```

```
e1 | e2
----+----
1 | 6
```

(1 row)

Процедура вывода массива включает в результат явное указание размерностей, только если нижняя граница в одной или нескольких размерностях отличается от 1.

Если в качестве значения элемента задается NULL (в любом регистре), этот элемент считается равным непосредственно NULL. Если же оно включает кавычки или обратную косую черту, элементу присваивается текстовая строка «NULL». Кроме того, для обратной совместимости с версиями PostgreSQL, параметр конфигурации `array_nulls` можно выключить (присвоив ему `off`), чтобы строки NULL не воспринимались как значения NULL.

Как было показано ранее, записывая значение массива, любой его элемент можно заключить в кавычки. Это нужно делать, если при разборе значения массива без кавычек возможна неоднозначность. Например, в кавычки необходимо заключать элементы, содержащие фигурные скобки, запятую (или разделитель, определенный для данного типа), кавычки, обратную косую черту, а также пробельные символы в начале или конце строки. Пустые строки и строки, содержащие одно слово NULL, также нужно заключать в кавычки. Чтобы включить кавычки или обратную косую черту в значение, заключенное в кавычки, используйте спецпоследовательности с обратной косой чертой. С другой стороны, чтобы обойтись без кавычек, такими спецпоследовательностями можно защитить все символы в данных, которые могут быть восприняты как часть синтаксиса массива.

Перед открывающей и после закрывающей скобки можно добавлять пробельные символы. Пробелы также могут окружать каждую отдельную строку значения. Во всех случаях такие пробельные символы игнорируются. Однако все пробелы в строках, заключенных в кавычки, или окруженные не пробельными символами, напротив, учитываются.

Записывать значения массивов в командах SQL часто бывает удобнее с помощью конструктора `ARRAY`. В `ARRAY` отдельные значения элементов записываются так же, как если бы они не были членами массива.

6.17. Составные типы

Составной тип представляет структуру табличной строки или записи; по сути это просто список имен полей и соответствующих типов данных. Синергия-БД позволяет использовать составные типы во многом так же, как и простые типы. Например, в определении таблицы можно объявить колонку составного типа.

6.17.1. Объявление составных типов

Ниже приведены два простых примера определения составных типов:

```
CREATE TYPE complex AS (  
    r          double precision,  
    i          double precision  
);
```

```
CREATE TYPE inventory_item AS (  
    name          text,  
    supplier_id   integer,  
    price         numeric  
);
```

Синтаксис очень похож на `CREATE TABLE`, за исключением того, что он допускает только названия полей и их типы, какие-либо ограничения (такие как `NOT NULL`) в настоящее время не поддерживаются. Заметьте, что ключевое слово `AS` здесь имеет значение; без него система будет считать, что подразумевается другой тип команды `CREATE TYPE`, и выдаст неожиданную синтаксическую ошибку.

Определив такие типы, мы можем использовать их в таблицах:

```
CREATE TABLE on_hand (  
    item          inventory_item,  
    count         integer  
);
```

```
INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99),  
1000);
```

или функциях:

```
CREATE FUNCTION price_extension(inventory_item, integer)
```

```
RETURNS numeric
AS 'SELECT $1.price * $2' LANGUAGE SQL;
```

```
SELECT price_extension(item, 10) FROM on_hand;
```

Всякий раз, когда создается таблица, вместе с ней автоматически создается составной тип, представляющий тип строки таблицы, именем которого будет имя таблицы. Например, при выполнении команды:

```
CREATE TABLE inventory_item (
    name                text,
    supplier_id         integer REFERENCES suppliers,
    price               numeric CHECK (price > 0)
);
```

в качестве побочного эффекта будет создан составной тип `inventory_item`, в точности соответствующий тому, что был показан выше, и использовать его можно так же. Заметьте, что в текущей реализации есть один недостаток: так как с составным типом не могут быть связаны ограничения, описанные в определении таблицы ограничения не применяются к значениям составного типа вне таблицы. Чтобы обойти этот недостаток, создайте домен поверх составного типа и добавьте желаемые ограничения в виде ограничений `CHECK` для данного домена.

6.17.2. Конструирование составных значений

Чтобы записать значение составного типа в виде текстовой константы, его поля нужно заключить в круглые скобки и разделить их запятыми. Значение любого поля можно заключить в кавычки, а если оно содержит запятые или скобки, это делать обязательно. Подробности описаны ниже. Таким образом, в общем виде константа составного типа записывается так:

```
'( значение1 , значение2 , ... )'
```

Например, эта запись:

```
'(«fuzzy dice», 42, 1.99)'
```

будет допустимой для описанного выше типа `inventory_item`. Чтобы присвоить `NULL` одному из полей, в соответствующем месте в списке нужно

оставить пустое место. Например, эта константа задает значение NULL для третьего поля:

```
' («fuzzy dice», 42, ) '
```

Если же вместо NULL требуется вставить пустую строку, нужно записать пару кавычек:

```
' («», 42, ) '
```

Здесь в первом поле окажется пустая строка, а в третьем – NULL.

Такого рода константы массивов на самом деле представляют собой всего лишь частный случай констант, описанных в подразделе 2.1.2.7. Константа изначально воспринимается как строка и передается процедуре преобразования составного типа. При этом может потребоваться явно указать тип к которому будет приведена константа.

Значения составных типов также можно конструировать, используя синтаксис выражения ROW. В большинстве случаев это значительно проще, чем записывать значения в строке, так как при этом не нужно беспокоиться о вложенности кавычек. Мы уже обсуждали этот метод ранее:

```
ROW('fuzzy dice', 42, 1.99)
```

```
ROW(' ', 42, NULL)
```

Ключевое слово ROW на самом деле может быть необязательным, если в выражении определяются несколько полей, так что эту запись можно упростить до:

```
('fuzzy dice', 42, 1.99)
```

```
(' ', 42, NULL)
```

Синтаксис выражения ROW более подробно рассматривается в подразделе 2.2.13.

6.17.3. Обращение к составным типам

Чтобы обратиться к полю колонки составного типа, после имени колонки нужно добавить точку и имя поля, подобно тому, как указывается колонка после имени таблицы. На самом деле, эти обращения неотличимы, так что часто бывает необходимо использовать скобки, чтобы команда была разобрана правильно. Например, можно попытаться выбрать поле колонки из тестовой таблицы on_hand

таким образом:

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

Но это не будет работать, так как согласно правилам SQL имя `item` здесь воспринимается как имя таблицы, а не колонки в таблице `on_hand`. Поэтому этот запрос нужно переписать так:

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

либо указать также и имя таблицы (например, в запросе с многими таблицами), примерно так:

```
SELECT (on_hand.item).name FROM on_hand WHERE  
(on_hand.item).price > 9.99;
```

В результате объект в скобках будет правильно интерпретирован как ссылка на колонку `item`, из которой выбирается поле.

При выборке поля из значения составного типа также возможны подобные синтаксические казусы. Например, чтобы выбрать одно поле из результата функции, возвращающей составное значение, потребуется написать что-то подобное:

```
SELECT (my_func(...)).field FROM ...
```

Без дополнительных скобок в этом запросе произойдет синтаксическая ошибка.

6.17.4. Изменение составных типов

Ниже приведены примеры правильных команд добавления и изменения значений составных колонок. Первые команды иллюстрируют добавление или изменение всей колонки:

```
INSERT INTO mytab (complex_col) VALUES ((1.1,2.2));
```

```
UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

В первом примере опущено ключевое слово `ROW`, а во втором оно есть; присутствовать или отсутствовать оно может в обоих случаях.

Мы можем изменить также отдельное поле составной колонки:

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1  
WHERE ...;
```

Заметьте, что при этом не нужно (и на самом деле даже нельзя) заключать в скобки имя колонки, следующее сразу за предложением SET, но в ссылке на ту же колонку в выражении, находящемся по правую сторону знака равенства, скобки обязательны.

И мы также можем указать поля в качестве цели команды INSERT:

```
INSERT INTO mytab (complex_col.r, complex_col.i)
VALUES (1.1, 2.2);
```

Если при этом мы не укажем значения для всех полей колонки, оставшиеся поля будут заполнены значениями NULL.

6.17.5. Использование составных типов в запросах

С составными типами в запросах связаны особые правила синтаксиса и поведение. Эти правила образуют полезные конструкции, но они могут быть неочевидными, если не понимать стоящую за ними логику.

В Синергия-БД ссылка на имя таблицы (или её псевдоним) в запросе по сути является ссылкой на составное значение текущей строки в этой таблице. Например, имея таблицу `inventory_item`, показанную выше, мы можем написать:

```
SELECT c FROM inventory_item c;
```

Этот запрос выдаёт один столбец с составным значением, и его результат может быть таким:

```
      c
-----
("fuzzy dice", 42, 1.99)
(1 row)
```

Заметьте, однако, что простые имена сопоставляются сначала с именами столбцов, и только потом с именами таблиц, так что такой результат получается только потому, что в таблицах запроса не оказалось столбца с именем `c`.

Обычную запись полного имени столбца вида `имя_таблицы.имя_столбца` можно понимать как применение выбора поля к составному значению текущей строки таблицы. Из соображений эффективности на самом деле это реализовано по-другому.

Когда мы пишем:

```
SELECT c.* FROM inventory_item c;
```

то, согласно стандарту SQL, мы должны получить содержимое таблицы, развёрнутое в отдельные столбцы:

```

      name      | supplier_id | price
-----+-----+-----
fuzzy dice |          42 |   1.99
(1 row)
```

как с запросом:

```
SELECT c.name, c.supplier_id, c.price FROM
inventory_item c;
```

Синергия-БД применяет такое развёртывание для любых выражений с составными значениями, хотя как показано выше, необходимо заключить в скобки значение, к которому применяется `.*`, если только это не простое имя таблицы. Например, если `myfunc()` – функция, возвращающая составной тип со столбцами `a`, `b` и `c`, то эти два запроса выдадут одинаковый результат:

```

SELECT (myfunc(x)).* FROM some_table;
SELECT (myfunc(x)).a, (myfunc(x)).b, (myfunc(x)).c FROM
some_table;
```

Примечание. Синергия-БД осуществляет развёртывание столбцов фактически переводя первую форму во вторую. Таким образом, в данном примере `myfunc()` будет вызываться три раза для каждой строки и с одним, и с другим синтаксисом. Если это дорогостоящая функция, и вы хотите избежать лишних вызовов, можно использовать такой запрос:

```
SELECT m.* FROM some_table, LATERAL myfunc(x) AS m;
```

Размещение вызова функции в элементе `FROM LATERAL` гарантирует, что она будет вызываться для строки не более одного раза. Конструкция `m.*` так же разворачивается в `m.a`, `m.b`, `m.c`, но теперь эти переменные просто ссылаются на выходные значения `FROM`. Ключевое слово `LATERAL` здесь является необязательным, но мы добавили его, чтобы подчеркнуть, что функция получает `x` из `some_table`.

Запись `составное_значение.*` приводит к такому развёртыванию столбцов, когда она фигурирует на верхнем уровне выходного списка `SELECT`, в списке

RETURNING команд INSERT/UPDATE/DELETE, в предложении VALUES или в конструкторе строки. Во всех других контекстах (включая вложенные в одну из этих конструкций), добавление .* к составному значению не меняет это значение, так как это воспринимается как «все столбцы» и поэтому выдаётся то же составное значение. Например, если функция somefunc() принимает в качестве аргумента составное значение, эти запросы равносильны:

```
SELECT somefunc(c.*) FROM inventory_item c;  
SELECT somefunc(c) FROM inventory_item c;
```

В обоих случаях текущая строка таблицы inventory_item передаётся функции как один аргумент с составным значением. И хотя дополнение .* в этих случаях не играет роли, использовать его считается хорошим стилем, так как это ясно указывает на использование составного значения. В частности анализатор запроса воспримет c в записи c.* как ссылку на имя или псевдоним таблицы, а не имя столбца, что избавляет от неоднозначности; тогда как без .* неясно, означает ли c имя таблицы или имя столбца, и на самом деле при наличии столбца с именем c будет выбрано второе прочтение.

Эту концепцию демонстрирует и следующий пример, все запросы в котором действуют одинаково:

```
SELECT * FROM inventory_item c ORDER BY c;  
SELECT * FROM inventory_item c ORDER BY c.*;  
SELECT * FROM inventory_item c ORDER BY ROW(c.*);
```

Все эти предложения ORDER BY обращаются к составному значению строки. Однако, если в inventory_item содержится столбец с именем c, первый запрос будет отличаться от других, так как в нём выполнится сортировка только по данному столбцу. С показанными выше именами столбцов предыдущим запросам также равнозначны следующие:

```
SELECT * FROM inventory_item c ORDER BY ROW(c.name,  
c.supplier_id, c.price);  
SELECT * FROM inventory_item c ORDER BY (c.name,  
c.supplier_id, c.price);
```

В последнем случае используется конструктор строки, в котором опущено

ключевое слово ROW.

Другая особенность синтаксиса, связанная с составными значениями, состоит в том, что мы можем использовать функциональную запись для извлечения поля составного значения. Это легко можно объяснить тем, что записи *поле(таблица)* и *таблица.поле* взаимозаменяемы. Например, следующие запросы равнозначны:

```
SELECT c.name FROM inventory_item c WHERE c.price > 1000;
```

```
SELECT name(c) FROM inventory_item c WHERE price(c) > 1000;
```

Более того, если у нас есть функция, принимающая один аргумент составного типа, мы можем вызвать её в любой записи. Все эти запросы равносильны:

```
SELECT somefunc(c) FROM inventory_item c;
```

```
SELECT somefunc(c.*) FROM inventory_item c;
```

```
SELECT c.somefunc FROM inventory_item c;
```

Эта равнозначность записи с полем и функциональной записи позволяет использовать с составными типами функции, реализующие «вычисляемые поля». При этом приложению, использующему последний из предыдущих запросов, не нужно знать, что фактически *somefunc* – не настоящий столбец таблицы.

Учитывая такое поведение, будет неразумно давать функции, принимающей один аргумент составного типа, то же имя, что и одному из полей данного составного типа. В случае неоднозначности прочтение имени поля будет выбрано при использовании синтаксиса обращения к полю, а прочтение имени функции – если используется синтаксис вызова функции.

6.17.6. Синтаксис вводимых и выводимых значений составного типа

Внешнее текстовое представление составного значения состоит из записи элементов, интерпретируемых по правилам ввода/вывода для соответствующих типов полей, и оформления структуры составного типа. Оформление состоит из круглых скобок ((и)) окружающих все значение, и запятых (,) между его элементами. Пробельные символы вне скобок игнорируются, но внутри они считаются частью соответствующего элемента и могут учитываться или не учитываться в зависимости от правил преобразования вводимых данных для типа

этого элемента. Например, в записи:

```
' ( 42) '
```

пробелы будут игнорироваться, если соответствующее поле имеет целочисленный тип, но не текстовый.

Как было показано ранее, записывая составное значение, любой его элемент можно заключить в кавычки. Это нужно делать, если при разборе этого значения без кавычек возможна неоднозначность. Например, в кавычки нужно заключать элементы, содержащие скобки, кавычки, запятую или обратную косую черту. Чтобы включить в поле составного значения, заключенное в кавычки, такие символы, как кавычки или обратная косая черта, перед ними нужно добавить обратную косую черту. Кроме того, продублированные кавычки в значении поля, заключенного в кавычки, воспринимаются как одинарные, подобно апострофам в строках SQL. С другой стороны, можно обойтись без кавычек, защитив все символы в данных, которые могут быть восприняты как часть синтаксиса составного значения, с помощью спецпоследовательностей.

Значение `NULL` в этой записи представляется пустым местом (когда между запятыми или скобками нет никаких символов). Чтобы ввести именно пустую строку, а не `NULL`, нужно написать «`»`».

Функция вывода составного значения заключает значения полей в кавычки, если они представляют собой пустые строки, либо содержат скобки, запятые, кавычки или обратную косую черту, либо состоят из одних пробелов. В последнем случае можно обойтись без кавычек, но они добавляются для удобочитаемости. Кавычки и обратная косая черта, заключенные в значения полей, при выводе дублируются.

Примечание. Помните, что написанная SQL-команда прежде всего интерпретируется как текстовая строка, а затем как составное значение. Вследствие этого число символов обратной косой черты удваивается (если используются спецпоследовательности). Например, чтобы ввести в поле составной колонки значение типа `text` с обратной косой чертой и кавычками, команду нужно будет записать так:

```
INSERT ... VALUES ( ' ("\"\\\"") ' );
```

Сначала обработчик спецпоследовательностей удаляет один уровень обратной

косой черты, так что анализатор составного значения получает на вход («\»\\»). В свою очередь, он передает эту строку процедуре ввода значения типа `text`, где она преобразуется в «\». Если бы мы работали с типом данных, процедура ввода которого также интерпретирует обратную косую черту особым образом, например `bytea`, нам могло бы понадобиться уже восемь таких символов, чтобы сохранить этот символ в поле составного значения. Во избежание такого дублирования спецсимволов строки можно заключать в доллары (см. подраздел 2.1.2.4).

Примечание. Записывать составные значения в командах SQL часто бывает удобнее с помощью конструктора `ROW`. В `ROW` отдельные значения элементов записываются так же, как если бы они не были членами составного выражения.

6.18. Диапазонные типы

Диапазонные типы представляют диапазоны значений некоторого типа данных (он также называется подтипом диапазона). Например, диапазон типа `timestamp` может представлять временной интервал, когда зарезервирован зал заседаний. В данном случае типом данных будет `tsrange` (сокращение от «`timestamp range`»), а подтипом – `timestamp`. Подтип должен быть полностью упорядочиваемым, чтобы можно было однозначно определить, где находится значение по отношению к диапазону: внутри, до или после него.

Диапазонные типы полезны тем, что позволяют представить множество возможных значений в одной структуре данных и четко выразить такие понятия, как пересечение диапазонов. Наиболее очевидный вариант их использования – применять диапазоны даты и времени для составления расписания, но также полезными могут оказаться диапазоны цен, интервалы измерений и т.д.

6.18.1. Встроенные диапазонные типы

Синергия-БД имеет следующие встроенные диапазонные типы:

- `int4range` – диапазон подтипа `integer`;
- `int8range` – диапазон подтипа `bigint`;
- `numrange` – диапазон подтипа `numeric`;
- `tsrange` – диапазон подтипа `timestamp without time zone`;

- tstzrange – диапазон подтипа timestamp with time zone;
- daterange – диапазон подтипа date.

Помимо этого, вы можете определять собственные типы; подробнее это описано в CREATE TYPE.

6.18.2. Примеры

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');
```

```
-- Вхождение
```

```
SELECT int4range(10, 20) @> 3;
```

```
-- Перекрытие
```

```
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);
```

```
-- Получение верхней границы
```

```
SELECT upper(int8range(15, 25));
```

```
-- Вычисление пересечения
```

```
SELECT int4range(10, 20) * int4range(15, 25);
```

```
-- Является ли диапазон пустым?
```

```
SELECT isempty(numrange(1, 5));
```

6.18.3. Включение и исключение границ

Любой непустой диапазон имеет две границы, верхнюю и нижнюю, и включает все точки между этими значениями. В него также может входить точка, лежащая на границе, если диапазон включает эту границу. И наоборот, если диапазон не включает границу, считается, что точка, лежащая на этой границе, в него не входит.

В текстовой записи диапазона включение нижней границы обозначается символом «[», а исключением – символом «(». Для верхней границы включение

обозначается аналогично, символом «]», а исключение – символом «)». Подробнее это описано в подразделе 6.17.5.

Для проверки, включается ли нижняя или верхняя граница в диапазон, предназначены функции `lower_inc` и `upper_inc`, соответственно.

6.18.4. Неограниченные (бесконечные) диапазоны

Нижнюю границу диапазона можно опустить и определить тем самым диапазон, включающий все точки, лежащие ниже верхней границы. Подобным образом, если не определить верхнюю границу, в диапазон войдут все точки, лежащие выше нижней границы. Если же опущена и нижняя, и правая границы, такой диапазон будет включать все возможные значения своего подтипа.

Это равнозначно тому, что нижней границей будет считаться «минус бесконечность», а верхней – «плюс бесконечность». Но заметьте, что эти бесконечные значения не являются значениями подтипа диапазона и поэтому также не могут входить в диапазон. Как следствие, нет такого понятия, как включаемая нижняя бесконечная граница – если попытаться записать такой диапазон, она будет автоматически преобразована в исключаемую.

Кроме этого, в некоторых типах есть понятие «бесконечность», но в данном контексте оно считается просто одним из значений. Например, во временных диапазонах `[today, ]` означает то же самое, что и `[today, )`. Но диапазон `[today, infinity]` отличается от `[today, infinity)` – в последнем специальное значение `infinity` типа `timestamp` не входит в диапазон.

Проверить, определена ли верхняя или нижняя граница, можно с помощью функций `lower_inf` и `upper_inf`, соответственно.

6.18.5. Ввод/вывод диапазонов

Вводимое значение диапазона должно записываться в одной из следующих форм:

(нижняя-граница, верхняя-граница)
(нижняя-граница, верхняя-граница]
[нижняя-граница, верхняя-граница)

[нижняя-граница, верхняя-граница]

empty

Тип скобок (квадратные или круглые) определяет, включаются ли в диапазон соответствующие границы, как описано выше. Заметьте, что последняя форма содержит только слово `empty` и определяет пустой диапазон (диапазон, не содержащий точек).

Здесь нижняя-граница может быть строкой с допустимым значением подтипа или быть пустой (тогда диапазон будет без нижней границы). Аналогично, верхняя-граница может задаваться одним из значений подтипа или быть неопределенной (пустой).

Любое значение диапазона можно заключить в кавычки («»). А если значение содержит круглые или квадратные скобки, запятые, кавычки или обратную косую черту, использовать кавычки необходимо, чтобы эти символы не рассматривались как часть синтаксиса диапазона. Чтобы включить в значение диапазона, заключенное в кавычки, такие символы, как кавычки или обратная косая черта, перед ними нужно добавить обратную косую черту. Кроме того, продублированные кавычки в значении диапазона, заключенного в кавычки, воспринимаются как одинарные, подобно апострофам в строках SQL. С другой стороны, можно обойтись без кавычек, защитив все символы в данных, которые могут быть восприняты как часть синтаксиса диапазона, с помощью спецпоследовательностей. Чтобы задать в качестве границы пустую строку, нужно ввести «», так как пустая строка без кавычек будет означать отсутствие границы.

Пробельные символы до и после определения диапазона игнорируются, но когда они присутствуют внутри скобок, они воспринимаются как часть значения верхней или нижней границы. Хотя они могут также игнорироваться в зависимости от подтипа диапазона.

Примечание. Эти правила очень похожи на правила записи значений для полей составных типов. Дополнительные замечания приведены в подразделе 6.16.5.

Примеры:

-- в диапазон включается 3, не включается 7 и включаются все точки между ними

```
SELECT '[3,7) '::int4range;
```

```
-- в диапазон не включаются 3 и 7, но включаются все  
точки между ними
```

```
SELECT '(3,7) '::int4range;
```

```
-- в диапазон включается только одно значение 4
```

```
SELECT '[4,4] '::int4range;
```

```
-- диапазон не включает никаких точек (нормализация  
заменит его определение
```

```
-- на 'empty')
```

```
SELECT '[4,4) '::int4range;
```

6.18.6. Конструирование диапазонов

Для каждого диапазонного типа определена функция конструктора, имеющая то же имя, что и данный тип. Использовать этот конструктор обычно удобнее, чем записывать текстовую константу диапазона, так как это избавляет от потребности в дополнительных кавычках. Функция конструктора может принимать два или три параметра. Вариант с двумя параметрами создает диапазон в стандартной форме (нижняя граница включается, верхняя исключается), тогда как для варианта с тремя параметрами включение границ определяется третьим параметром. Третий параметр должен содержать одну из строк: «()», «()», «[]» или «[]», например:

```
-- Полная форма: нижняя граница, верхняя граница и  
текстовая строка, определяющая
```

```
-- включение/исключение границ.
```

```
SELECT numrange(1.0, 14.0, '[]');
```

```
-- Если третий аргумент опущен, подразумевается '[]'.
```

```
SELECT numrange(1.0, 14.0);
```

```
-- Хотя здесь указывается '[]', при выводе значение  
будет приведено к
```

```
-- каноническому виду, так как int8range - тип
```

дискретного диапазона (см. ниже).

```
SELECT int8range(1, 14, '()');
```

```
-- Когда вместо любой границы указывается NULL,  
соответствующей границы
```

```
-- у диапазона не будет.
```

```
SELECT numrange(NULL, 2.2);
```

6.18.7. Типы дискретных диапазонов

Дискретным диапазоном считается диапазон, для подтипа которого однозначно определен «шаг», как например для типов `integer` и `date`. Значения этих двух типов можно назвать соседними, когда между ними нет никаких других значений. В непрерывных диапазонах, напротив, всегда (или почти всегда) можно найти ещё одно значение между двумя данными. Например, непрерывным диапазоном будет диапазон с подтипами `numeric` и `timestamp`. Хотя `timestamp` имеет ограниченную точность, то есть теоретически он является дискретным, но все же лучше считать его непрерывным, так как шаг его обычно не определен.

Можно также считать дискретным подтип диапазона, в котором четко определены понятия «следующего» и «предыдущего» элемента для каждого значения. Такие определения позволяют преобразовывать границы диапазона из включаемых в исключаемые, выбирая следующий или предыдущий элемент вместо заданного значения. Например, диапазоны целочисленного типа `[4,8]` и `(3,9)` описывают одно и то же множество значений; но для диапазона подтипа `numeric` это не так.

Для типа дискретного диапазона определяется функция канонизации, учитывающая размер шага для данного подтипа. Задача этой функции — преобразовать равнозначные диапазоны к единственному представлению, в частности нормализовать включаемые и исключаемые границы. Если функция канонизации не определена, диапазоны с различным определением будут всегда считаться разными, даже когда они на самом деле представляют одно множество значений.

Для встроенных типов `int4range`, `int8range` и `daterange` каноническое

представление включает нижнюю границу и не включает верхнюю; то есть диапазон приводится к виду `]`. Однако для нестандартных типов можно использовать и другие соглашения.

6.18.8. Определение новых диапазонных типов

Пользователи могут определять собственные диапазонные типы. Это может быть полезно, когда нужно использовать диапазоны с подтипами, для которых нет встроенных диапазонных типов. Например, можно определить новый тип диапазона для подтипа `float8`:

```
CREATE TYPE floatrange AS RANGE (
    subtype = float8,
    subtype_diff = float8mi
);
SELECT '[1.234, 5.678]'::floatrange;
```

Так как для `float8` осмысленное значение «шага» не определено, функция канонизации в данном примере не задается.

Определяя собственный диапазонный тип, вы также можете выбрать другие правила сортировки или класс оператора В-дерева для его подтипа, что позволит изменить порядок значений, от которого зависит, какие значения попадают в заданный диапазон.

Если подтип можно рассматривать как дискретный, а не непрерывный, в команде `CREATE TYPE` следует также задать функцию канонизации. Этой функции будет передаваться значение диапазона, а она должна вернуть равнозначное значение, но, возможно, с другими границами и форматированием. Для двух диапазонов, представляющих одно множество значений, например, целочисленные диапазоны `[1, 7]` и `[1, 8)`, функция канонизации должна выдавать один результат. Какое именно представление будет считаться каноническим, не имеет значения — главное, чтобы два равнозначных диапазона, отформатированных по-разному, всегда преобразовывались в одно значение с одинаковым форматированием. Помимо исправления формата включаемых/исключаемых границ, функция канонизации может округлять значения границ, если размер шага превышает точность хранения

подтипа. Например, в типе диапазона для подтипа `timestamp` можно определить размер шага, равный часу, тогда функция канонизации должна будет округлить границы, заданные, например с точностью до минут, либо вместо этого выдать ошибку.

Помимо этого, для любого диапазонного типа, ориентированного на использование с индексами GiST или SP-GiST, должна быть определена разница значений подтипов, функция `subtype_diff`. Индекс сможет работать и без `subtype_diff`, но в большинстве случаев это будет не так эффективно. Эта функция принимает на вход два значения подтипа и возвращает их разницу (т. е. X минус Y) в значении типа `float8`. В показанном выше примере может использоваться функция `float8mi`, определяющая нижележащую реализацию обычного оператора «минус» для типа `float8`, но для другого подтипа могут потребоваться дополнительные преобразования. Иногда для представления разницы в числовом виде требуется ещё и творческий подход. Функция `subtype_diff`, насколько это возможно, должна быть согласована с порядком сортировки, вытекающим из выбранных правил сортировки и класса оператора; то есть, её результат должен быть положительным, если согласно порядку сортировки первый её аргумент больше второго.

Ещё один, не столь тривиальный пример функции `subtype_diff`:

```
CREATE FUNCTION time_subtype_diff(x time, y time)
RETURNS float8 AS
'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT
IMMUTABLE;
```

```
CREATE TYPE timerange AS RANGE (
    subtype = time,
    subtype_diff = time_subtype_diff
);
```

```
SELECT '[11:10, 23:00]':::timerange;
```

6.18.9. Индексация

Для колонок, имеющих диапазонный тип, можно создать индексы GiST и SP-GiST. Например, так создается индекс GiST:

```
CREATE INDEX reservation_idx ON reservation USING gist
(during);
```

Индекс GiST или SP-GiST помогает ускорить запросы со следующими операторами: =, &&, <@, @>, <<, >>, -|-, &< и &> (дополнительно о них можно узнать в таблице 7-47).

Кроме того, для таких колонок можно создать индексы на основе хеша и В-деревьев. Для индексов таких типов полезен по сути только один оператор диапазона – равно. Порядок сортировки В-дерева определяется для значений диапазона соответствующими операторами < и >, но этот порядок может быть произвольным и он не очень важен в реальном мире. Поддержка В-деревьев и хешей диапазонными типами нужна в основном для сортировки и хеширования при выполнении запросов, но не для создания самих индексов.

6.18.10. Ограничения для диапазонов

Тогда как для скалярных значений естественным ограничением является UNIQUE, оно обычно не подходит для диапазонных типов. Вместо этого чаще оказываются полезнее ограничения-исключения. Такие ограничения позволяют, например определить условие «непересечения» диапазонов, например:

```
CREATE TABLE reservation (
    during tsrange,
    EXCLUDE USING gist (during WITH &&)
);
```

Это ограничение не позволит одновременно сохранить в таблице несколько диапазонов, которые накладываются друг на друга:

```
INSERT INTO reservation VALUES
(' [2010-01-01 11:30, 2010-01-01 15:00) ');
INSERT 0 1
```



```
INSERT INTO reservation VALUES
```

```
(' [2010-01-01 14:45, 2010-01-01 15:45)');
```

ОШИБКА: конфликтующее значение ключа нарушает ограничение-исключение «reservation_during_excl»

ПОДРОБНОСТИ: Ключ (during)=([«2010-01-01 14:45:00», »2010-01-01 15:45:00»))

конфликтует с существующим ключом (during)=([«2010-01-01 11:30:00», »2010-01-01 15:00:00»))

Для максимальной гибкости в ограничении-исключении можно сочетать простые скалярные типы данных с диапазонами, используя расширение btree_gist. Например, если btree_gist установлено, следующее ограничение не будет допускать пересекающиеся диапазоны, только если совпадают также и номера комнат:

```
CREATE EXTENSION btree_gist;
```

```
CREATE TABLE room_reservation (
```

```
    room text,
```

```
    during tsrange,
```

```
    EXCLUDE USING gist (room WITH =, during WITH &&)
```

```
);
```

```
INSERT INTO room_reservation VALUES
```

```
('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');
```

```
INSERT 0 1
```

```
INSERT INTO room_reservation VALUES
```

```
('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');
```

ОШИБКА: конфликтующее значение ключа нарушает ограничение-исключение

«room_reservation_room_during_excl»

ПОДРОБНОСТИ: Ключ (room, during)=(123A, [2010-01-01 14:30:00,

2010-01-01 15:30:00)) конфликтует

с существующим ключом (room, during)=(123A, [«2010-01-01 14:00:00», »2010-01-01 15:00:00»)).

```
INSERT INTO room_reservation VALUES  
    ('123B', '[2010-01-01 14:30, 2010-01-01 15:30)');  
INSERT 0 1
```

6.19. Типы доменов

Домен – пользовательский тип данных, основанный на другом нижележащем типе. Он может быть определён с условиями, ограничивающими множество допустимых значений подмножеством значений нижележащего типа. В остальном он ведёт себя как нижележащий тип – например, с доменными типом будут работать любые операторы или функции, применимые к нижележащему типу. Нижележащим типом может быть любой встроенный или пользовательский базовый тип, тип-перечисление, массив, составной тип, диапазон или другой домен.

Например, мы можем создать домен поверх целых чисел, принимающий только положительные числа:

```
CREATE DOMAIN posint AS integer CHECK (VALUE > 0);  
CREATE TABLE mytable (id posint);  
INSERT INTO mytable VALUES(1);    -- работает  
INSERT INTO mytable VALUES(-1);  -- ошибка
```

Когда к значению доменного типа применяются операторы или функции, предназначенные для нижележащего типа, домен автоматически приводится к нижележащему типу. Так, например, результат операции `mytable.id - 1` будет считаться имеющим тип `integer`, а не `posint`. Мы могли бы записать `(mytable.id - 1)::posint`, чтобы снова привести результат к типу `posint`, что повлечёт перепроверку ограничений домена. В этом случае, если данное выражение будет применено к `id`, равному 1, произойдёт ошибка. Значение нижележащего типа можно присвоить полю или переменной доменного типа, не записывая приведение явно, но и в этом случае ограничения домена будут проверяться.

6.20. Идентификаторы объектов

Идентификатор объекта (Object Identifier, OID) используется внутри

Синергия-БД в качестве первичного ключа различных системных таблиц. В пользовательские таблицы колонка OID добавляется, только если при создании таблицы указывается WITH OIDS или включен параметр конфигурации `default_with_oids`. Идентификатор объекта представляется в типе `oid`. Также для типа `oid` определены следующие псевдонимы: `regproc`, `regprocedure`, `regoper`, `regoperator`, `regclass`, `regtype`, `regrole`, `regnamespace`, `regconfig` и `regdictionary`. Обзор этих типов приведен в таблице 6.24.

В настоящее время тип `oid` реализован как четырехбайтное целое. Таким образом оно может быть недостаточно большим для обеспечения уникальности в базе данных или даже в отдельных больших таблицах. Поэтому в пользовательских таблицах использовать колонку типа OID в качестве первичного ключа не рекомендуется. Лучше всего ограничить применение этого типа обращениями к системным таблицам.

Для самого типа `oid` помимо сравнения определены всего несколько операторов. Однако его можно привести к целому и затем задействовать в обычных целочисленных вычислениях. При этом следует опасаться путаницы со знаковыми/беззнаковыми значениями.

Типы-псевдонимы OID сами по себе не вводят новых операций и отличаются только специализированными функциями ввода/вывода. Эти функции могут принимать и выводить не просто числовые значения, как тип `oid`, а символические имена системных объектов. Эти типы позволяют упростить поиск объектов по значениям OID. Например, чтобы выбрать из `pg_attribute` строки, относящиеся к таблице `mytable`, можно написать:

```
SELECT * FROM pg_attribute WHERE attrelid =
'mytable'::regclass;
```

вместо:

```
SELECT * FROM pg_attribute
WHERE attrelid = (SELECT oid FROM pg_class WHERE
relname = 'mytable');
```

Хотя второй вариант выглядит не таким уж плохим, но это лишь очень простой запрос. Если же потребуется выбрать правильный OID, когда таблица `mytable` есть

в нескольких схемах, вложенный подзапрос будет гораздо сложнее. Преобразователь вводимого значения типа `regclass` находит таблицу согласно заданному пути поиска схем, так что он делает «все правильно» автоматически. Аналогично, приведя идентификатор таблицы к типу `regclass`, можно получить символическое представление числового кода.

Таблица 6.24 – Идентификаторы объектов

Имя	Ссылки	Описание	Пример значения
<code>oid</code>	<code>any</code>	числовой идентификатор объекта	564182
<code>regproc</code>	<code>pg_proc</code>	имя функции	<code>sum</code>
<code>regprocedure</code>	<code>pg_proc</code>	функция с типами аргументов	<code>sum(int4)</code>
<code>regoper</code>	<code>pg_operator</code>	имя оператора	<code>+</code>
<code>regoperator</code>	<code>pg_operator</code>	оператор с типами аргументов	<code>*(integer, integer)</code> или <code>-(NONE, integer)</code>
<code>regclass</code>	<code>pg_class</code>	имя отношения	<code>pg_type</code>
<code>regtype</code>	<code>pg_type</code>	имя типа данных	<code>integer</code>
<code>regrole</code>	<code>pg_authid</code>	имя роли	<code>smith</code>
<code>regnamespace</code>	<code>pg_namespac e</code>	пространство имён	<code>pg_catalog</code>
<code>regconfig</code>	<code>pg_ts_confi g</code>	конфигурация текстового поиска	<code>english</code>
<code>regdictionary</code>	<code>pg_ts_dict</code>	словарь текстового поиска	<code>simple</code>

Все типы псевдонимов `OID` для объектов, сгруппированных в пространство имён принимают имена, дополненные именем схемы, и выводят имена со схемой, если данный объект нельзя будет найти в текущем пути поиска без имени схемы. Типы `regproc` и `regoper` принимают только уникальные вводимые имена (не перегруженные), что ограничивает их применимость; в большинстве случаев лучше использовать `regprocedure` или `regoperator`. Для типа `regoperator` в записи унарного оператора неиспользуемый операнд заменяется словом `NONE`.

Дополнительным свойством типов псевдонимов `OID` является образование зависимостей. Когда в сохраненном выражении фигурирует константа одного из этих типов (например, в представлении или в значении колонки по умолчанию), это создает зависимость от целевого объекта. Например, если значение по умолчанию определяется выражением `nextval('my_seq'::regclass)`, Синергия-БД понимает, что это выражение зависит от последовательности `my_seq`, и не позволит удалить последовательность раньше, чем будет удалено это выражение. Единственным ограничением является тип `regrole`. Константы этого типа в таких

выражениях не допускаются.

Примечание. Типы псевдонимов `OID` не полностью следуют правилам изоляции транзакций. Планировщик тоже воспринимает их как простые константы, что может привести к неоптимальному планированию запросов.

Есть ещё один тип системных идентификаторов, `xid`, представляющий идентификатор транзакции (сокращенно `xact`). Этот тип имеют системные колонки `xmin` и `xmax`. Идентификаторы транзакций определяются 32-битными числами.

Третий тип идентификаторов, используемых в системе, – `cid`, идентификатор команды (`command identifier`). Этот тип данных имеют системные колонки `ctid` и `stid`. Идентификаторы команд – это тоже 32-битные числа.

И наконец, последний тип системных идентификаторов – `tid`, идентификатор строки/кортежа (`tuple identifier`). Этот тип данных имеет системная колонка `ctid`. Идентификатор кортежа представляет собой пару (из номера блока и индекса кортежа в блоке), идентифицирующую физическое расположение строки в таблице.

Подробнее о системных колонках рассказывается в разделе 3.4.

6.21. Тип `pg_lsn`

Тип данных `pg_lsn` может применяться для хранения значения LSN (последовательный номер в журнале, Log Sequence Number), которое представляет собой указатель на позицию в журнале WAL. Этот тип содержит `XLogRecPtr` и является внутренним системным типом Синергия-БД.

Примечание. Технически LSN – это 64-битное целое, представляющее байтовое смещение в потоке журнала предзаписи. Он выводится в виде двух шестнадцатеричных чисел до 8 цифр каждое, через косую черту, например: 16/B374D848. Тип `pg_lsn` поддерживает стандартные операторы сравнения, такие как `=` и `>`. Можно также вычесть один LSN из другого с помощью оператора `-`; результатом будет число байт между этими двумя позициями в журнале предзаписи.

6.22. Псевдотипы

В систему типов Синергия-БД включены несколько специальных элементов, которые в совокупности называются псевдотипами. Псевдотип нельзя использовать в качестве типа данных колонки, но можно объявить функцию с аргументом или результатом такого типа. Каждый из существующих псевдотипов полезен в

ситуациях, когда характер функции не позволяет просто получить или вернуть определенный тип данных SQL. Все существующие псевдотипы перечислены в таблице 6.25.

Таблица 6.25 – Псевдотипы

Имя	Описание
<code>any</code>	Указывает, что функция принимает любой вводимый тип данных.
<code>anyelement</code>	Указывает, что функция принимает любой тип данных.
<code>anyarray</code>	Указывает, что функция принимает любой тип массива.
<code>anynonarray</code>	Указывает, что функция принимает любой тип данных, кроме массивов.
<code>anyenum</code>	Указывает, что функция принимает любое перечисление (см. раздел 6.7).
<code>anyrange</code>	Указывает, что функция принимает любой диапазонный тип данных (см. раздел 6.17).
<code>cstring</code>	Указывает, что функция принимает или возвращает строку в стиле C.
<code>internal</code>	Указывает, что функция принимает или возвращает внутренний серверный тип данных.
<code>language_handler</code>	Обработчик процедурного языка объявляется как возвращающий тип <code>language_handler</code> .
<code>fdw_handler</code>	Обработчик обертки сторонних данных объявляется как возвращающий тип <code>fdw_handler</code> .
<code>index_am_handler</code>	Обработчик метода доступа индекса объявляется как возвращающий тип <code>index_am_handler</code> .
<code>tsm_handler</code>	Обработчик метода выборки из таблицы объявляется как возвращающий тип <code>tsm_handler</code> .
<code>record</code>	Указывает функцию, возвращающую неопределенный тип строки.
<code>trigger</code>	Триггерная функция объявляется как возвращающая тип <code>trigger</code> .
<code>event_trigger</code>	Функция событийного триггера объявляется как возвращающая тип <code>event_trigger</code> .
<code>pg_ddl_command</code>	Обозначает представление команд DDL, доступное событийным триггерам.
<code>event_trigger</code>	Функция событийного триггера объявляется как возвращающая тип <code>event_trigger</code> .
<code>pg_ddl_command</code>	Обозначает представление команд DDL, доступное событийным триггерам.
<code>opaque</code>	Устаревший тип, который раньше использовался во многих вышеперечисленных случаях.

Функции, написанные на языке C (встроенные или динамически загружаемые), могут быть объявлены с параметрами или результатами любого из этих типов. Ответственность за безопасное поведение функции с аргументами таких типов ложится на разработчика функции.

Функции, написанные на процедурных языках, могут использовать псевдотипы, только если это позволяет соответствующий язык. В настоящее время большинство процедурных языков запрещают использовать псевдотипы в качестве типа аргумента и позволяют использовать для результатов только типы `void` и `record` (и `trigger` или `event_trigger`, когда функция реализует триггер или событийный триггер). Некоторые языки также поддерживают полиморфные функции с типами `anyelement`, `anyarray`, `anynonarray`, `anyenum` и `anyrange`.

Псевдотип `internal` используется в объявлениях функций, предназначенных только для внутреннего использования в СУБД, но не для прямого вызова в запросах

SQL. Если у функции есть как хотя бы один аргумент типа `internal`, её нельзя будет вызывать из SQL. Чтобы сохранить типобезопасность при таком ограничении, следуйте важному правилу: не создавайте функцию, возвращающую результат типа `internal`, если у нее нет ни одного аргумента `internal`.

Поэтому понятие «значение» включает и элементы массивов, хотя в терминологии JSON иногда элементы массивов считаются отличными от значений внутри объектов.

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

SQL	-	Structured Query Language (язык структурированных запросов)
WAL	-	Write-Ahead Log (журнал упреждающей записи)
БД	-	база данных
КСЗ	-	комплекс средств защиты
ОЗУ	-	оперативное запоминающее устройство
ОС	-	операционная система
ПРД	-	правила разграничения доступа
СУБД	-	система управления базами данных

[illegible][illegible]

Федеральное государственное унитарное предприятие
Российский федеральный ядерный центр
Всероссийский научно-исследовательский институт экспериментальной физики

УТВЕРЖДЕН
07623615.00082-02 34 01-ЛУ

КОМПЛЕКС ПРОГРАММ В ЗАЩИЩЕННОМ ИСПОЛНЕНИИ
«СИСТЕМА ПОЛНОГО ЖИЗНЕННОГО ЦИКЛА ИЗДЕЛИЙ
«ЦИФРОВОЕ ПРЕДПРИЯТИЕ»

**Программный модуль
«Система управления базами данных «Синергия-БД»**

Руководство оператора

Часть 2

07623615.00082-02 34 01-2

Листов 286

Име. № подл.	Подп. и дата	Взам. инв. №	Име. № дубл.	Подп. и дата

СОДЕРЖАНИЕ

1. Функции и операторы	5
1.1. Логические операторы	5
1.2. Функции и операторы сравнения	6
1.3. Математические функции и операторы	9
1.4. Строковые функции и операторы	12
1.5. Функция форматирования	22
1.6. Функции и операторы двоичных строк.....	26
1.7. Функции и операторы для работы с битовыми строками	28
1.8. Поиск по шаблону	29
1.9. Функции форматирования данных	51
1.10. Операторы и функции даты/времени	58
1.11. Функции для перечислений.....	75
1.12. Геометрические функции и операторы	76
1.13. Функции и операторы для работы с сетевыми адресами.....	79
1.14. Функции и операторы текстового поиска	81
1.15. XML-функции.....	86
1.16. Функции и операторы JSON	106
1.17. Функции для работы с последовательностями	113
1.18. Условные выражения	116
1.19. Функции и операторы для работы с массивами.....	120
1.20. Диапазонные функции и операторы	123
1.21. Агрегатные функции	125
1.22. Оконные функции.....	133
1.23. Выражения подзапросов.....	135
1.24. Сравнение табличных строк и массивов	140
1.25. Функции, возвращающие множества.....	144
1.26. Системные информационные функции	149
1.27. Функции для системного администрирования	170
1.28. Триггерные функции.....	191

1.29. Функции событийных триггеров	192
2. Команды SQL	195
2.1. ABORT	195
2.2. ALTER AGGREGATE.....	196
2.3. ALTER COLLATION	198
2.4. ALTER CONVERSION.....	200
2.5. ALTER DATABASE	201
2.6. ALTER DEFAULT PRIVILEGES	204
2.7. ALTER DOMAIN	208
2.8. ALTER EVENT TRIGGER.....	212
2.9. ALTER EXTENSION	213
2.10. ALTER FOREIGN DATA WRAPPER.....	217
2.11. ALTER FOREIGN TABLE	219
2.12. ALTER FUNCTION	226
2.13. ALTER GROUP	231
2.14. ALTER INDEX	231
2.15. ALTER LANGUAGE	235
2.16. ALTER LARGE OBJECT	236
2.17. ALTER MATERIALIZED VIEW.....	236
2.18. ALTER OPERATOR.....	239
2.19. ALTER OPERATOR CLASS	241
2.20. ALTER OPERATOR FAMILY	242
2.21. ALTER POLICY	247
2.22. ALTER PROCEDURE.....	248
2.23. ALTER PUBLICATION	251
2.24. ALTER ROLE	253
2.25. ALTER ROUTINE	258
2.26. ALTER RULE	260
2.27. ALTER SCHEMA.....	260
2.28. ALTER SEQUENCE	261
2.29. ALTER SERVER	266

2.30. ALTER SYSTEM.....	267
2.31. ALTER TABLE	269
Перечень сокращений	285

1. ФУНКЦИИ И ОПЕРАТОРЫ

Синергия-БД предоставляет огромное количество функций и операторов для встроенных типов данных. Кроме того, пользователи могут определять свои функции операторы. Просмотреть все существующие функции и операторы можно в `psql` с помощью команд `\df` и `\do`, соответственно.

Если для вас важна переносимость, учтите, что практически все функции и операторы, описанные в этой главе, за исключением простейших арифметических и операторов сравнения, а также явно отмеченных функций, не описаны в стандарте SQL. Тем не менее, частично эта расширенная функциональность присутствует и в других СУБД SQL и во многих случаях различные реализации одинаковых функций оказываются аналогичными и совместимыми. В этой главе не описываются абсолютно все функции; некоторые дополнительные функции рассматриваются в других разделах документации.

1.1. Логические операторы

Набор логических операторов включает обычные:

- AND;
- OR;
- NOT.

В SQL работает логическая система с тремя состояниями: `true` (истина), `false` (ложь) и `NULL`, «неопределенное» состояние. Рассмотрите следующие параметры истинности:

a	b	a AND b	a OR b
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	NOT a
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Операторы AND и OR коммутативны, то есть от перемены мест операндов результат не меняется. Однако значение может иметь порядок вычисления

подвыражений. Подробнее это описано в подразделе 2.2.14 (Часть 1. Руководство оператора).

1.2. Функции и операторы сравнения

Набор операторов сравнения включает обычные операторы, перечисленные в таблице 1.1.

Таблица 1.1 – Операторы сравнения

Оператор	Описание
<	меньше
>	больше
<=	меньше или равно
>=	больше или равно
=	равно
<> или !=	не равно

Примечание. Оператор != преобразуется в <> на стадии разбора запроса. Как следствие, реализовать операторы != и <> по-разному невозможно.

Операторы сравнения определены для всех типов данных, для которых они имеют смысл. Все операторы сравнения представляют собой бинарные операторы, возвращающие значения типа boolean; при этом выражения вида $1 < 2 < 3$ недопустимы (так как не существует оператора <, который бы сравнивал булево значение с 3).

Существует также несколько предикатов сравнения; они приведены в таблице 1.2. Они работают подобно операторам, но имеют особый синтаксис, установленный стандартом SQL.

Таблица 1.2 – Предикаты сравнения

Предикат	Описание
a BETWEEN x AND y	между
a NOT BETWEEN x AND y	не между
a BETWEEN SYMMETRIC x AND y	между, после сортировки сравниваемых значений
a NOT BETWEEN SYMMETRIC x AND y	не между, после сортировки сравниваемых значений
a IS DISTINCT FROM b	не равно, при этом NULL воспринимается как обычное значение
a IS NOT DISTINCT FROM b	равно, при этом NULL воспринимается как обычное значение
выражение IS NULL	эквивалентно NULL
выражение IS NOT NULL	не эквивалентно NULL
выражение ISNULL	эквивалентно NULL (нестандартный синтаксис)
выражение NOTNULL	не эквивалентно NULL (нестандартный синтаксис)
логическое_выражение IS TRUE	истина
логическое_выражение IS NOT TRUE	ложь или неопределённость

Окончание таблицы 1.2

Предикат	Описание
логическое_выражение IS NOT FALSE	истина или неопределённость
логическое_выражение IS UNKNOWN	неопределённость
логическое_выражение IS NOT UNKNOWN	истина или ложь

Предикат BETWEEN упрощает проверки интервала:

$a \text{ BETWEEN } x \text{ AND } y$

равнозначно выражению:

$a \geq x \text{ AND } a \leq y$

Заметьте, что BETWEEN считает, что границы интервала также включаются в интервал. NOT BETWEEN выполняет противоположное сравнение:

$a \text{ NOT BETWEEN } x \text{ AND } y$

равнозначно выражению:

$a < x \text{ OR } a > y$

Предикат BETWEEN SYMMETRIC аналогична BETWEEN, за исключением того, что аргумент слева от AND не обязательно должен быть меньше или равен аргументу справа. Если это не так, аргументы автоматически меняются местами, так что всегда подразумевается непустой интервал.

Обычные операторы сравнения выдают NULL (что означает «неопределенность»), а не true или false, когда любое из сравниваемых значений NULL. Например, $7 = \text{NULL}$ выдает NULL, так же, как и $7 <> \text{NULL}$. Когда это поведение нежелательно, можно использовать конструкции IS [NOT] DISTINCT FROM:

$a \text{ IS DISTINCT FROM } b$

$a \text{ IS NOT DISTINCT FROM } b$

Для значений не NULL условие IS DISTINCT FROM работает так же, как оператор $\neq$. Однако, если оба сравниваемых значения NULL, результат будет false, и только если одно из значений NULL, возвращается true. Аналогично, условие IS NOT DISTINCT FROM равносильно $=$ для значений не NULL, но возвращает true, если оба сравниваемых значения NULL и false в противном случае. Таким образом, эти конструкции по сути работают с NULL, как с обычным

значением, а не с «неопределенностью».

Для проверки, содержит ли значение NULL или нет, используются предикаты:

выражение IS NULL

выражение IS NOT NULL

или равнозначные (но нестандартные) предикаты:

выражение ISNULL

выражение NOTNULL

Заметьте, что проверка *выражение* = NULL не будет работать, так как NULL считается не «равным» NULL. Значение NULL представляет неопределённость, и равны ли две неопределённости, тоже не определено.

Примечание. Некоторые приложения могут ожидать, что *выражение* = NULL вернёт true, если результатом выражения является NULL. Такие приложения настоятельно рекомендуется исправить и привести в соответствие со стандартом SQL. Однако, в случаях, когда это невозможно, это поведение можно изменить с помощью параметра конфигурации `transform_null_equals`. Когда этот параметр включён, Синергия-БД преобразует условие `x = NULL` в `x IS NULL`.

Если выражение возвращает табличную строку, тогда IS NULL будет истинным, когда само выражение – NULL или все поля строки – NULL, а IS NOT NULL будет истинным, когда само выражение не NULL, и все поля строки так же не NULL. Вследствие такого определения, IS NULL и IS NOT NULL не всегда будут возвращать взаимодополняющие результаты для таких выражений; в частности такие выражения со строками, одни поля которых NULL, а другие не NULL, будут ложными одновременно. В некоторых случаях имеет смысл написать строка IS DISTINCT FROM NULL или строка IS NOT DISTINCT FROM NULL, чтобы просто проверить, равно ли NULL всё значение строки, без каких-либо дополнительных проверок полей строки.

Логические значения можно также проверить с помощью условий:

логическое выражение IS TRUE

логическое выражение IS NOT TRUE

логическое выражение IS FALSE

логическое выражение IS NOT FALSE

логическое выражение IS UNKNOWN

логическое выражение IS NOT UNKNOWN

Они всегда возвращают true или false и никогда NULL, даже если какой-либо операнд NULL. Они интерпретируют значение NULL как «неопределённость».. Заметьте, что IS UNKNOWN и IS NOT UNKNOWN по сути равнозначны IS NULL и IS NOT NULL, соответственно, за исключением того, что выражение может быть только булевого типа.

Также имеется несколько связанных со сравнениями функций; они перечислены в таблице 1.3.

Таблица 1.3 – Функции сравнения

Функция	Описание	Пример	Результат примера
num_nonnulls (VARIADIC "any")	возвращает число аргументов, отличных от NULL	num_nonnulls(1, NULL, 2)	2
num_nulls (VARIADIC "any")	возвращает число аргументов NULL	num_nulls(1, NULL, 2)	1

1.3. Математические функции и операторы

Математические операторы определены для множества типов Синергия-БД. Как работают эти операции с типами, для которых нет стандартных соглашений о математических действиях (например, с типами даты/времени), мы опишем в последующих разделах.

В таблице 1.4 перечислены все доступные математические операторы.

Таблица 1.4 – Математические операторы

Оператор	Описание	Пример	Результат
+	сложение	2 + 3	5
-	вычитание	2 - 3	-1
*	умножение	2 * 3	6
/	деление (при целочисленном делении остаток отбрасывается)	4 / 2	2
%	остаток от деления	5 % 4	1
^	возведение в степень (вычисляется слева направо)	2.0 ^ 3.0	8
/	квадратный корень	/ 25.0	5
/	кубический корень	/ 27.0	3
!	факториал	5 !	120
!!	факториал (префиксная форма)	!! 5	120
@	модуль числа (абсолютное значение)	@ -5.0	5
&	битовый AND	91 & 15	11
	битовый OR	32 3	35
#	битовый XOR	17 # 5	20
~	битовый NOT	~1	-2
<<	битовый сдвиг влево	1 << 4	16
>>	битовый сдвиг вправо	8 >> 2	2

Битовые операторы работают только с целостными типами данных, тогда как другие работают и с остальными числовыми типами. Битовые операции также

работают с битовыми строками bit и bit varying.

В таблице 1.5 перечислены все существующие математические функции. Сокращение dp в ней обозначает тип double precision (плавающее с двойной точностью). Многие из этих функций имеют несколько форм с разными типами аргументов. За исключением случаев, где это указано явно, любая форма функции возвращает результат того же типа, что и аргумент. Функции, работающие с данными double precision, в массе своей используют реализации из системных библиотек сервера, поэтому точность и поведение в граничных случаях может зависеть от системы сервера.

Таблица 1.5 – Математические функции

Функция	Тип результата	Описание	Пример	Результат
abs(x)	тип аргумента	модуль числа (абсолютное значение)	abs(-17.4)	17.4
cbrt(dp)	dp	кубический корень	cbrt(27.0)	3
ceil(dp или numeric)	тип аргумента	ближайшее целое, большее или равное аргументу	ceil(-42.8)	-42
ceiling(dp или numeric)	тип аргумента	ближайшее целое, большее или равное аргументу (равнозначно ceil)	ceiling(-95.3)	-95
degrees(dp)	dp	преобразование радианов в градусы	degrees(0.5)	28.6478897565412
div(y numeric, x numeric)	numeric	целочисленный результат y/x	div(9,4)	2
exp(dp или numeric)	тип аргумента	экспонента	exp(1.0)	2.71828182845905
floor(dp или numeric)	тип аргумента	ближайшее целое, меньшее или равное аргументу	floor(-42.8)	-43
ln(dp или numeric)	тип аргумента	натуральный логарифм	ln(2.0)	0.693147180559945
log(dp или numeric)	тип аргумента	логарифм по основанию 10	log(100.0)	2
log(b numeric, x numeric)	numeric	логарифм по основанию b	log(2.0, 64.0)	6.0000000000
mod(y, x)	зависит от типов аргументов	остаток от деления y/x	mod(9,4)	1
pi()	dp	константа «π»	pi()	3.14159265358979
power(a dp, b dp)	dp	a возводится в степень b	power(9.0, 3.0)	729
power(a numeric, b numeric)	numeric	a возводится в степень b	power(9.0, 3.0)	729
radians(dp)	dp	преобразование градусов в радианы	radians(45.0)	0.785398163397448
round(dp или numeric)	тип аргумента	округление до ближайшего целого	round(42.4)	42
round(v numeric, s int)	numeric	округление v до s десятичных знаков	round(42.4382, 2)	42.44

Окончание таблицы 1.5

Функция	Тип результата	Описание	Пример	Результат
<code>scale(numeric)</code>	<code>integer</code>	масштаб аргумента (число десятичных цифр в дробной части)	<code>scale(8.41)</code>	2
<code>sign(dp или numeric)</code>	тип аргумента	знак аргумента (-1, 0, +1)	<code>sign(-8.4)</code>	-1
<code>sqrt(dp или numeric)</code>	тип аргумента	квадратный корень	<code>sqrt(2.0)</code>	1.4142135623731
<code>trunc(dp или numeric)</code>	тип аргумента	округление к нулю	<code>trunc(42.8)</code>	42
<code>trunc(v numeric, s int)</code>	<code>numeric</code>	округление к 0 до s десятичных знаков	<code>trunc(42.4382, 2)</code>	42.43
<code>width_bucket(operand dp, b1 dp, b2 dp, count int)</code>	<code>int</code>	возвращает номер группы, в которую попадет operand в гистограмме с числом групп count равного размера, в диапазоне от b1 до b2; возвращает 0 или count+1, если операнд лежит вне диапазона	<code>width_bucket(5.35, 0.024, 10.06, 5)</code>	3
<code>width_bucket(operand numeric, b1 numeric, b2 numeric, count int)</code>	<code>int</code>	возвращает номер группы, в которую попадет operand в гистограмме с числом групп count равного размера, в диапазоне от b1 до b2; возвращает 0 или count+1, если операнд лежит вне диапазона	<code>width_bucket(5.35, 0.024, 10.06, 5)</code>	3
<code>width_bucket(operand anyelement, thresholds anyarray)</code>	<code>int</code>	возвращает номер группы, в которую попадет operand (группы определяются нижними границами, передаваемыми в thresholds); возвращает 0, если операнд оказывается левее нижней границы; массив thresholds должен быть отсортирован по возрастанию, иначе будут получены неожиданные результаты	<code>width_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestamp_tz[])</code>	2

В таблице 1.6 перечислены все функции для генерации случайных чисел.

Таблица 1.6 – Случайные функции

Функция	Тип результата	Описание
<code>random()</code>	<code>dp</code>	случайное число в диапазоне $0.0 \leq x < 1.0$
<code>setseed(dp)</code>	<code>void</code>	задает отправную точку для последующих вызовов <code>random()</code> (значение между -1.0 и 1.0, включая границы)

Характеристики значений, возвращаемых функцией `random()` зависят от

системы. Для применения в криптографии они непригодны; альтернативы описаны в pgcrypto.

Наконец, в таблице 1.7 перечислены все тригонометрические функции. Все эти функции принимают аргументы и возвращают значения типа `double precision`. У каждой функции имеются две вариации – одна измеряет углы в радианах, а вторая в градусах.

Таблица 1.7 – Тригонометрические функции

Функции (в радианах)	Функции (в градусах)	Описание
<code>acos(x)</code>	<code>acosd(x)</code>	арккосинус
<code>asin(x)</code>	<code>asind(x)</code>	арксинус
<code>atan(x)</code>	<code>atand(x)</code>	арктангенс
<code>atan2(y, x)</code>	<code>atan2d(y, x)</code>	арктангенс y/x
<code>cos(x)</code>	<code>cosd(x)</code>	косинус
<code>cot(x)</code>	<code>cotd(x)</code>	котангенс
<code>sin(x)</code>	<code>sind(x)</code>	синус
<code>tan(x)</code>	<code>tand(x)</code>	тангенс

1.4. Строковые функции и операторы

В этом разделе описаны функции и операторы для работы с текстовыми строками. Под строками в данном контексте подразумеваются значения типов `character`, `character varying` и `text`. Если не отмечено обратное, все нижеперечисленные функции работают со всеми этими типами, хотя с типом `character` следует учитывать возможные эффекты автоматического дополнения строк пробелами. Некоторые из этих функций также поддерживают битовые строки.

В SQL определены несколько строковых функций, в которых аргументы разделяются не запятыми, а ключевыми словами. Они перечислены в таблице 1.8. Синергия-БД также предоставляет варианты этих функций с синтаксисом, обычным для функций (таблица 1.9).

Таблица 1.8 – Строковые функции и операторы языка SQL

Функция	Тип результата	Описание	Пример	Результат
<code>string string</code>	<code>text</code>	Конкатенация строк	<code>'Post' 'greSQL'</code>	PostgreSQL
<code>string не string или не string string</code>	<code>text</code>	Конкатенация строк с одним не строковым операндом	<code>'Value: ' 42</code>	Value: 42
<code>bit_length(string)</code>	<code>int</code>	Число бит в строке	<code>bit_length('jose')</code>	32

Окончание таблицы 1.8

Функция	Тип результата	Описание	Пример	Результат
<code>char_length(string)</code> или <code>character_length(string)</code>	int	Число символов в строке	<code>char_length('jose')</code>	4
<code>lower(string)</code>	text	Переводит символы строки в нижний регистр	<code>lower('TOM')</code>	tom
<code>octet_length(string)</code>	int	Число байт в строке	<code>octet_length('jose')</code>	4
<code>overlay(string placing string from int [for int])</code>	text	Заменяет подстроку	<code>overlay('Txxxxas' placing 'hom' from 2 for 4)</code>	Thomas
<code>position(substring in string)</code>	int	Положение указанной подстроки	<code>position('om' in 'Thomas')</code>	3
<code>substring(string [from int] [for int])</code>	text	Извлекает подстроку	<code>substring('Thomas' from 2 for 3)</code>	hom
<code>substring(string from шаблон)</code>	text	Извлекает подстроку, соответствующую регулярному выражению в стиле POSIX. Подробно шаблоны описаны в разделе 7.7.	<code>substring('Thomas' from '...\$')</code>	mas
<code>substring(string from шаблон for спецсимвол)</code>	text	Извлекает подстроку, соответствующую регулярному выражению в стиле SQL. Подробно шаблоны описаны в разделе 7.7.	<code>substring('Thomas' from '%#>o_a#>_' for '#')</code>	oma
<code>trim([leading trailing both] [characters] from string)</code>	text	Удаляет наибольшую подстроку, содержащую только символы characters (по умолчанию пробелы), с начала (leading), с конца (trailing) или с обеих сторон (both, (по умолчанию)) строки string	<code>trim(both 'xyz' from 'yxTomxx')</code>	Tom
<code>trim([leading trailing both] [from] string [, characters])</code>	text	Нестандартный синтаксис trim()	<code>trim(both from 'yxTomxx', 'xyz')</code>	Tom
<code>upper(string)</code>	text	Переводит символы строки в верхний регистр	<code>upper('tom')</code>	TOM

Кроме этого, в Синергия-БД есть и другие функции для работы со строками, перечисленные в таблице 1.9. Некоторые из них используются в качестве внутренней реализации стандартных строковых функций SQL, приведенных в таблице 1.8.

Таблица 1.9 – Другие строковые функции

Функция	Тип результата	Описание	Пример	Результат
<code>ascii(string)</code>	<code>int</code>	Возвращает ASCII-код первого символа аргумента. Для UTF8 возвращает код символа в Unicode. Для других многобайтных кодировок аргумент должен быть ASCII-символом.	<code>ascii('x')</code>	120
<code>btrim(string text [, characters text])</code>	<code>text</code>	Удаляет наибольшую подстроку, состоящую только из символов <code>characters</code> (по умолчанию пробелов), с начала и с конца строки <code>string</code>	<code>btrim('xyxtrimyyx', 'xy')</code>	<code>trim</code>
<code>chr(int)</code>	<code>text</code>	Возвращает символ с данным кодом. Для UTF8 аргумент воспринимается как код символа Unicode, а для других кодировок он должен указывать на ASCII-символ. Код 0 (NULL) не допускается, так как байты с нулевым кодом в текстовых строках сохранить нельзя.	<code>chr(65)</code>	A
<code>concat(str «any» [, str «any» [, ...]])</code>	<code>text</code>	Соединяет текстовые представления всех аргументов, игнорируя NULL.	<code>concat('abcde', 2, NULL, 22)</code>	abcde222
<code>concat_ws(sep text, str «any» [, str «any» [, ...]])</code>	<code>text</code>	Соединяет все аргументы, кроме первого, через разделитель, игнорируя аргументы NULL. Разделитель указывается в первом аргументе.	<code>concat_ws(' ', 'abcde', 2, NULL, 22)</code>	abcde,2,22
<code>convert(string bytea, src_encoding name, dest_encoding name)</code>	<code>bytea</code>	Преобразует строку <code>string</code> из кодировки <code>src_encoding</code> в <code>dest_encoding</code> . Переданная строка должна быть допустимой для исходной кодировки. Преобразования могут быть определены с помощью CREATE CONVERSION. Все встроенные преобразования перечислены в таблице 7-8.	<code>convert('text_in_utf8', 'UTF8', 'LATIN1')</code>	строка <code>text_in_utf8</code> , представленная в кодировке Latin-1 (ISO 8859-1)

Продолжение таблицы 1.9

Функция	Тип результата	Описание	Пример	Результат
<code>convert_from(string bytea, src_encoding name)</code>	text	Преобразует строку string из кодировки src_encoding в кодировку базы данных. Переданная строка должна быть допустимой для исходной кодировки.	<code>convert_from('text_in_utf8', 'UTF8')</code>	строка text_in_utf8, представленная в кодировке текущей базы данных
<code>convert_to(string text, dest_encoding name)</code>	bytea	Преобразует строку в кодировку dest_encoding.	<code>convert_to('некоторый текст', 'UTF8')</code>	некоторый текст, представленный в кодировке UTF8
<code>decode(string text, format text)</code>	bytea	Получает двоичные данные из текстового представления в string. Значения параметра format те же, что и для функции encode.	<code>decode('MTIzAAE=', 'base64')</code>	\x3132330001
<code>encode(data bytea, format text)</code>	text	Переводит двоичные данные в текстовое представление в одном из форматов: base64, hex, escape. Формат escape преобразует нулевые байты и байты с 1 в старшем бите в восьмеричные последовательности \nnn и дублирует обратную косую черту.	<code>encode('123\000\001', 'base64')</code>	MTIzAAE=
<code>format(formatstr text [, formatarg «any» [, ...]])</code>	text	Форматирует аргумент в соответствии со строкой формата. Эта функция работает подобно sprintf в языке C. См. подраздел 1.5.1.	<code>format('Hello %s, %1\$s', 'World')</code>	Hello World, World
<code>initcap(string)</code>	text	Переводит первую букву каждого слова в строке в верхний регистр, а остальные – в нижний. Словами считаются последовательности алфавитно-цифровых символов, разделенные любыми другими символами.	<code>initcap('hi THOMAS')</code>	Hi Thomas
<code>left(string text, n int)</code>	text	Возвращает первые n символов в строке. Когда n меньше нуля, возвращаются все символы слева, кроме последних n .	<code>left('abcde', 2)</code>	ab
<code>length(string)</code>	int	Число символов в строке string	<code>length('jose')</code>	4

Продолжение таблицы 1.9

Функция	Тип результата	Описание	Пример	Результат
<code>length(string bytea, encoding name)</code>	<code>int</code>	Число символов, которые содержит строка <code>string</code> в заданной кодировке <code>encoding</code> . Переданная строка должна быть допустимой в этой кодировке.	<code>length('jose', 'UTF8')</code>	4
<code>lpad(string text, length int [, fill text])</code>	<code>text</code>	Дополняет строку <code>string</code> слева до длины <code>length</code> символами <code>fill</code> (по умолчанию пробелами). Если длина строки уже больше заданной, она обрезается справа.	<code>lpad('hi', 5, 'xy')</code>	<code>xyxhi</code>
<code>ltrim(string text [, characters text])</code>	<code>text</code>	Удаляет наибольшую подстроку, содержащую только символы <code>characters</code> (по умолчанию пробелы), с начала строки <code>string</code>	<code>ltrim('zzytrim', 'xyz')</code>	<code>trim</code>
<code>md5(string)</code>	<code>text</code>	Вычисляет MD5-хеш строки <code>string</code> и возвращает результат в 16-ричном виде	<code>md5('abc')</code>	90015098 3cd24fb0 d6963f7d 28e17f72
<code>parse_ident(qualified_identifier text [, strictmode boolean DEFAULT true])</code>	<code>text[]</code>	Раскладывает полный идентификатор, задаваемый параметром <code>qualified_identifier</code> , на массив идентификаторов, удаляя кавычки, обрамляющие отдельные идентификаторы. По умолчанию лишние символы после последнего идентификатора вызывают ошибку, но если отключить строгий режим (передать во втором параметре <code>false</code>), такие символы игнорируются. (Это поведение полезно для разбора имён таких объектов, как функции.) Заметьте, что эта функция не усекает чрезмерно длинные идентификаторы. Если вы хотите получить усечённые имена, можно привести результат к <code>name[]</code> .	<code>parse_ident('"SomeSchema".someTable')</code>	<code>{SomeSchema,sometable}</code>

Продолжение таблицы 1.9

Функция	Тип результата	Описание	Пример	Результат
<code>pg_client_encoding()</code>	name	Возвращает имя текущей клиентской кодировки	<code>pg_client_encoding()</code>	SQL_ASCII
<code>quote_ident(string text)</code>	text	Переданная строка оформляется для использования в качестве идентификатора в SQL - операторе. При необходимости идентификатор заключается в кавычки (например, если он содержит символы, недопустимые в открытом виде, или буквы в разном регистре). Если переданная строка содержит кавычки, они дублируются.	<code>quote_ident('Foo bar')</code>	«Foo bar»
<code>quote_literal(string text)</code>	text	Переданная строка оформляется для использования в качестве текстовой строки в SQL-операторе. Включенные символы апостроф и обратная косая черта при этом дублируются. Заметьте, что <code>quote_literal</code> возвращает NULL, когда на вход ей передается строка NULL; если же нужно получить представление и такого аргумента, лучше использовать <code>quote_nullable</code> .	<code>quote_literal(E'O\'Reilly')</code>	'O'Reilly'
<code>quote_literal(value anyelement)</code>	text	Переводит данное значение в текстовый вид и заключает в апострофы как текстовую строку. Символы апостроф и обратная косая черта при этом дублируются.	<code>quote_literal(42.5)</code>	'42.5'
<code>quote_nullable(string text)</code>	text	Переданная строка оформляется для использования в качестве текстовой строки в SQL-операторе; при этом для аргумента NULL возвращается строка NULL. Символы апостроф и обратная косая черта дублируются должным образом.	<code>quote_nullable(NULL)</code>	NULL

Продолжение таблицы 1.9

Функция	Тип результата	Описание	Пример	Результат
<code>quote_nullable(value anyelement)</code>	text	Переводит данное значение в текстовый вид и заключает в апострофы как текстовую строку, при этом для аргумента NULL возвращается строка NULL. Символы апостроф и обратная косая черта дублируются должным образом.	<code>quote_nullable(42.5)</code>	'42.5'
<code>regexp_match(string text, pattern text [, flags text])</code>	text[]	Возвращает подходящие подстроки, полученные из первого вхождения регулярного выражения POSIX в строке string.	<code>regexp_match('foobarbequebaz', '(bar) (beque)')</code>	{bar, beque}
<code>regexp_matches(string text, pattern text [, flags text])</code>	setof text[]	Возвращает все подходящие подстроки, полученные в результате применения регулярного выражения POSIX к string. Подробности описаны в подразделе 7.7.3.	<code>regexp_matches('foobarbequebaz', 'ba.', 'g')</code>	{bar} {baz} (2 строки)
<code>regexp_replace(string text, pattern text, replacement text [, flags text])</code>	text	Заменяет подстроки, соответствующие заданному регулярному выражению POSIX. Подробности описаны в подразделе 7.7.3.	<code>regexp_replace('Thomas', '[mN]a.', 'M')</code>	ThM
<code>regexp_split_to_array(string text, pattern text [, flags text])</code>	text[]	Разделяет содержимое string на элементы, используя в качестве разделителя регулярное выражение POSIX. Подробности описаны в подразделе 7.7.3.	<code>regexp_split_to_array('hello world', '\s+')</code>	{hello, world}
<code>regexp_split_to_table(string text, pattern text [, flags text])</code>	setof text	Разделяет содержимое string на элементы, используя в качестве разделителя регулярное выражение POSIX. Подробности описаны в подразделе 7.7.3.	<code>regexp_split_to_table('hello world', '\s+')</code>	hello world (2 строки)
<code>repeat(string text, number int)</code>	text	Повторяет содержимое string указанное число (number) раз	<code>repeat('Pg', 4)</code>	PgPgPgPg
<code>replace(string text, from text, to text)</code>	text	Заменяет все вхождения в string подстроки from подстрокой to	<code>replace('abcdefabcdef', 'cd', 'XX')</code>	abXXefabXXef
<code>reverse(str)</code>	text	Возвращает перевернутую строку	<code>reverse('abcde')</code>	edcba

Продолжение таблицы 1.9

Функция	Тип результата	Описание	Пример	Результат
<code>right(str text, n int)</code>	text	Возвращает последние n символов в строке. Когда n меньше нуля, возвращаются все символы справа, кроме первых n .	<code>right('abcde', 2)</code>	de
<code>rpads(string text, length int [, fill text])</code>	text	Дополняет строку string справа до длины length символами fill (по умолчанию пробелами). Если длина строки уже больше заданной, она обрезается.	<code>rpads('hi', 5, 'xy')</code>	hixyx
<code>rtrim(string text [, characters text])</code>	text	Удаляет наибольшую подстроку, содержащую только символы characters (по умолчанию пробелы), с конца строки string	<code>rtrim('trimxxxx', 'x')</code>	trim
<code>split_part(string text, delimiter text, field int)</code>	text	Разделяет строку string по символу delimiter и возвращает элемент по заданному номеру (считая с 1)	<code>split_part('abc~@~def~@~ghi', '~@~', 2)</code>	def
<code>strpos(string, substring)</code>	int	Возвращает положение указанной подстроки (подобно position(substring in string), но с другим порядком аргументов)	<code>strpos('high', 'ig')</code>	2
<code>substr(string, from [, count])</code>	text	Извлекает подстроку (подобно substring(string from from for count))	<code>substr('alphabet', 3, 2)</code>	ph
<code>starts_with(string, prefix)</code>	bool	Возвращает true, если строка string начинается с подстроки prefix.	<code>starts_with('alphabet', 'alph')</code>	t
<code>to_ascii(string text [, encoding text])</code>	text	Преобразует string в ASCII из кодировки encoding (поддерживаются только LATIN1, LATIN2, LATIN9 и WIN1250)	<code>to_ascii('Karel')</code>	Karel
<code>to_hex(number int or bigint)</code>	text	Преобразует число number в 16-ричный вид	<code>to_hex(2147483647)</code>	7fffffff

Окончание таблицы 1.9

Функция	Тип результата	Описание	Пример	Результат
translate(string text, from text, to text)	text	Заменяет символы в string, найденные в наборе from, на соответствующие символы в множестве to. Если строка from длиннее to, найденные в исходной строке лишние символы from удаляются.	translate('12345', '143', 'ax')	a2x5

Функции `concat`, `concat_ws` и `format` принимают переменное число аргументов, так что им для объединения или форматирования можно передавать значения в виде массива, помеченного ключевым словом `VARIADIC`. Элементы такого массива обрабатываются, как если бы они были обычными аргументами функции. Если вместо массива в соответствующем аргументе передается `NULL`, функции `concat` и `concat_ws` возвращают `NULL`, а `format` воспринимает `NULL` как массив нулевого размера.

См. также агрегатную функцию `string_agg` в разделе 1.21.

Таблица 1.10 – Встроенные преобразования

Имя преобразования	Исходная кодировка	Целевая кодировка
<code>ascii to mic</code>	SQL ASCII	MULE INTERNAL
<code>ascii to utf8</code>	SQL ASCII	UTF8
<code>big5 to euc tw</code>	BIG5	EUC TW
<code>big5 to mic</code>	BIG5	MULE INTERNAL
<code>big5 to utf8</code>	BIG5	UTF8
<code>euc cn to mic</code>	EUC CN	MULE INTERNAL
<code>euc cn to utf8</code>	EUC CN	UTF8
<code>euc jp to mic</code>	EUC JP	MULE INTERNAL
<code>euc jp to sjis</code>	EUC JP	SJIS
<code>euc jp to utf8</code>	EUC JP	UTF8
<code>euc kr to mic</code>	EUC KR	MULE INTERNAL
<code>euc kr to utf8</code>	EUC KR	UTF8
<code>euc tw to big5</code>	EUC TW	BIG5
<code>euc tw to mic</code>	EUC TW	MULE INTERNAL
<code>euc tw to utf8</code>	EUC TW	UTF8
<code>gb18030 to utf8</code>	GB18030	UTF8
<code>gbk to utf8</code>	GBK	UTF8
<code>iso 8859 10 to utf8</code>	LATIN6	UTF8
<code>iso 8859 13 to utf8</code>	LATIN7	UTF8
<code>iso 8859 14 to utf8</code>	LATIN8	UTF8
<code>iso 8859 15 to utf8</code>	LATIN9	UTF8
<code>iso 8859 16 to utf8</code>	LATIN10	UTF8
<code>iso 8859 1 to mic</code>	LATIN1	MULE INTERNAL
<code>iso 8859 1 to utf8</code>	LATIN1	UTF8
<code>iso 8859 2 to mic</code>	LATIN2	MULE INTERNAL
<code>iso 8859 2 to utf8</code>	LATIN2	UTF8

Продолжение таблицы 1.10

Имя преобразования	Исходная кодировка	Целевая кодировка
iso 8859 2 to windows 1250	LATIN2	WIN1250
iso 8859 3 to mic	LATIN3	MULE_INTERNAL
iso 8859 3 to utf8	LATIN3	UTF8
iso 8859 4 to mic	LATIN4	MULE_INTERNAL
iso 8859 4 to utf8	LATIN4	UTF8
iso 8859 5 to koi8_r	ISO 8859 5	KOI8R
iso 8859 5 to mic	ISO 8859 5	MULE_INTERNAL
iso 8859 5 to utf8	ISO 8859 5	UTF8
iso 8859 5 to windows 1251	ISO 8859 5	WIN1251
iso 8859 5 to windows 866	ISO 8859 5	WIN866
iso 8859 6 to utf8	ISO 8859 6	UTF8
iso 8859 7 to utf8	ISO 8859 7	UTF8
iso 8859 8 to utf8	ISO 8859 8	UTF8
iso 8859 9 to utf8	LATIN5	UTF8
johab to utf8	JOHAB	UTF8
koi8_r to iso 8859 5	KOI8R	ISO 8859 5
koi8_r to mic	KOI8R	MULE_INTERNAL
koi8_r to utf8	KOI8R	UTF8
koi8_r to windows 1251	KOI8R	WIN1251
koi8_r to windows 866	KOI8R	WIN866
koi8_u to utf8	KOI8U	UTF8
mic to ascii	MULE_INTERNAL	SQL_ASCII
mic to big5	MULE_INTERNAL	BIG5
mic to euc cn	MULE_INTERNAL	EUC_CN
mic to euc jp	MULE_INTERNAL	EUC_JP
mic to euc kr	MULE_INTERNAL	EUC_KR
mic to euc tw	MULE_INTERNAL	EUC_TW
mic to iso 8859 1	MULE_INTERNAL	LATIN1
mic to iso 8859 2	MULE_INTERNAL	LATIN2
mic to iso 8859 3	MULE_INTERNAL	LATIN3
mic to iso 8859 4	MULE_INTERNAL	LATIN4
mic to iso 8859 5	MULE_INTERNAL	ISO 8859 5
mic to koi8_r	MULE_INTERNAL	KOI8R
mic to sjis	MULE_INTERNAL	SJIS
mic to windows 1250	MULE_INTERNAL	WIN1250
mic to windows 1251	MULE_INTERNAL	WIN1251
mic to windows 866	MULE_INTERNAL	WIN866
sjis to euc jp	SJIS	EUC_JP
sjis to mic	SJIS	MULE_INTERNAL
sjis to utf8	SJIS	UTF8
tcvn to utf8	WIN1258	UTF8
uhc to utf8	UHC	UTF8
utf8 to ascii	UTF8	SQL_ASCII
utf8 to big5	UTF8	BIG5
utf8 to euc cn	UTF8	EUC_CN
utf8 to euc jp	UTF8	EUC_JP
utf8 to euc kr	UTF8	EUC_KR
utf8 to euc tw	UTF8	EUC_TW
utf8 to gbl8030	UTF8	GB18030
utf8 to gbk	UTF8	GBK
utf8 to iso 8859 1	UTF8	LATIN1
utf8 to iso 8859 10	UTF8	LATIN6
utf8 to iso 8859 13	UTF8	LATIN7
utf8 to iso 8859 14	UTF8	LATIN8
utf8 to iso 8859 15	UTF8	LATIN9
utf8 to iso 8859 16	UTF8	LATIN10
utf8 to iso 8859 2	UTF8	LATIN2

Окончание таблицы 1.10

Имя преобразования	Исходная кодировка	Целевая кодировка
utf8 to iso 8859 3	UTF8	LATIN3
utf8 to iso 8859 4	UTF8	LATIN4
utf8 to iso 8859 5	UTF8	ISO 8859 5
utf8 to iso 8859 6	UTF8	ISO 8859 6
utf8 to iso 8859 7	UTF8	ISO 8859 7
utf8 to iso 8859 8	UTF8	ISO 8859 8
utf8 to iso 8859 9	UTF8	LATIN5
utf8 to johab	UTF8	JOHAB
utf8 to koi8 r	UTF8	KOI8R
utf8 to koi8 u	UTF8	KOI8U
utf8 to sjis	UTF8	SJIS
utf8 to tcvn	UTF8	WIN1258
utf8 to uhc	UTF8	UHC
utf8 to windows 1250	UTF8	WIN1250
utf8 to windows 1251	UTF8	WIN1251
utf8 to windows 1252	UTF8	WIN1252
utf8 to windows 1253	UTF8	WIN1253
utf8 to windows 1254	UTF8	WIN1254
utf8 to windows 1255	UTF8	WIN1255
utf8 to windows 1256	UTF8	WIN1256
utf8 to windows 1257	UTF8	WIN1257
utf8 to windows 866	UTF8	WIN866
utf8 to windows 874	UTF8	WIN874
windows 1250 to iso 8859 2	WIN1250	LATIN2
windows 1250 to mic	WIN1250	MULE_INTERNAL
windows 1250 to utf8	WIN1250	UTF8
windows 1251 to iso 8859 5	WIN1251	ISO 8859 5
windows 1251 to koi8 r	WIN1251	KOI8R
windows 1251 to mic	WIN1251	MULE_INTERNAL
windows 1251 to utf8	WIN1251	UTF8
windows 1251 to windows 866	WIN1251	WIN866
windows 1252 to utf8	WIN1252	UTF8
windows 1256 to utf8	WIN1256	UTF8
windows 866 to iso 8859 5	WIN866	ISO 8859 5
windows 866 to koi8 r	WIN866	KOI8R
windows 866 to mic	WIN866	MULE_INTERNAL
windows 866 to utf8	WIN866	UTF8
windows 866 to windows 1251	WIN866	WIN
windows 874 to utf8	WIN874	UTF8
euc jis 2004 to utf8	EUC JIS 2004	UTF8
utf8 to euc jis 2004	UTF8	EUC_JIS_2004
shift jis 2004 to utf8	SHIFT JIS 2004	UTF8
utf8 to shift jis 2004	UTF8	SHIFT_JIS_2004
euc jis 2004 to shift jis 2004	EUC JIS 2004	SHIFT_JIS_2004
shift jis 2004 to euc jis 2004	SHIFT JIS 2004	EUC_JIS_2004

1.5. Функция форматирования

Функция `format` выдает текст, отформатированный в соответствии со строкой формата, подобно функции `sprintf` в C.

```
format(formatstr text [, formatarg «any» [, ...] ])
```

`formatstr` – строка, определяющая, как будет форматироваться результат.

Обычный текст в строке формата непосредственно копируется в результат, за

исключением спецификаторов формата. Спецификаторы формата представляют собой местозаполнители, определяющие, как должны форматироваться и выводиться в результате аргументы функции. Каждый аргумент `formatarg` преобразуется в текст по правилам вывода своего типа данных, а затем форматируется и вставляется в результирующую строку согласно спецификаторам формата.

Спецификаторы формата предваряются символом `%` и имеют форму:

`% [позиция] [флаги] [ширина] тип`

здесь:

1) позиция (необязателен)

Строка вида `n$`, где `n` – индекс выводимого аргумента. Индекс, равный 1, выбирает первый аргумент после `formatstr`. Если позиция опускается, по умолчанию используется следующий аргумент по порядку;

2) флаги (необязателен)

Дополнительные параметры, управляющие форматированием данного спецификатора. В настоящее время поддерживается только знак минус (`-`), который выравнивает результата спецификатора по левому краю. Он работает, только если также определена ширина;

3) ширина (необязателен)

Задаёт минимальное число символов, которое будет занимать результат данного спецификатора. Выводимое значение выравнивается по правой или левой стороне (в зависимости от флага `-`) с дополнением необходимым числом пробелов. Если ширина слишком мала, она просто игнорируется, т.е. результат не усекается. Ширину можно обозначить положительным целым, звездочкой (`*`), тогда ширина будет получена из следующего аргумента функции, или строкой вида `*n$`, тогда ширина будет задаваться в `n`-ом аргументе функции.

Если ширина передается в аргументе функции, этот аргумент выбирается до аргумента, используемого для спецификатора. Если аргумент ширины отрицательный, результат выравнивается по левой стороне (как если бы был указан флаг `-`) в рамках поля длины `abs(ширина)`;

4) тип (обязателен)

Тип спецификатора определяет преобразование соответствующего выводимого значения. Поддерживаются следующие типы:

- `s` форматирует значение аргумента как простую строку. Значение `NULL` представляется пустой строкой;
- `I` обрабатывает значение аргумента как SQL-идентификатор, при необходимости заключая его в кавычки. Значение `NULL` для такого преобразования считается ошибочным;
- `L` заключает значение аргумента в апострофы, как строку SQL. Значение `NULL` выводится буквально, как `NULL`, без кавычек.

В дополнение к спецификаторам, описанным выше, можно использовать спецпоследовательность `%%`, которая просто выведет символ `%`.

Несколько пример простых преобразований формата:

```
SELECT format('Hello %s', 'World');
```

Результат: Hello World

```
SELECT format('Testing %s, %s, %s, %%', 'one', 'two',  
'three');
```

Результат: Testing one, two, three, %

```
SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar',  
E'O\'Reilly');
```

Результат: INSERT INTO «Foo bar» VALUES('O'Reilly')

```
SELECT format('INSERT INTO %I VALUES(%L)', 'locations',  
E'C:\\Program Files');
```

Результат: INSERT INTO locations VALUES(E'C:\\Program Files')

Следующие примеры иллюстрируют использование поля ширина и флага -:

```
SELECT format('|%10s|', 'foo');
```

Результат: | foo|

```
SELECT format('|%-10s|', 'foo');
```

Результат: |foo|

```
SELECT format('|%*s|', 10, 'foo');
```

Результат: |foo|

```
SELECT format('|%*s|', -10, 'foo');
```

Результат: |foo|

```
SELECT format('|%-*s|', 10, 'foo');
```

Результат: |foo|

```
SELECT format('|%-*s|', -10, 'foo');
```

Результат: |foo|

Эти примеры показывают применение полей позиция:

```
SELECT format('Testing %3$s, %2$s, %1$s', 'one', 'two', 'three');
```

Результат: Testing three, two, one

```
SELECT format('|%*2$s|', 'foo', 10, 'bar');
```

Результат: |bar|

```
SELECT format('|%1$*2$s|', 'foo', 10, 'bar');
```

Результат: |foo|

В отличие от стандартной функции C `sprintf`, функция `format` в Синергия-БД позволяет комбинировать в одной строке спецификаторы с полями позиция и без них. Спецификатор формата без поля позиция всегда использует следующий аргумент после последнего выбранного. Кроме того, функция `format` не требует, чтобы в строке формата использовались все аргументы функции. Пример этого поведения:

```
SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');
```

Результат: Testing three, two, three

Спецификаторы формата `%I` и `%L` особенно полезны для безопасного

составления динамических операторов SQL.

1.6. Функции и операторы двоичных строк

В этом разделе описываются функции и операторы для работы с данными типа `bytea`.

В SQL определены несколько строковых функций, в которых аргументы разделяются не запятыми, а ключевыми словами. Подробнее это описано в таблице 1.11. Синергия-БД также предоставляет варианты этих функций с синтаксисом, обычным для функций (таблица 1.12).

Примечание. В примерах, приведенных на этой странице, подразумевается, что параметр сервера `bytea_output` равен `escape` (выбран традиционный формат Синергия-БД).

Таблица 1.11 – SQL-функции и операторы для работы с двоичными строками

Функция	Тип результата	Описание	Пример	Результат
<code>string string</code>	<code>bytea</code>	Конкатенация строк	<code>'\\Post'::bytea '047gres\000'::bytea</code>	<code>\\Post'gres\000</code>
<code>octet_length(string)</code>	<code>int</code>	Число байт в двоичной строке	<code>octet_length('jo\000se'::bytea)</code>	5
<code>overlay(string placing string from int [for int])</code>	<code>bytea</code>	Заменяет подстроку	<code>overlay('Th\000omas'::bytea placing '\002\003'::bytea from 2 for 3)</code>	<code>T\\002\\003mas</code>
<code>position(substring in string)</code>	<code>int</code>	Положение указанной подстроки	<code>position('\000om'::bytea in 'Th\000omas'::bytea)</code>	3
<code>substring(string [from int] [for int])</code>	<code>bytea</code>	Извлекает подстроку	<code>substring('Th\000omas'::bytea from 2 for 3)</code>	<code>h\000o</code>
<code>trim([both] bytes from string)</code>	<code>bytea</code>	Удаляет наибольшую строку, содержащую только байты, заданные в параметре <code>bytes</code> , с начала и с конца строки <code>string</code>	<code>trim('\000\001'::bytea from '\000Tom\001'::bytea)</code>	Tom

В Синергия-БД есть и другие функции для работы с двоичными строками, перечисленные в таблице 1.12. Некоторые из них используются в качестве внутренней реализации стандартных функций SQL, приведенных в таблице 1.11.

Таблица 1.12. Другие функции для работы с двоичными строками

Функция	Тип результата	Описание	Пример	Результат
btrim(string bytea, bytes bytea)	bytea	Удаляет наибольшую строку, содержащую только байты, заданные в параметре bytes, с начала и с конца строки string	btrim('\000trim\001'::bytea, '\000\001'::bytea)	trim
decode(string text, format text)	bytea	Получает двоичные данные из текстового представления в string. Значения параметра format те же, что и для функции encode.	decode('123\000456', 'escape')	123\000456
encode(data bytea, format text)	text	Переводит двоичные данные в текстовое представление в одном из форматов: base64, hex, escape. Формат escape преобразует нулевые байты и байты с 1 в старшем бите в восьмеричные последовательности \nnn и дублирует обратную косую черту.	encode('123\000456'::bytea, 'escape')	123\000456
get_bit(string, offset)	int	Извлекает бит из строки	get_bit('Th\000omas'::bytea, 45)	1
get_byte(string, offset)	int	Извлекает байт из строки	get_byte('Th\000omas'::bytea, 4)	109
length(string)	int	Длина двоичной строки	length('jo\000se'::bytea)	5
md5(string)	text	Вычисляет MD5-хеш строки string и возвращает результат в 16-ричном виде	md5('Th\000omas'::bytea)	8ab2d3c9689aaf18b4958c334c82d8b1
set_bit(string, offset, newvalue)	bytea	Устанавливает значение бита в строке	set_bit('Th\000omas'::bytea, 45, 0)	Th\000omAs
set_byte(string, offset, newvalue)	bytea	Устанавливает значение байта в строке	set_byte('Th\000omas'::bytea, 4, 64)	Th\000o@as
sha224(bytea)	bytea	Хеш SHA-224	sha224('abc')	\x23097d223405d8228642a477bda255b32aadbce4bda0b3f7e36c9da7
sha256(bytea)	bytea	Хеш SHA-256	sha256('abc')	\xba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad

Окончание таблицы 1.12

Функция	Тип результата	Описание	Пример	Результат
sha384 (bytea)	bytea	Хеш SHA-384	sha384 ('abc')	\xcb00753f45a35e8bb5a03d699ac65007272c32ab0eded1631a8b605a43ff5bed8086072ba1e7cc2358baeca134c825a7
sha512 (bytea)	bytea	Хеш SHA-512	sha512 ('abc')	\xddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee64b55d39a2192992a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f

Для функций `get_byte` и `set_byte` байты нумеруются с «0». Функции `get_bit` и `set_bit` нумеруют биты справа налево; например, бит «0» будет меньшим значащим битом первого байта, а бит 15 – большим значащим битом второго байта.

Заметьте, что по историческим причинам функция `md5` возвращает значение в шестнадцатеричном виде в типе `text`, тогда как функции SHA-2 возвращают тип `bytea`. Используйте функции `encode` и `decode` для преобразования значений из одного представления в другое. Например, вызвав `encode (sha256 ('abc'), 'hex')`, вы получите значение в шестнадцатеричном виде в строке.

См. также агрегатную функцию `string_agg` в разделе 1.21.

1.7. Функции и операторы для работы с битовыми строками

В этом разделе описываются функции и операторы, предназначенные для работы с битовыми строками, то есть с данными типов `bit` и `bit varying`. Помимо обычных операторов сравнения, с такими данными можно использовать операторы, перечисленные в таблице 1.13. Заметьте, что операторы `&`, `|` и `#` работают только с двоичными строками одинаковой длины. Операторы побитового сдвига сохраняют длины исходных строк, как показано в примерах.

Таблица 1.13 – Операторы для работы с битовыми строками

Оператор	Описание	Пример	Результат
	конкатенация	B'10001' B'011'	10001011
&	битовый AND	B'10001' & B'01101'	00001
	битовый OR	B'10001' B'01101'	11101
#	битовый XOR	B'10001' # B'01101'	11100
~	битовый NOT	~ B'10001'	01110
<<	битовый сдвиг влево	B'10001' << 3	01000
>>	битовый сдвиг вправо	B'10001' >> 2	00100

Следующие функции языка SQL работают как с символьными, так и с битовыми строками: `length`, `bit_length`, `octet_length`, `position`, `substring`, `overlay`.

С битовыми и двоичными строками работают функции `get_bit` и `set_bit`. При работе с битовыми строками эти функции нумеруют биты слева направо и самый левый бит считается нулевым.

Кроме того, целые значения можно преобразовать в тип `bit` и обратно. Например:

```

44::bit(10)           0000101100
44::bit(3)            100
cast(-44 as bit(12))  111111010100
'1110'::bit(4)::integer 14

```

Заметьте, что приведение к типу «`bit`» без длины будет означать приведение к `bit(1)`, и в результате будет получен только один менее значащий бит числа.

Примечание. Приведение целого числа к типу `bit(n)` копирует правые `n` бит числа. Если же целое преобразуется в битовую строку большей длины, чем требуется для этого числа, она дополняется слева битами знака числа.

1.8. Поиск по шаблону

Синергия-БД предлагает три разных способа поиска текста по шаблону: традиционный оператор `LIKE` языка SQL, более современный `SIMILAR TO` (добавленный в SQL:1999) и регулярные выражения в стиле POSIX. Помимо простых операторов, отвечающих на вопрос «соответствует ли строка этому шаблону?», в Синергия-БД есть функции для извлечения или замены соответствующих подстрок и для разделения строки по заданному шаблону.

Примечания:

1. Если этих встроенных возможностей оказывается недостаточно, вы можете

написать собственные функции на языке Perl или Tcl.

2. Хотя чаще всего поиск по регулярному выражению бывает очень быстрым, регулярные выражения бывают и настолько сложными, что их обработка может занять приличное время и объём памяти. Поэтому опасайтесь шаблонов регулярных выражений, поступающих из недоверенных источников. Если у вас нет другого выхода, рекомендуется ввести тайм-аут для операторов.

Поиск с шаблонами SIMILAR TO несёт те же риски безопасности, так как конструкция SIMILAR TO предоставляет во многом те же возможности, что и регулярные выражения в стиле POSIX.

Поиск с LIKE гораздо проще, чем два другие варианта, поэтому его безопаснее использовать с недоверенными источниками шаблонов поиска.

1.8.1. LIKE

строка LIKE шаблон [ESCAPE спецсимвол]

строка NOT LIKE шаблон [ESCAPE спецсимвол]

Выражение LIKE возвращает true, если строка соответствует заданному шаблону. Как можно было ожидать, выражение NOT LIKE возвращает false, когда LIKE возвращает true, и наоборот. Этому выражению равносильно выражение NOT (строка LIKE шаблон).

Если шаблон не содержит знаков процента и подчеркиваний, тогда шаблон представляет в точности строку и LIKE работает как оператор сравнения. Подчеркивание (_) в шаблоне подменяет (вместо него подходит) любой символ; а знак процента (%) подменяет любую (в том числе и пустую) последовательность символов.

Несколько примеров:

'abc' LIKE 'abc'	true
'abc' LIKE 'a%'	true
'abc' LIKE '_b_'	true
'abc' LIKE 'c'	false

При проверке по шаблону LIKE всегда рассматривается вся строка. Поэтому, если нужно найти последовательность символов где-то в середине строки, шаблон должен начинаться и заканчиваться знаками процента.

Чтобы найти в строке буквальное вхождение знака процента или

подчеркивания, перед соответствующим символом в шаблоне нужно добавить спецсимвол. По умолчанию в качестве спецсимвола выбрана обратная косая черта, но с помощью предложения `ESCAPE` можно выбрать и другой. Чтобы включить спецсимвол в шаблон поиска, продублируйте его.

Примечание. Если параметр `standard_conforming_strings` выключен, каждый символ обратной косой черты, записываемый в текстовой константе, нужно дублировать. Подробнее это описано в подразделе 2.1.2.1. (Часть 1. Руководство оператора).

Также можно отказаться от спецсимвола, написав `ESCAPE ''`. При этом механизм спецпоследовательностей фактически отключается и использовать знаки процента и подчеркивания буквально в шаблоне нельзя.

Вместо `LIKE` можно использовать ключевое слово `ILIKE`, чтобы поиск был регистр-независимым с учетом текущей языковой среды. Этот оператор не описан в стандарте SQL; это расширение Синергия-БД.

Кроме того, в Синергия-БД есть оператор `~~`, равнозначный `LIKE`, и `~~*`, соответствующий `ILIKE`. Есть также два оператора `!~~` и `!~~*`, представляющие `NOT LIKE` и `NOT ILIKE`, соответственно. Все эти операторы можно отнести к особенностям Синергия-БД.

Также существует оператор проверки префикса `^@` и соответствующая функция `starts_with`, которые покрывают случаи, когда нужно выполнить поиск только по началу строки.

1.8.2. Регулярные выражения **SIMILAR TO**

строка `SIMILAR TO` шаблон [`ESCAPE` спецсимвол]

строка `NOT SIMILAR TO` шаблон [`ESCAPE` спецсимвол]

Оператор `SIMILAR TO` возвращает `true` или `false` в зависимости от того, соответствует ли данная строка шаблону или нет. Он работает подобно оператору `LIKE`, только его шаблоны соответствуют определению регулярных выражений в стандарте SQL. Регулярные выражения SQL представляют собой любопытный гибрид синтаксиса `LIKE` с синтаксисом обычных регулярных выражений.

Как и `LIKE`, условие `SIMILAR TO` истинно, только если шаблон соответствует всей строке; это отличается от условий с регулярными выражениями,

в которых шаблон может соответствовать любой части строки. Также подобно LIKE, SIMILAR TO воспринимает символы _ и % как знаки подстановки, подменяющие любой один символ или любую подстроку, соответственно (в регулярных выражениях POSIX им аналогичны символы . и .*).

Помимо средств описания шаблонов, позаимствованных от LIKE, SIMILAR TO поддерживает следующие метасимволы, унаследованные от регулярных выражений POSIX:

- | означает выбор (одного из двух вариантов);
- * означает повторение предыдущего элемента 0 и более раз;
- + означает повторение предыдущего элемента 1 и более раз;
- ? означает вхождение предыдущего элемента 0 или 1 раз;
- {m} означает повторяет предыдущего элемента ровно m раз;
- {m,} означает повторение предыдущего элемента m или более раз;
- {m,n} означает повторение предыдущего элемента не менее чем m и не более чем n раз;
- скобки () объединяют несколько элементов в одну логическую группу;
- квадратные скобки [...] обозначают класс символов так же, как и в регулярных выражениях POSIX.

Обратите внимание, точка (.) не является метасимволом для оператора SIMILAR TO.

Как и с LIKE, обратная косая черта отменяет специальное значение любого из этих метасимволов, а предложение ESCAPE позволяет выбрать другой спецсимвол.

Несколько примеров:

```
'abc' SIMILAR TO 'abc'      true
'abc' SIMILAR TO 'a'        false
'abc' SIMILAR TO '%(b|d)%'  true
'abc' SIMILAR TO '(b|c)%'   false
```

Функция substring с тремя параметрами, substring (строка from шаблон for спецсимвол) извлекает подстроку, соответствующую шаблону регулярного выражения SQL. Как и с SIMILAR TO, указанному шаблону должна

соответствовать вся строка, в противном случае функция не найдет ничего и вернет NULL. Для обозначения части шаблона, которая должна быть возвращена в случае успеха, шаблон должен содержать два спецсимвола и кавычки («») после каждого. Эта функция возвращает часть шаблона между двумя такими маркерами.

Несколько примеров с маркерами #», выделяющими возвращаемую строку:

```
substring('foobar' from '%#>o_b#>%' for '#')    oob
substring('foobar' from '#>o_b#>%' for '#')    NULL
```

1.8.3. Регулярные выражения POSIX

В таблице 1.14 перечислены все существующие операторы для проверки строк регулярными выражениями POSIX.

Таблица 1.14 – Операторы регулярных выражений

Оператор	Описание	Пример
~	Проверяет соответствие регулярному выражению с учетом регистра	'thomas' ~ '.*thomas.*'
~*	Проверяет соответствие регулярному выражению без учета регистра	'thomas' ~* '.*Thomas.*'
!~	Проверяет несоответствие регулярному выражению с учетом регистра	'thomas' !~ '.*Thomas.*'
!~*	Проверяет несоответствие регулярному выражению без учета регистра	'thomas' !~* '.*vadim.*'

Регулярные выражения POSIX предоставляют более мощные средства поиска по шаблонам, чем операторы LIKE и SIMILAR TO. Во многих командах Unix, таких как egrep, sed и awk используется язык шаблонов, похожий на описанный здесь.

Регулярное выражение – это последовательность символов, представляющая собой краткое определение набора строк (регулярное множество). Строка считается соответствующей регулярному выражению, если она является членом регулярного множества, описываемого регулярным выражением. Как и для LIKE, символы шаблона непосредственно соответствуют символам строки, за исключением специальных символов языка регулярных выражений. При этом спецсимволы регулярных выражений отличается от спецсимволов LIKE. В отличие от шаблонов LIKE, регулярное выражение может совпадать с любой частью строки, если только оно не привязано явно к началу и/или концу строки.

Несколько примеров:

```
'abc' ~ 'abc'    true
```

```
'abc' ~ '^a'      true
'abc' ~ '(b|d)'    true
'abc' ~ '^(b|c)'  false
```

Более подробно язык шаблонов в стиле POSIX описан ниже.

Функция `substring` с двумя параметрами, `substring(строка from шаблон)`, извлекает подстроку, соответствующую шаблону регулярного выражения POSIX. Она возвращает фрагмент текста, подходящий шаблону, если таковой находится в строке, либо `NULL` в противном случае. Но если шаблон содержит скобки, она возвращает первое подвыражение, заключенное в скобки (то, которое начинается с самой первой открывающей скобки). Если вы хотите использовать скобки, но не в таком особом режиме, можно просто заключить в них все выражение. Если же вам нужно включить скобки в шаблон до подвыражения, которое вы хотите извлечь, это можно сделать, используя группы без захвата, которые будут описаны ниже.

Несколько примеров:

```
substring('foobar' from 'o.b')      oob
substring('foobar' from 'o(.)b')    o
```

Функция `regexp_match` возвращает текстовый массив из всех подходящих подстрок, полученных из первого вхождения шаблона регулярного выражения POSIX в строке. Она имеет синтаксис `regexp_match (строка, шаблон [, флаги])`. Если вхождение не находится, результатом будет `NULL`. Если вхождение находится и шаблон не содержит подвыражений в скобках, результатом будет текстовый массив с одним элементом, содержащим подстроку, соответствующую всему шаблону. Если вхождение находится и шаблон содержит подвыражения в скобках, результатом будет текстовый массив, в котором *n*-ым элементом будет *n*-ое заключённое в скобки подвыражение шаблона (не считая «незахватывающих» скобок; подробнее см. ниже). В параметре флаги передаётся необязательная текстовая строка, содержащая ноль или более однобуквенных флагов, меняющих поведение функции.

Несколько примеров:

```
SELECT regexp_match('foobarbequebaz', 'bar.*que');
       regexp_match
```

```
-----  
{barbeque}  
(1 row)
```

```
SELECT regexp_match('foobarbequebaz', '(bar) (beque)');  
      regexp_match  
-----  
{bar,beque}  
(1 row)
```

В общем случае просто получить всю найденную подстроку или NULL, если нет соответствия, можно примерно так:

```
SELECT (regexp_match('foobarbequebaz', 'bar.*que'))[1];  
      regexp_match  
-----  
barbeque  
(1 row)
```

Функция `regexp_matches` возвращает набор текстовых массивов со всеми подходящими подстроками, полученными в результате применения регулярного выражения POSIX к строке. Она имеет тот же синтаксис, что и `regexp_match`. Эта функция не возвращает никаких строк, если вхождений нет; возвращает одну строку, если найдено одно вхождение и не передан флаг `g`, или `N` строк, если найдено `N` вхождений и передан флаг `g`. Каждая возвращаемая строка представляет собой текстовый массив, содержащий всю найденную подстроку или подстроки, соответствующие заключённым в скобки подвыражениям шаблона, как и описанный выше результат `regexp_match`. Функция `regexp_matches` принимает все флаги, а также флаг `g`, указывающий ей выдать все вхождения, а не только первое.

Несколько примеров:

```
SELECT regexp_matches('foo', 'not there');  
      regexp_matches  
-----  
(0 rows)
```

```
SELECT regexp_matches('foobarbequebazilbarfbonk',
' (b[^b]+) (b[^b]+)', 'g');
    regexp_matches
-----
    {bar,beque}
    {bazil,barf}
(2 rows)
```

Примечание. В большинстве случаев `regexp_matches()` должна применяться с флагом `g`, так как если вас интересует только первое вхождение, проще и эффективнее использовать функцию `regexp_match()`. Однако `regexp_match()` существует только в Синергия-БД и в PostgreSQL версии 10 и выше. В старых версиях обычно помещали вызов `regexp_matches()` во вложенный `SELECT`, например, так:

```
SELECT col1, (SELECT regexp_matches(col2,
' (bar) (beque) ')) FROM tab;
```

В результате выдаётся текстовый массив, если вхождение найдено, или `NULL` в противном случае, так же как с `regexp_match()`. Без вложенного `SELECT` этот запрос не возвращает никакие строки, если соответствие не находится, а это обычно не то, что нужно.

Функция `regexp_split_to_table` разделяет строку, используя в качестве разделителя шаблон регулярного выражения POSIX. Она имеет синтаксис `regexp_split_to_table(строка, шаблон [, флаги])`. Если шаблон не находится в переданной строке, возвращается вся строка целиком. Если находится минимум одно вхождение, для каждого такого вхождения возвращается текст от конца предыдущего вхождения (или начала строки) до начала вхождения. После последнего найденного вхождения возвращается фрагмент от его конца до конца строки. В необязательном параметре флаги передается текстовая строка, содержащая ноль или более однобуквенных флагов, меняющих поведение функции.

Функция `regexp_split_to_array` ведет себя подобно `regexp_split_to_table`, за исключением того, что `regexp_split_to_array` возвращает результат в массиве элементов типа `text`. Она имеет синтаксис `regexp_split_to_array(строка, шаблон [, флаги])`. Параметры у этой функции те же, что и у `regexp_split_to_table`.

Несколько примеров:

```
SELECT foo FROM regexp_split_to_table('the quick brown  
fox jumps over the lazy dog', '\s+') AS foo;
```

```
foo  
-----  
the  
quick  
brown  
fox  
jumps  
over  
the  
lazy  
dog  
(9 rows)
```

```
SELECT regexp_split_to_array('the quick brown fox jumps  
over the lazy dog', '\s+');
```

```
regexp_split_to_array  
-----  
{the,quick,brown,fox,jumps,over,the,lazy,dog}  
(1 row)
```

```
SELECT foo FROM regexp_split_to_table('the quick brown  
fox', E'\\s*') AS foo;
```

```
foo  
-----  
t  
h  
e  
q  
u  
i  
c
```

```
k  
b  
r  
o  
w  
n  
f  
o  
x  
(16 rows)
```

Как показывает последний пример, функции разделения по регулярным выражениям игнорируют вхождения нулевой длины, идущие в начале и в конце строки, а также непосредственно за предыдущим вхождением. Это поведение противоречит строгому определению поиска по регулярным выражениям, которые реализуют функции `regexp_match` и `regexp_matches`, но обычно более удобно на практике. Подобное поведение наблюдается и в других программных средах, например в Perl.

1.8.3.1. Подробное описание регулярных выражений

Регулярные выражения в Синергия-БД реализованы с использованием программного пакета, который разработал Генри Спенсер (Henry Spencer). Практически все следующее описание регулярных выражений дословно скопировано из его руководства.

Регулярное выражение (Regular expression, RE), согласно определению в POSIX 1003.2, может иметь две формы: расширенное RE или ERE (грубо говоря, это выражения которые понимает egrep) и простое RE или BRE (грубо говоря, это выражения для ed). Синергия-БД поддерживает обе формы, а кроме того реализует некоторые расширения, не предусмотренные стандартом POSIX, но широко используемые вследствие их доступности в некоторых языках программирования, например в Perl и Tcl. Регулярные выражения, использующие эти несовместимые с POSIX расширения, здесь называются усовершенствованными RE или ARE. ARE практически представляют собой надмножество ERE, тогда как BRE отличаются некоторой несовместимостью в записи (помимо того, что они гораздо более

ограничены). Сначала мы опишем формы ARE и ERE, отметив особенности, присущие только ARE, а затем расскажем, чем от них отличаются BRE.

Примечание. Синергия-БД изначально всегда предполагает, что регулярное выражение следует правилам ARE. Однако можно переключиться на более ограниченные правила ERE или BRE, добавив в шаблон RE встроенный параметр, как описано в подразделе 1.7.3.4. Это может быть полезно для совместимости с приложениями, ожидающими от СУБД строгого следования правилам POSIX 1003.2.

Регулярное выражение определяется как одна или более ветвей, разделенных символами |. Оно считается соответствующим всему, что соответствует одной из этих ветвей.

Ветвь – это ноль или несколько количественных атомов или ограничений, соединенных вместе. Соответствие ветви в целом образуется из соответствия первой части, за которым следует соответствие второй части и т.д.; пустой ветви соответствует пустая строка.

Количественный атом – это атом, за которым может следовать определитель количества. Без этого определителя ему соответствует одно вхождение атома. С определителем количества ему может соответствовать некоторое число вхождений этого атома. Все возможные атомы перечислены в таблице 1.15. Варианты определителей количества и их значения перечислены в таблице 1.16.

Ограничению соответствует пустая строка, но это соответствие возможно только при выполнении определенных условий. Ограничения могут использоваться там же, где и атомы, за исключением того, что их нельзя дополнять определителями количества. Простые ограничения показаны в таблице 1.17; некоторые дополнительные ограничения описаны ниже.

Таблица 1.15 – Атомы регулярных выражений

Атом	Описание
(re)	(где re – любое регулярное выражение) описывает соответствие re, при этом данное соответствие захватывается для последующей обработки
(?:re)	подобно предыдущему, но соответствие не захватывается (т.е. это набор скобок «без захвата») (применимо только к ARE)
.	соответствует любому символу
[символы]	выражение в квадратных скобках, соответствует любому из символов (подробнее это описано в подразделе 1.8.3.2)
\k	(где k – не алфавитно-цифровой символ) соответствует обычному символу буквально, т.е. \\ соответствует обратной косой черте
\с	где с – алфавитно-цифровой символ (за которым могут следовать другие символы), это спецсимвол, см. подраздел 1.8.3.3 (применим только к ARE; в ERE и BRE этому атому соответствует с)
{	когда за этим символом следует любой символ, кроме цифры, этот атом соответствует левой фигурной скобке ({), если же за ним следует цифра, это обозначает начало границы
x	(где x – один символ, не имеющий специального значения) соответствует этому символу

Выражение RE не может заканчиваться обратной косой чертой (\).

Примечание. Если параметр `standard_conforming_strings` выключен, каждый символ обратной косой черты, записываемый в текстовой константе, нужно дублировать. Подробнее это описано в подразделе 2.1.2.1. (Часть 1. Руководство оператора).

Таблица 1.16 – Определители количества в регулярных выражениях

Определитель	Соответствует
*	0 или более вхождений атома
+	1 или более вхождений атома
?	0 или 1 вхождение атома
{m}	ровно m вхождений атома
{m, }	m или более вхождений атома
{m, n}	от m до n (включая границы) вхождений атома; m не может быть больше n
*?	не жадная версия *
+?	не жадная версия +
??	не жадная версия ?
{m}?	не жадная версия {m}
{m, }?	не жадная версия {m, }
{m, n}?	не жадная версия {m, n}

В формах с {...} числа m и n определяют так называемые границы количества. Эти числа должны быть беззнаковыми десятичными целыми в диапазоне от 0 до 255 включительно.

«Не жадные» определители (допустимые только в ARE) описывают те же возможные соответствия, что и аналогичные им обычные («жадные»), но предпочитают выбирать наименьшее, а не наибольшее количество вхождений.

Примечание. Определители количества не могут следовать один за другим, например запись ** будет ошибочной. Кроме того, определители не могут стоять в начале выражения или подвыражения и идти сразу после ^ или |.

Таблица 1.17 – Ограничения в регулярных выражениях

Ограничение	Описание
^	соответствует началу строки
\$	соответствует концу строки
(?= re)	позитивный <i>просмотр вперёд</i> находит соответствие там, где начинается подстрока, соответствующая re (только для ARE)
(?! re)	негативный <i>просмотр вперёд</i> находит соответствие там, где не начинается подстрока, соответствующая re (только для ARE)
(?<= re)	позитивный <i>просмотр назад</i> находит соответствие там, где заканчивается подстрока, соответствующая re (только для ARE)
(?<! re)	негативный <i>просмотр назад</i> находит соответствие там, где не заканчивается подстрока, соответствующая re (только для ARE)

Ограничения предпросмотра не могут содержать ссылки назад и все скобки в них считаются «скобками без захвата».

1.8.3.2. Выражения в квадратных скобках

Выражение в квадратных скобках содержит список символов, заключенный в

[]. Обычно ему соответствует любой символ из списка (об исключении написано ниже). Если список начинается с ^, ему соответствует любой символ, который не перечисляется далее в этом списке. Если два символа в списке разделяются знаком -, это воспринимается как краткая запись полного интервала символов между двумя заданными (и включая их) в порядке сортировки; например выражению [0-9] в ASCII соответствует любая десятичная цифра. Два интервала не могут разделять одну границу, т.е. выражение а-с-е недопустимо. Интервалы зависят от порядка сортировки, который может меняться, поэтому в переносимых программах их лучше не использовать.

Чтобы включить в список], этот символ нужно написать первым (сразу за ^, если он присутствует). Чтобы включить в список символ -, его нужно написать первым или последним, либо как вторую границу интервала. Указать - в качестве первой границы интервал можно, заключив его между [. и .], чтобы он стал элементом сортировки. За исключением этих символов, некоторых комбинаций с [(см. следующие абзацы) и спецсимволов (в ARE), все остальные специальные символы в квадратных скобках теряют свое особое значение. В частности, символ \ по правилам ERE или BRE воспринимается как обычный, хотя в ARE он экранирует символ, следующий за ним.

Выражения в квадратных скобках могут содержать элемент сортировки (символ или последовательность символов или имя такой последовательности), определение которого заключается между [. и .]. Определяющая его последовательность воспринимается в выражении в скобках как один элемент. Это позволяет включать в такие выражения элементы, соответствующие последовательности нескольких символов. Например, с элементом сортировки ch в квадратных скобках регулярному выражению [[.ch.]]*с будут соответствовать первые пять символов строки chchcc.

Примечание. В настоящее время Синергия-БД не поддерживает элементы сортировки, состоящие из нескольких символов. Эта информация относится к возможному в будущем поведению.

В квадратных скобках могут содержаться элементы сортировки, заключенные между [= и =], обозначающие классы эквивалентности, т.е. последовательности

символов из всех элементов сортировки, эквивалентных указанному, включая его самого. (Если для этого символа нет эквивалентных, он обрабатывается, как заключенный между [. и .].) Например, если `e` и `e` – члены одного класса эквивалентности, выражения `[[=e=]]`, `[[=e=]]` и `[ee]` будут равнозначными. Класс эквивалентности нельзя указать в качестве границы интервала.

В квадратных скобках может также содержаться имя класса символов, заключенное между [: и :], и заменяющее список всех символов этого класса. Стандартные имена классов: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper` и `xdigit`. Весь этот набор классов определен в `ctype` и он может меняться в зависимости от локали (языковой среды). Класс символов также нельзя использовать в качестве границы интервала.

Есть два особых вида выражений в квадратных скобках: выражения `[[:<:]]` и `[[:>:]]`, представляющие собой ограничения, соответствующие пустым строкам в начале и конце слова. Слово в данном контексте определяется как последовательность словосоставляющих символов, перед или после которой нет словосоставляющих символов. Словосоставляющий символ – это символ класса `alnum` (определенного в `ctype`) или подчеркивание. Это расширение совместимо со стандартом POSIX 1003.2, но не описано в нем, и поэтому его следует использовать с осторожностью там, где важна совместимость с другими системами. Обычно лучше использовать ограничивающие спецсимволы, описанные ниже; они также не совсем стандартны, но набрать их легче.

1.8.3.3. Спецсимволы регулярных выражений

Спецсимволы – это специальные команды, состоящие из \ и последующего алфавитно-цифрового символа. Можно выделить следующие категории спецсимволов: обозначения символов, коды классов, ограничения и ссылки назад. Символ \, за которым идет алфавитно-цифровой символ, не образующий допустимый спецсимвол, считается ошибочным в ARE. В ERE спецсимволов нет: вне квадратных скобок пара из \ и последующего алфавитно-цифрового символа, воспринимается просто как данный символ, а в квадратных скобках и сам символ \ воспринимается просто как обратная косая черта. Последнее на самом деле нарушает совместимость между ERE и ARE.

Спецобозначения символов введены для того, чтобы облегчить ввод в RE непечатаемых и других неудобных символов. Они приведены в таблице 1.18.

Коды классов представляют собой краткий способ записи имен некоторых распространенных классов символов. Они перечислены в таблице 1.19.

Спецсимволы ограничений обозначают ограничения, которым при совпадении определенных условий соответствует пустая строка. Они перечислены в таблице 1.20.

Ссылка назад (`\n`) соответствует той же строке, какой соответствовало предыдущее подвыражение в скобках под номером *n* (см. таблицу 1.21). Например, `([bc])\1` соответствует `bb` или `cc`, но не `bc` или `cb`. Это подвыражение должно полностью предшествовать ссылке назад в RE. Нумеруются подвыражения в порядке следования их открывающих скобок. При этом скобки без захвата исключаются из рассмотрения.

Таблица 1.18 – Спецобозначения символов в регулярных выражениях

Спецсимвол	Описание
<code>\a</code>	символ звонка, как в C
<code>\b</code>	символ «забой», как в C
<code>\B</code>	синоним для обратной косой черты (<code>\</code>), сокращающий потребность в дублировании этого символа
<code>\c X</code>	(где <i>X</i> – любой символ) символ, младшие 5 бит которого те же, что и у <i>X</i> , а остальные равны 0
<code>\e</code>	символ, определённый в последовательности сортировки с именем ESC, либо, если таковой не определён, символ с восьмеричным значением 033
<code>\f</code>	подача формы, как в C
<code>\n</code>	новая строка, как в C
<code>\r</code>	возврат каретки, как в C
<code>\t</code>	горизонтальная табуляция, как в C
<code>\u wxyz</code>	(где <i>wxyz</i> ровно четыре шестнадцатеричные цифры) символ с шестнадцатеричным кодом <code>0xwxyz</code>
<code>\U stuvwxyz</code>	(где <i>stuvwxyz</i> ровно восемь шестнадцатеричных цифр) символ с шестнадцатеричным кодом <code>0xstuvwxyz</code>
<code>\v</code>	вертикальная табуляция, как в C
<code>\x hhh</code>	(где <i>hhh</i> – несколько шестнадцатеричных цифр) символ с шестнадцатеричным кодом <code>0xhhh</code> (символ всегда один вне зависимости от числа шестнадцатеричных цифр)
<code>\0</code>	символ с кодом 0 (нулевой байт)
<code>\ xy</code>	(где <i>xy</i> – ровно две восьмеричных цифры, не ссылка назад) символ с восьмеричным кодом <code>0xy</code>
<code>\ xyz</code>	(где <i>xyz</i> – ровно три восьмеричных цифры, не ссылка назад) символ с восьмеричным кодом <code>0xyz</code>

Шестнадцатеричные цифры записываются символами 0-9 и a-f или A-F. Восьмеричные цифры – цифры от 0 до 7.

Спецпоследовательности с числовыми кодами, задающими значения вне диапазона ASCII (0-127), воспринимаются по-разному в зависимости от кодировки

базы данных. Когда база данных имеет кодировку UTF-8, спецкод равнозначен позиции символа в Unicode, например, \u1234 обозначает символ U+1234. Для других многобайтных кодировок спецпоследовательности обычно просто задают серию байт, определяющих символ. Если в кодировке базы данных отсутствует символ, заданный спецпоследовательностью, ошибки не будет, но и никакие данные не будут ей соответствовать.

Символы, переданные спецобозначением, всегда воспринимаются как обычные символы. Например, \135 кодирует] в ASCII, но спецпоследовательность \135 не будет закрывать выражение в квадратных скобках.

Таблица 1.19 – Спецкоды классов в регулярных выражениях

Спецсимвол	Описание
\d	[[[:digit:]]
\s	[[[:space:]]
\w	[[[:alnum:]]] (подчеркивание также включается)
\D	[^[:digit:]]
\S	[^[:space:]]
\W	[^[:alnum:]]] (подчеркивание также включается)

В выражениях в квадратных скобках спецсимволы \d, \s и \w теряют свои внешние квадратные скобки, а \D, \S и \W – недопустимы. Так что, например запись [a-c\d] равнозначна [a-c[:digit:]]. А запись [a-c\D], которая была бы равнозначна [a-c^[:digit:]], – недопустима.

Таблица 1.20 – Спецсимволы ограничений в регулярных выражений

Спецсимвол	Описание
\A	соответствует только началу строки (чем это отличается от ^, описано в подразделе 7.7.3.5)
\b	соответствует только началу слова
\B	соответствует только концу слова
\b	соответствует только началу или концу слова
\B	соответствует только положению не в начале и не в конце слова
\Z	соответствует только концу строки (чем это отличается от \$, описано в подразделе 7.7.3.5)

Определением слова здесь служит то же, что было приведено выше в описании [[[:<:]] и [[[:>:]]. В квадратных скобках спецсимволы ограничений не допускаются.

Таблица 1.21 – Ссылки назад в регулярных выражениях

Спецсимвол	Описание
\m	(где m – цифра, отличная от 0) – ссылка назад на подвыражение под номером m
\mnn	(где m – цифра, отличная от 0, а nn – ещё несколько цифр с десятичным значением mnn, не превышающим число закрытых до этого скобок с захватом) ссылка назад на подвыражение под номером mnn

Регулярным выражениям присуща неоднозначность между восьмеричными

кодами символов и ссылками назад, которая разрешается следующим образом (это упоминалось выше). Ведущий ноль всегда считается признаком восьмеричной последовательности. Единственная цифра, отличная от 0, за которой не следует ещё одна цифра, всегда воспринимается как ссылка назад. Последовательность из нескольких цифр, которая начинается не с «0», воспринимается как ссылка назад, если она идет за подходящим подвыражением (т.е. число оказывается в диапазоне, допустимом для ссылки назад), в противном случае она воспринимается как восьмеричное число.

1.8.3.4.Метасинтаксис регулярных выражений

В дополнение к основному синтаксису, описанному выше, можно использовать также несколько особых форм и разнообразные синтаксические удобства.

Регулярное выражение может начинаться с одного из двух специальных префиксов режима. Если RE начинается с *****:**, его продолжение рассматривается как ARE. В Синергия-БД это обычно не имеет значения, так как регулярные выражения воспринимаются как ARE по умолчанию; но это может быть полезно, когда параметр флаги функций **regex** включает режим ERE или BRE. Если RE начинается с *****=**, его продолжение воспринимается как обычная текстовая строка, все его символы воспринимаются буквально.

ARE может начинаться со встроенных параметров: последовательности (**?xyz**) (где **xyz** – один или несколько алфавитно-цифровых символов), определяющих параметры остального регулярного выражения. Эти параметры переопределяют любые ранее определенные параметры, в частности они могут переопределить режим чувствительности к регистру, подразумеваемый для оператора **regex**, или параметр флаги функции **regex**. Допустимые буквы параметров показаны в таблице 1.22. Заметьте, что те же буквы используются в параметре флаги функций **regex**.

Таблица 1.22 – Буквы встроенных параметров ARE

Параметр	Описание
b	продолжение регулярного выражения – BRE
c	поиск соответствий с учетом регистра (переопределяет тип оператора)
e	продолжение RE – ERE

Окончание таблицы 1.22

Параметр	Описание
i	поиск соответствий без учета регистра (см. в подразделе 1.8.3.5) (переопределяет тип оператора)
m	исторически сложившийся синоним n
n	поиск соответствий с учетом перевода строк
p	переводы строк учитываются частично
q	продолжение регулярного выражения – обычная строка («в кавычках»), содержимое которой воспринимается буквально
s	поиск соответствий без учета перевода строк (по умолчанию)
t	компактный синтаксис (по умолчанию)
w	переводы строк учитываются частично, но в другом, «странном» режиме (см. в подразделе 1.8.3.5)
x	развернутый синтаксис

Внедренные параметры начинают действовать сразу после скобки), завершающей их последовательность. Они могут находиться только в начале ARE (после указания ***, если оно присутствует).

Помимо обычного (компактного) синтаксиса RE, в котором имеют значение все символы, поддерживается также развернутый синтаксис, включить который можно с помощью встроенного параметра x. В развернутом синтаксисе игнорируются пробельные символы, а также все символы от # до конца строки (или конца RE). Это позволяет разделять RE на строки и добавлять в него комментарии. Но есть три исключения:

- пробельный символ или #, за которым следует \, сохраняется;
- пробельный символ или # внутри выражения в квадратных скобках сохраняется;
- пробельные символы и комментарии не могут присутствовать в составных символах.

В данном контексте пробельными символами считаются пробел, табуляция, перевод строки и любой другой символ, относящийся к классу символов space.

И наконец, в ARE последовательность (?#ttt) (где ttt – любой текст, не содержащий)) вне квадратных скобок также считается комментарием и полностью игнорируется. При этом она так же не может находиться внутри составных символов, таких как (?:. Эти комментарии в большей степени историческое наследие, чем полезное средство; они считаются устаревшими, а вместо них рекомендуется использовать развернутый синтаксис.

Ни одно из этих расширений метасинтаксиса не будет работать, если

выражение начинается с префикса *****=**, после которого строка воспринимается буквально, а не как RE.

1.8.3.5. Правила соответствия регулярным выражениям

В случае, когда RE может соответствовать более чем одной подстроке в заданной строке, соответствующей RE считается подстрока, которая начинается в ней первой. Если к данной позиции подобных соответствующих подстрок оказывается несколько, из них выбирается либо самая длинная, либо самая короткая из возможных, в зависимости от того, какой режим выбран в RE: жадный или не жадный.

Где жадный или не жадный характер RE определяется по следующим правилам:

- большинство атомов и все ограничения не имеют признака жадности (так как они все равно не могут соответствовать подстрокам разного состава);
- скобки, окружающие RE, не влияют на его «жадность»;
- атом с определителем фиксированного количества ($\{m\}$ или $\{m\}?$) имеет ту же характеристику жадности (или может не иметь ее), как и сам атом;
- атом с другими обычными определителями количества (включая $\{m,n\}$, где m равняется n) считается жадным (предпочитает соответствие максимальной длины);
- атом с не жадным определителем количества (включая $\{m,n\}?$, где m равно n) считается не жадным (предпочитает соответствие минимальной длины);
- ветвь (RE без оператора $|$ на верхнем уровне) имеет ту же характеристику жадности, что и первый количественный атом в нем, имеющий атрибут жадности.
- RE, образованное из двух или более ветвей, соединенных оператором $|$, всегда считается жадным.

Эти правила связывают характеристики жадности не только с отдельными количественными атомами, но и с ветвями и целыми RE, содержащими количественные атомы. Это означает, что при сопоставлении ветвь или целое RE может соответствовать максимально длинной или короткой подстроке в целом. Когда определена длина всего соответствия, часть его, соответствующая конкретному

подвыражению, определяется с учетом характеристики жадности для этого подвыражения, при этом подвыражения, начинающиеся в RE раньше, имеют больший приоритет, чем следующие за ними.

Это иллюстрирует следующий пример:

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
```

Результат: 123

```
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
```

Результат: 1

В первом случае, RE в целом жадное, так как жадным является атом Y*. Соответствие ему начинается с буквы Y и оно включает подстроку максимальной длины с этого места, т.е. подстроку Y123. Результат выражения – её часть, соответствующая подвыражению в скобках, т.е. 123. Во втором случае, RE в целом наследует не жадный характер от атома Y*?. Соответствие ему так же начинается с Y, но включает оно подстроку минимальной длины с этого места, т.е. Y1. И хотя подвыражение [0-9]{1,3} имеет жадный характер, оно не может повлиять на выбор длины соответствия в целом, поэтому ему остается только подстрока 1.

Другими словами, когда RE содержит и жадные, и не жадные подвыражения, все соответствие будет максимально длинным или коротким в зависимости от характеристики всего RE. Характеристики, связанные с подвыражениями, влияют только на то, какую часть подстроки может «поглотить» одно подвыражение относительно другого.

Чтобы явно придать характеристику «жадности» или «нежадности» подвыражению или всему RE, можно использовать определители {1,1} и {1,1}?, соответственно. Это полезно, когда вам нужно, чтобы общая характеристика жадности RE отличалась от той, что вытекает из его элементов. Например, предположим, что вы пытаетесь выделить из строки, содержащей несколько цифр, эти цифры и части до и после них. Можно попытаться сделать это так:

```
SELECT regexp_match('abc01234xyz', '(.*) (\d+) (.*)');
```

Результат: {abc0123, 4, xyz}

Но это не будет работать: первая группа .* – «жадная», она «съест» всё, что сможет, оставляя для соответствия \d+ только последнюю возможность, то есть

последнюю цифру. Можно попытаться сделать запрос «нежадным»:

```
SELECT regexp_match('abc01234xyz', '(.*) (\d+) (.*)');
```

Результат: {abc,0,""}

И это не будет работать, так теперь весь RE в целом стал нежадным, и все соответствия завершаются как можно раньше. Но мы можем получить нужный результат, явно сделав жадным всё RE:

```
SELECT regexp_match('abc01234xyz',  
'(?: (.*) (\d+) (.*)) {1,1}');
```

Результат: {abc,01234,xyz}

Управление общей характеристикой «жадности» RE независимо от «жадности» его компонентов даёт большую гибкость в описании шаблонов переменной длины.

При определении более длинного или более короткого соответствия длины соответствий определяются в символах, а не в элементах сортировки. Пустая строка считается длиннее, чем отсутствие соответствия. Например, выражению `bb*` соответствуют три символа в середине строки `abbbbc`, выражению `(week|wee)(night|knights)` – все десять символов `weeknights`; когда выражение `(.*)*` сопоставляется со строкой `abc`, подвыражению в скобках соответствуют все три символа; а когда `(a*)*` сопоставляется со строкой `bc`, то и RE в целом, и подстроке в скобках соответствует пустая строка.

Игнорирование регистра символов даёт практически тот же эффект, как если бы в алфавите исчезли различия прописных и строчных букв. Если буква, существующая и в верхнем, и в нижнем регистре, фигурирует вне квадратных скобок как обычный символ, она по сути преобразуется в выражение в квадратных скобках, содержащее оба варианта, например `x` становится `[xX]`. Если же она фигурирует в выражении в квадратных скобках, в это выражение добавляются все её варианты, например `[x]` становится `[xX]`, а `^[x]` – `^[xX]`.

Когда включен режим учета перевода строк, атом `.` и выражения в квадратных скобках с `^` никогда не будут соответствовать символам конца строки (так что соответствия никогда не будут пересекать границы строк, если в RE нет явных указаний на эти символы), а `^` и `$` будут соответствовать пустой подстроке не только

в начале и конце всего текста, но и в начале и конце каждой отдельной его строки. Однако спецсимволы ARE \A и \Z по-прежнему будут соответствовать только началу и концу всего текста.

В режиме, когда переводы строк учитываются частично, особый смысл перевод строк имеет для атома . и выражений в квадратных скобках, но не для ^ и \$.

В обратном частичном режиме, перевод строк имеет особый смысл для ^ и \$, как и в режиме с учетом перевода строк, но не для . и выражений в квадратных скобках. Данный режим не очень полезен, но существует для симметрии.

1.8.3.6. Пределы и совместимость

В текущей реализации отсутствует какой-либо явно заданный предел длины RE. Однако, разрабатывая программы высокой степени переносимости, не следует применять RE длиннее 256 байт, так как другая POSIX-совместимая реализация может отказаться обрабатывать такие регулярные выражения.

Единственная особенность ARE, действительно несовместимая с ERE стандарта POSIX проявляется в том, что в ARE знак \ не теряет свое специальное значение в квадратных скобках. Все другие расширения ARE используют синтаксические возможности, которые не определены, не допустимы или не поддерживаются в ERE; синтаксис переключения режимов (***) также выходит за рамки синтаксиса POSIX как для BRE, так и для ERE.

Многие расширения ARE заимствованы из языка Perl, но некоторые были изменены, оптимизированы, а некоторые расширения Perl были исключены. В результате имеют место следующие несовместимости: атомы \b и \B, отсутствие специальной обработки завершающего перевода строки, добавление исключений в квадратных скобках в число случаев, когда учитывается перевод строк, особые условия для скобок и ссылок назад в ограничениях предпросмотра и семантика «наиболее длинное/короткое соответствие» (вместо «первое соответствие»).

1.8.3.7. Простые регулярные выражения

BRE имеют ряд отличий от ERE. В BRE знаки |, + и ? теряют специальное значение, а замены им нет. Границы количества окружаются символами \{ и \}, тогда как { и } рассматриваются как обычные символы. Вложенные подвыражения помещаются между \ (и \), а (и) представляют обычные символы. Символ ^

воспринимается как обычный, если только он не находится в начале RE или подвыражения в скобках, \$ – тоже обычный символ, если он находится не в конце RE или в конце подвыражения в скобках, и * – обычный символ, когда он находится в начале RE или подвыражения в скобках (возможно, после начального ^). И, наконец, в BRE работают ссылки назад с одной цифрой, \< и \> – синонимы для [[:<:]] и [[:>:]], соответственно; никакие другие спецсимволы в BRE не поддерживаются.

1.9. Функции форматирования данных

Функции форматирования в Синергия-БД предоставляют богатый набор инструментов для преобразования самых разных типов данных (дата/время, целое, числа с плавающей и фиксированной точкой) в форматированные строки и обратно. Все они перечислены в таблице 1.23. Все эти функции следует одному соглашению: в первом аргументе передается значение, которое нужно отформатировать, а во втором – шаблон, определяющий формат ввода или вывода.

Таблица 1.23 – Функции форматирования

Функция	Тип результата	Описание	Пример
to_char(timestamp, text)	text	преобразует время в текст	to_char(current_timestamp, 'HH12:MI:SS')
to_char(interval, text)	text	преобразует интервал в текст	to_char(interval '15h 2m 12s', 'HH24:MI:SS')
to_char(int, text)	text	преобразует целое в текст	to_char(125, '999')
to_char(double precision, text)	text	преобразует плавающее одинарной/двойной точности в текст	to_char(125.8::real, '999D9')
to_char(numeric, text)	text	преобразует числовое значение в текст	to_char(-125.8, '999D99S')
to_date(text, text)	date	преобразует текст в дату	to_date('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	преобразует текст в число	to_number('12,454.8-', '99G999D9S')
to_timestamp(text, text)	timestamp with time zone	преобразует строку во время	to_timestamp('05 Dec 2000', 'DD Mon YYYY')

Примечания:

1. Также имеется функция to_timestamp с одним аргументом.
2. Функции to_timestamp и to_date предназначены для работы с входными форматами, которые нельзя преобразовать простым приведением. Для большинства стандартных форматов даты/времени работает простое приведение исходной строки к требуемому типу и использовать его гораздо легче. Так же и функцию to_number нет необходимости использовать для стандартных

представлений чисел.

Шаблон вывода `to_char` может содержать ряд кодов, которые распознаются при форматировании и заменяются соответствующими данными. Любой текст, который не является кодом, копируется в результат в неизменном виде. Подобным образом, в строке шаблона ввода (для других функций) коды шаблона определяют, какие значения содержит передаваемая текстовая строка.

Все коды форматирования даты и времени перечислены в таблице 1.24.

Таблица 1.24 – Коды форматирования даты/времени

Код	Описание
HH	час (01-12)
HH12	час (01-12)
HH24	час (00-23)
MI	минута (00-59)
SS	секунда (00-59)
MS	миллисекунда (000-999)
US	микросекунда (000000-999999)
SSSS	число секунд с начала суток (0-86399)
AM, am, PM или pm	обозначение времени до/после полудня (без точек)
A.M., a.m., P.M. или p.m.	обозначение времени до/после полудня (с точками)
Y, YYY	год (4 или более цифр) с разделителем
YYYY	год (4 или более цифр)
YYY	последние 3 цифры года
YY	последние 2 цифры года
Y	последняя цифра года
IYYY	недельный год по ISO 8601 (4 или более цифр)
IYY	последние 3 цифры недельного года по ISO 8601
IY	последние 2 цифры недельного года по ISO 8601
I	последняя цифра недельного года по ISO 8601
BC, bc, AD или ad	обозначение эры (без точек)
B.C., b.c., A.D. или a.d.	обозначение эры (с точками)
MONTH	полное название месяца в верхнем регистре (дополненное пробелами до 9 символов)
Month	полное название месяца с большой буквы (дополненное пробелами до 9 символов)
month	полное название месяца в нижнем регистре (дополненное пробелами до 9 символов)
MON	сокращенное название месяца в верхнем регистре (3 буквы в английском; в других языках длина может меняться)
Mon	сокращенное название месяца с большой буквы (3 буквы в английском; в других языках длина может меняться)
mon	сокращенное название месяца в нижнем регистре (3 буквы в английском; в других языках длина может меняться)
MM	номер месяца (01-12)
DAY	полное название дня недели в верхнем регистре (дополненное пробелами до 9 символов)
Day	полное название дня недели с большой буквы (дополненное пробелами до 9 символов)
day	полное название дня недели в нижнем регистре (дополненное пробелами до 9 символов)
DY	сокращенное название дня недели в верхнем регистре (3 буквы в английском; в других языках может меняться)
Dy	сокращенное название дня недели с большой буквы (3 буквы в английском; в других языках длина может меняться)
dy	сокращенное название дня недели в нижнем регистре (3 буквы в английском; в других языках длина может меняться)
DDD	номер дня в году (001-366)
IDDD	номер дня в году по ISO 8601 (001-371; 1 день – понедельник первой недели по ISO)

Окончание таблицы 1.24

Код	Описание
DD	день месяца (01-31)
D	номер дня недели, считая с воскресенья (1) до субботы (7)
ID	номер дня недели по ISO 8601, считая с понедельника (1) до воскресенья (7)
W	неделя месяца (1-5) (первая неделя начинается в первое число месяца)
WW	номер недели в году (1-53) (первая неделя начинается в первый день года)
IW	номер недели в году по ISO 8601 (01-53; первый четверг года относится к неделе 1)
CC	век (2 цифры) (двадцать первый век начался 2001-01-01)
J	День по юлианскому календарю (номер дня с 24 ноября 4714 г. до н.э.)
Q	квартал (игнорируется функциями <code>to_date</code> и <code>to_timestamp</code>)
RM	номер месяца римскими цифрами в верхнем регистре (I-XII; I=январь)
rm	номер месяца римскими цифрами в нижнем регистре (i-xii; i=январь)
TZ	название часового пояса в верхнем регистре
tz	название часового пояса в нижнем регистре
TZH	часы часового пояса
TZM	минуты часового пояса
OF	смещение часового пояса от UTC (поддерживается только в <code>to_char</code>)

К любым кодам форматирования можно добавить модификаторы, изменяющие их поведение. Например, шаблон форматирования FMMonth включает код Month с модификатором FM. Модификаторы, предназначенные для форматирования даты/времени, перечислены в таблице 1.25.

Таблица 1.25 – Модификаторы кодов для форматирования даты/времени

Модификатор	Описание	Пример
Приставка FM	режим заполнения (подавляет ведущие нули и дополнение пробелами)	FMMonth
Окончание TH	окончание порядкового числительного в верхнем регистре	DDTH, например 12TH
Окончание th	окончание порядкового числительного в нижнем регистре	DDth, например 12th
Приставка FX	глобальный параметр фиксированного формата (см. замечания)	FX Month DD Day
Приставка TM	режим перевода (выводятся локализованные названия дней и месяцев, исходя из <code>lc_time</code>)	TMMonth
Окончание SP	режим числа прописью (не реализован)	DDSP

Замечания по использованию форматов даты/времени:

- 1) FM подавляет дополняющие пробелы и нули справа, которые в противном случае будут добавлены, чтобы результат имел фиксированную ширину. В Синергия-БД модификатор FM действует только на следующий код, тогда как в Oracle FM её действие распространяется на все последующие коды, пока не будет отключено последующим модификатором FM;
- 2) TM не затрагивает замыкающие пробелы. Функции `to_timestamp` и `to_date` игнорируют указание TM;
- 3) `to_timestamp` и `to_date` пропускают повторяющиеся пробелы во входной строке, если только не используется параметр FX. Например,

`to_timestamp('2000 JUN', 'YYYY MON')` будет работать, но `to_timestamp('2000 JUN', 'FXYYYY MON')` вернет ошибку, так как `to_timestamp` в данном случае ожидает только один разделяющий пробел.

Приставка `FX` должна быть первой в шаблоне;

4) шаблоны для `to_char` могут содержать обычный текст; он будет выведен в неизменном виде. Чтобы принудительно вывести текст буквально, даже если он содержит шаблонные коды, подстроку с ним можно заключить в кавычки. Например, в строке `"Hello Year "YYYY"`, код `YYYY` будет заменён годом, а буква `Y` в слове `Year` останется неизменной. В функциях `to_date`, `to_number` и `to_timestamp` при обработке подстрок в кавычках и буквального текста некоторой длины пропускается такое же число символов во входной строке; например, при обработке подстроки `"XX"` будут пропущены два символа (любые, не обязательно `XX`);

5) если вам нужно получить на выходе двойные кавычки, вы должны предварить их символом обратной косой черты, например: `"\"YYYY Month\""`. В остальном этот символ вне кавычек воспринимается как обычный. Внутри строки в кавычках он указывает, что следующий символ должен восприниматься буквально, каким бы он ни был (но это имеет смысл, только если следующий символ – кавычки или обратная косая черта);

6) если в функциях `to_timestamp` и `to_date` формат года определяется менее, чем 4 цифрами, например, как `YYYY`, и в переданном значении года тоже меньше 4 цифр, год пересчитывается в максимально близкий к году 2020, т. е. 95 воспринимается как 1995;

7) в функциях `to_timestamp` и `to_date` с преобразованием `YYYY` связано ограничение, когда обрабатываемый год записывается более чем 4 цифрами. После `YYYY` необходимо будет добавить нецифровой символ или соответствующий код, иначе год всегда будет восприниматься как 4 цифры. Например, в `to_date('200001131', 'YYYYMMDD')` (с годом 20000) год будет интерпретирован как состоящий из 4 цифр; чтобы исправить ситуацию, нужно добавить нецифровой разделитель после года, как в `to_date('20000-1131',`

'YYYY-MMDD'), или код как в `to_date('20000Nov31', 'YYYYMonDD');`

8) функции `to_timestamp` и `to_date` принимают поле `CC` (век), но игнорируют его, если в шаблоне есть поле `YYY`, `YYYY` или `Y,YYY`. Если `CC` используется с `YY` или `Y`, результатом будет год в данном столетии. Если присутствует только код столетия, без года, подразумевается первый год этого века;

9) функции `to_timestamp` и `to_date` принимают названия и номера дней недели (`DAY`, `D` и связанные типы полей), но игнорируют их при вычислении результата. То же самое происходит с полями квартала (`Q`);

10) функциям `to_timestamp` и `to_date` можно передать даты по недельному календарю ISO 8601 (отличающиеся от григорианских) одним из двух способов:

- год, номер недели и дня недели: например, `to_date('2006-42-4', 'YYYY-IW-ID')` возвращает дату 2006-10-19. Если день недели опускается, он считается равным 1 (понедельнику);

- год и день года: например, `to_date('2006-291', 'YYYY-IDDD')` также возвращает 2006-10-19.

Попытка ввести дату из смеси полей григорианского и недельного календаря ISO 8601 бессмысленна, поэтому это будет считаться ошибкой. В контексте ISO 8601 понятия «номер месяца» и «день месяца» не существуют, а в григорианском календаре нет понятия номера недели по ISO.

Примечание. Тогда как `to_date` не примет смесь полей григорианского и недельного календаря ISO, `to_char` способна на это, так как форматы вроде `YYYY-MM-DD` (`IYYYY-IDDD`) могут быть полезны. Но избегайте форматов типа `IYYYY-MM-DD`; в противном случае с датами в начале года возможны сюрпризы.

При преобразовании из текстовой строки в `timestamp`, миллисекунды (`MS`) или микросекунды (`US`) воспринимаются как дробная часть числа секунд. Например, `to_timestamp('12:3', 'SS:MS')` – это не 3 миллисекунды, а 300, так как это значение воспринимается как 12 + 0.3 сек. Это значит, что для формата `SS:MS` вводимые значения 12:3, 12:30 и 12:300 задают одно и то же число миллисекунд. Чтобы получить три миллисекунды, время нужно записать в виде 12:003, тогда это будет

воспринято как $12 + 0.003 = 12.003$ сек.

Ещё более сложный пример: `to_timestamp('15:12:02.020.001230', 'HH:MI:SS.MS.US')` будет преобразовано в 15 часов, 12 минут и 2 секунды + 20 миллисекунд + 1230 микросекунд = 2.021230 seconds.

Нумерация дней недели в `to_char(..., 'ID')` соответствует функции `extract(isodow from ...)`, но нумерация `to_char(..., 'D')` не соответствует нумерации, принятой в `extract(dow from ...)`.

Функция `to_char(interval)` обрабатывает форматы HH и HH12 в рамках 12 часов, то есть 0 и 36 часов будут выводиться как 12, тогда как HH24 выводит значение полностью и для интервалов выводимое значение может превышать 23.

Коды форматирования числовых значений перечислены в таблице 1.26.

Таблица 1.26 – Коды форматирования чисел

Код	Описание
9	позиция цифры (может отсутствовать, если цифра незначащая)
0	позиция цифры (присутствует всегда, даже если цифра незначащая)
. (точка)	десятичная точка
, (запятая)	разделитель групп (тысяч)
PR	отрицательное значение в угловых скобках
S	знак, добавляемый к числу (с учетом локали)
L	символ денежной единицы (с учетом локали)
D	разделитель целой и дробной части числа (с учетом локали)
G	разделитель групп (с учетом локали)
MI	знак минус в заданной позиции (если число < 0)
PL	знак плюс в заданной позиции (если число > 0)
SG	знак плюс или минус в заданной позиции
RN	число римскими цифрами (в диапазоне от 1 до 3999)
TH или th	окончание порядкового числительного
V	сдвиг на заданное количество цифр (см. замечания)
EEEE	экспоненциальная запись числа

Замечания по использованию форматов чисел:

1. «0» обозначает позицию цифры, которая будет выводиться всегда, даже если это незначащий ноль слева или справа. 9 также обозначает позицию цифры, но если это незначащий ноль слева, он заменяется пробелом, а если справа и задан режим заполнения, он удаляется. Для функции `to_number()` эти два символа равнозначны.

2. Символы шаблона S, L, D и G представляют знак, символ денежной единицы, десятичную точку и разделитель тысяч, как их определяет текущая локаль. Символы точка и запятая представляют те же символы, обозначающие десятичную точку и разделитель тысяч, но не зависят от локали.

3. Если в шаблоне `to_char()` отсутствует явное указание положения знака, для него резервируется одна позиция рядом с числом (слева от него). Если левее нескольких 9 помещён S, знак также будет приписан слева к числу.
4. Знак числа, полученный кодами SG, PL или MI, не присоединяется к числу; например, `to_char(-12, 'MI9999')` выдает '- 12', тогда как `to_char(-12, 'S9999')` – '-12'. В Oracle MI не может идти перед 9, наоборот 9 нужно указать перед MI.
5. TH не преобразует значения меньше 0 и не поддерживает дробные числа.
6. PL, SG и TH – расширения Синергия-БД.
7. В `to_number` при использовании шаблонных кодов, не обозначающих данные, таких как L и TH, пропускается соответствующее количество входных символов. При этом не имеет значения, совпадают ли они с символами шаблона, если только это не символы данных (то есть цифры, знак числа, десятичная точка или запятая). Например, для подстроки TH будут пропущены два символа, не представляющие данные.
8. V по сути умножает вводимое значение на 10^n , где n – число цифр, следующих за V. Функция `to_char` не поддерживает V с дробными числами (например, 99.9V99 не допускается).
9. Код EEEE (научная запись) не может сочетаться с любыми другими вариантами форматирования или модификаторами, за исключением цифр и десятичной точки, и должен располагаться в конце строки шаблона (например, 9.99EEEE – допустимый шаблон).

Для изменения поведения кодов к ним могут быть применены определенные модификаторы. Например, FM9999 обрабатывается как код 9999 с модификатором FM. Все модификаторы для форматирования чисел перечислены в таблице 1.27.

Таблица 1.27 – Модификаторы шаблонов для форматирования чисел

Модификатор	Описание	Пример
Приставка FM	режим заполнения (подавляет завершающие нули и дополнение пробелами)	FM9999
Окончание TH	окончание порядкового числительного в верхнем регистре	999TH
Окончание th	окончание порядкового числительного в нижнем регистре	999th

В таблице 1.28 приведены некоторые примеры использования функции

to_char.

Таблица 1.28 – Примеры to char

Выражение	Результат
to_char(current_timestamp, 'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
to_char(-0.1, '99.99')	' -.10'
to_char(-0.1, 'FM9.99')	'-.1'
to_char(-0.1, 'FM90.99')	'-0.1'
to_char(0.1, '0.9')	' 0.1'
to_char(12, '9990999.9')	' 0012.0'
to_char(12, 'FM9990999.9')	'0012.'
to_char(485, '999')	' 485'
to_char(-485, '999')	'-485'
to_char(485, '9 9 9')	' 4 8 5'
to_char(1485, '9,999')	' 1,485'
to_char(1485, '9G999')	' 1 485'
to_char(148.5, '999.999')	' 148.500'
to_char(148.5, 'FM999.999')	'148.5'
to_char(148.5, 'FM999.990')	'148.500'
to_char(148.5, '999D999')	' 148,500'
to_char(3148.5, '9G999D999')	' 3 148,500'
to_char(-485, '999S')	'485-'
to_char(-485, '999MI')	'485-'
to_char(485, '999MI')	'485 '
to_char(485, 'FM999MI')	'485'
to_char(485, 'PL999')	' +485'
to_char(485, 'SG999')	' +485'
to_char(-485, 'SG999')	'-485'
to_char(-485, '9SG99')	'4-85'
to_char(-485, '999PR')	'<485>'
to_char(485, 'L999')	'DM 485'
to_char(485, 'RN')	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'
to_char(5.2, 'FMRN')	'V'
to_char(482, '999th')	' 482nd'
to_char(485, '«Good number:»999')	'Good number: 485'
to_char(485.8, '«Pre:»999» Post:» .999')	'Pre: 485 Post: .800'
to_char(12, '99V999')	' 12000'
to_char(12.4, '99V999')	' 12400'
to_char(12.45, '99V9')	' 125'
to_char(0.0004859, '9.99EEEE')	' 4.86e-04'

1.10.Операторы и функции даты/времени

Все существующие функции для обработки даты/времени перечислены в таблице 1.30, а подробнее они описаны в следующих подразделах. Поведение основных арифметических операторов (+, * и т.д.) описано в таблице 1.29. Функции форматирования этих типов данных были перечислены в разделе 1.8.

Все описанные ниже функции и операторы принимают две разновидности типов time или timestamp: с часовым поясом (time with time zone и timestamp with time zone) и без него (time without time zone и

timestamp without time zone). Для краткости здесь они рассматриваются вместе. Кроме того, операторы + и * обладают переместительным свойством (например, `date + integer = integer + date`); здесь будет приведен только один вариант для каждой пары.

Таблица 1.29 – Операторы даты/времени

Оператор	Пример	Результат
+	<code>date '2001-09-28' + integer '7'</code>	<code>date '2001-10-05'</code>
+	<code>date '2001-09-28' + interval '1 hour'</code>	<code>timestamp '2001-09-28 01:00:00'</code>
+	<code>date '2001-09-28' + time '03:00'</code>	<code>timestamp '2001-09-28 03:00:00'</code>
+	<code>interval '1 day' + interval '1 hour'</code>	<code>interval '1 day 01:00:00'</code>
+	<code>timestamp '2001-09-28 01:00' + interval '23 hours'</code>	<code>timestamp '2001-09-29 00:00:00'</code>
+	<code>time '01:00' + interval '3 hours'</code>	<code>time '04:00:00'</code>
-	<code>- interval '23 hours'</code>	<code>interval '-23:00:00'</code>
-	<code>date '2001-10-01' - date '2001-09-28'</code>	<code>integer '3' (дня)</code>
-	<code>date '2001-10-01' - integer '7'</code>	<code>date '2001-09-24'</code>
-	<code>date '2001-09-28' - interval '1 hour'</code>	<code>timestamp '2001-09-27 23:00:00'</code>
-	<code>time '05:00' - time '03:00'</code>	<code>interval '02:00:00'</code>
-	<code>time '05:00' - interval '2 hours'</code>	<code>time '03:00:00'</code>
-	<code>timestamp '2001-09-28 23:00' - interval '23 hours'</code>	<code>timestamp '2001-09-28 00:00:00'</code>
-	<code>interval '1 day' - interval '1 hour'</code>	<code>interval '1 day -01:00:00'</code>
-	<code>timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'</code>	<code>interval '1 day 15:00:00'</code>
*	<code>900 * interval '1 second'</code>	<code>interval '00:15:00'</code>
*	<code>21 * interval '1 day'</code>	<code>interval '21 days'</code>
*	<code>double precision '3.5' * interval '1 hour'</code>	<code>interval '03:30:00'</code>
/	<code>interval '1 hour' / double precision '1.5'</code>	<code>interval '00:40:00'</code>

Таблица 1.30 – Функции даты/времени

Функция	Тип результата	Описание	Пример	Результат
<code>age(timestamp, timestamp)</code>	interval	Вычитает аргументы и выдает «символический» результат с годами и месяцами, а не просто днями	<code>age(timestamp '2001-04-10', timestamp '1957-06-13')</code>	43 years 9 mons 27 days (43 года 9 месяцев 27 дней)
<code>age(timestamp)</code>	interval	Вычитает дату/время из <code>current_date</code> (полночь текущего дня)	<code>age(timestamp '1957-06-13')</code>	43 years 8 mons 3 days (43 года 8 месяцев 3 дня)
<code>clock_timestamp()</code>	timestamp with time zone	Текущая дата и время (меняется в процессе выполнения операторов); см. подраздел 1.10.4		
<code>current_date</code>	date	Текущая дата; см. подраздел 1.10.4		
<code>current_time</code>	time with time zone	Текущее время суток; см. подраздел 1.10.4		

Продолжение таблицы 1.30

Функция	Тип результата	Описание	Пример	Результат
current_timestamp	timestamp with time zone	Текущая дата и время (на момент начала транзакции); см. подраздел 1.10.4		
date_part(text, timestamp)	double precision	Возвращает поле даты (равнозначно extract); см. подраздел 1.10.1	date_part('hour', timestamp '2001-02-16 20:38:40')	20
date_part(text, interval)	double precision	Возвращает поле даты (равнозначно extract); см. подраздел 1.10.1	date_part('month', interval '2 years 3 months')	3
date_trunc(text, timestamp)	timestamp	Отсекает компоненты даты до заданной точности; см. также подраздел 1.10.2	date_trunc('hour', timestamp '2001-02-16 20:38:40')	2001-02-16 20:00:00
date_trunc(text, interval)	interval	Отсекает компоненты даты до заданной точности; см. также подраздел 1.10.2	date_trunc('hour', interval '2 days 3 hours 40 minutes')	2 days 03:00:00
extract(field from timestamp)	double precision	Возвращает поле даты; см. подраздел 1.10.1	extract(hour from timestamp '2001-02-16 20:38:40')	20
extract(field from interval)	double precision	Возвращает поле даты; см. подраздел 1.10.1	extract(month from interval '2 years 3 months')	3
isfinite(date)	boolean	Проверяет конечность даты (ее отличие от +/- бесконечности)	isfinite(date '2001-02-16')	true
isfinite(timestamp)	boolean	Проверяет конечность времени (его отличие от +/- бесконечности)	isfinite(timestamp '2001-02-16 21:28:30')	true
isfinite(interval)	boolean	Проверяет конечность интервала	isfinite(interval '4 hours')	true
justify_days(interval)	interval	Преобразует интервал так, что каждый 30-дневный период считается одним месяцем	justify_days(interval '35 days')	1 mon 5 days (1 месяц 5 дней)
justify_hours(interval)	interval	Преобразует интервал так, что каждый 24-часовой период считается одним днем	justify_hours(interval '27 hours')	1 day 03:00:00 (1 день 03:00:00)

Продолжение таблицы 1.30

Функция	Тип результата	Описание	Пример	Результат
<code>justify_interval(interval)</code>	interval	Преобразует интервал с применением <code>justify_days</code> и <code>justify_hours</code> и дополнительно корректирует знаки	<code>justify_interval(interval '1 mon -1 hour')</code>	29 days 23:00:00 (29 дней 23:00:00)
<code>localtime</code>	time	Текущее время суток; см. подраздел 1.10.4		
<code>localtimestamp</code>	timestamp	Текущая дата и время (на момент начала транзакции); см. подраздел 1.10.4		
<code>make_date(year int, month int, day int)</code>	date	Образует дату из полей: <code>year</code> (год), <code>month</code> (месяц) и <code>day</code> (день)	<code>make_date(2013, 7, 15)</code>	2013-07-15
<code>make_interval(years int DEFAULT 0, months int DEFAULT 0, weeks int DEFAULT 0, days int DEFAULT 0, hours int DEFAULT 0, mins int DEFAULT 0, secs double precision DEFAULT 0.0)</code>	interval	Образует интервал из полей: <code>years</code> (годы), <code>months</code> (месяцы), <code>weeks</code> (недели), <code>days</code> (дни), <code>hours</code> (часы), <code>minutes</code> (минуты) и <code>secs</code> (секунды)	<code>make_interval(days := 10)</code>	10 days
<code>make_time(hour int, min int, sec double precision)</code>	time	Образует время из полей: <code>hour</code> (час), <code>minute</code> (минута) и <code>sec</code> (секунда)	<code>make_time(8, 15, 23.5)</code>	08:15:23.5
<code>make_timestamp(year int, month int, day int, hour int, min int, sec double precision)</code>	timestamp	Образует дату и время из полей: <code>year</code> (год), <code>month</code> (месяц), <code>day</code> (день), <code>hour</code> (час), <code>minute</code> (минута) и <code>sec</code> (секунда)	<code>make_timestamp(2013, 7, 15, 8, 15, 23.5)</code>	2013-07-15 08:15:23.5
<code>make_timestamptz(year int, month int, day int, hour int, min int, sec double precision, [timezone text])</code>	timestamp with time zone	Образует дату и время с часовым поясом из полей: <code>year</code> (год), <code>month</code> (месяц), <code>day</code> (день), <code>hour</code> (час), <code>minute</code> (минута) и <code>sec</code> (секунда). Если параметр <code>timezone</code> (часовой пояс) не указан, используется текущий часовой пояс.	<code>make_timestamptz(2013, 7, 15, 8, 15, 23.5)</code>	2013-07-15 08:15:23.5+01
<code>now()</code>	timestamp with time zone	Текущая дата и время (на момент начала транзакции); см. подраздел 1.10.4		

Окончание таблицы 1.30

Функция	Тип результата	Описание	Пример	Результат
statement_timestamp()	timestamp with time zone	Текущая дата и время (на момент начала текущего оператора); см. подраздел 1.10.4		
timeofday()	text	Текущая дата и время (как clock_timestamp, но в виде строки типа text); см. подраздел 1.10.4		
transaction_timestamp()	timestamp with time zone	Текущая дата и время (на момент начала транзакции); см. подраздел 1.10.4		
to_timestamp(double precision)	timestamp with time zone	Преобразует время эпохи Unix (число секунд с 1970-01-01 00:00:00+00) в стандартное время	to_timestamp(1284352323)	2010-09-13 04:32:03+00

В дополнение к этим функциям поддерживается SQL-оператор OVERLAPS:

(начало1, конец1) OVERLAPS (начало2, конец2)

(начало1, длительность1) OVERLAPS (начало2, длительность2)

Его результатом будет true, когда два периода времени (определенные своими границами) пересекаются, и false в противном случае. Границы периода можно задать либо в виде пары дат, времени или дат со временем, либо как дату, время (или дату со временем) с интервалом. Когда указывается пара значений, первым может быть и начало, и конец периода: OVERLAPS автоматически считает началом периода меньшее значение. Периоды времени считаются наполовину открытыми, т.е. начало<=время<конец, если только начало и конец не равны – в этом случае период представляет один момент времени. Это означает, например, что два периода, имеющие только общую границу, не будут считаться пересекающимися.

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
      (DATE '2001-10-30', DATE '2002-10-30');
```

Результат:true

```
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
      (DATE '2001-10-30', DATE '2002-10-30');
```

Результат:false

```
SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
```

```
(DATE '2001-10-30', DATE '2001-10-31');
```

Результат:false

```
SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS  
(DATE '2001-10-30', DATE '2001-10-31');
```

Результат:true

При добавлении к дате со временем типа `timestamp with time zone` значения `interval` (или при вычитании из него `interval`), поле дней в этой дате увеличивается (или уменьшается) на указанное число дней. При пересечении границы перехода на летнее время (если в часовом поясе текущего сеанса производится этот переход) это означает, что `interval '1 day'` и `interval '24 hours'` не обязательно будут равны. Например, в часовом поясе `CST7CDT` результатом выражения `timestamp with time zone '2005-04-02 12:00-07' + interval '1 day'` будет `timestamp with time zone '2005-04-03 12:00-06'`, тогда как, если добавить `interval '24 hours'` к тому же значению `timestamp with time zone`, в результате получится `timestamp with time zone '2005-04-03 13:00-06'`. Эта разница объясняется тем, что 2005-04-03 02:00 в часовом поясе `CST7CDT` произошел переход на летнее время.

Обратите внимание на возможную неоднозначность в поле `months` в результате функции `age`, вызванную тем, что число дней в разных месяцах неодинаково. Вычисляя оставшиеся дни месяца, Синергия-БД рассматривает месяц меньшей из двух дат. Например, результатом `age ('2004-06-01', '2004-04-30')` будет `1 mon 1 day`, так как в апреле 30 дней, а то же выражение с датой 30 мая выдаст `1 mon 2 days`, так как в мае 31 день.

Вычитание дат и дат со временем также может быть нетривиальной операцией. Один принципиально простой способ выполнить такое вычисление – преобразовать каждое значение в количество секунд, используя `EXTRACT(EPOCH FROM ...)`, а затем найти разницу результатов; при этом будет получено число секунд между двумя датами. При этом будет учтено неодинаковое число дней в месяцах, изменения часовых поясов и переходы на летнее время. При вычитании дат или дат со временем с помощью оператора «-» выдается число дней (по 24 часа) и

часов/минут/секунд между данными значениями, с учетом тех же факторов. Функция `age` возвращает число лет, месяцев, дней и часов/минут/секунд, выполняя вычитание по полям, а затем пересчитывая отрицательные значения. Различие этих подходов иллюстрируют следующие запросы. Показанные результаты были получены для часового пояса 'US/Eastern'; между двумя заданными датами произошел переход на летнее время:

```
SELECT EXTRACT(EPOCH FROM timestampz '2013-07-01  
12:00:00') -
```

```
EXTRACT(EPOCH FROM timestampz '2013-03-01  
12:00:00');
```

Результат:10537200

```
SELECT (EXTRACT(EPOCH FROM timestampz '2013-07-01  
12:00:00') -
```

```
EXTRACT(EPOCH FROM timestampz '2013-03-01  
12:00:00'))
```

```
/ 60 / 60 / 24;
```

Результат:121.958333333333

```
SELECT timestampz '2013-07-01 12:00:00' - timestampz  
'2013-03-01 12:00:00';
```

Результат:121 days 23:00:00

```
SELECT age(timestampz '2013-07-01 12:00:00',  
timestampz '2013-03-01 12:00:00');
```

Результат:4 mons

1.10.1. EXTRACT, date_part

```
EXTRACT(field FROM source)
```

Функция `extract` получает из значений даты/времени поля, такие как год или час. Здесь источник – значение типа `timestamp`, `time` или `interval`. (Выражения типа `date` приводятся к типу `timestamp`, так что допускается и этот тип.) Указанное поле представляет собой идентификатор, по которому из источника выбирается заданное поле. Функция `extract` возвращает значения типа `double`

precision. Допустимые поля:

century

Век:

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16  
12:21:13');
```

Результат:20

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16  
20:38:40');
```

Результат:21

Первый век начался 0001-01-01 00:00:00, хотя люди в то время и не считали так. Это определение распространяется на все страны с григорианским календарем. Века с номером 0 не было; считается, что 1 наступил после -1.

day

Для значений timestamp это день месяца (1 - 31), для значений interval – число дней.

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16  
20:38:40');
```

Результат:16

```
SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
```

Результат:40

decade

Год, деленный на 10.

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16  
20:38:40');
```

Результат:200

dow

День недели, считая с воскресенья (0) до субботы (6).

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16  
20:38:40');
```

Результат:5

Заметьте, что в extract дни недели нумеруются не так, как в функции to_char(..., 'D').

doy

День года (1 - 365/366).

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16  
20:38:40');
```

Результат:47.

epoch

Для значений timestamp with time zone это число секунд с 1970-01-01 00:00:00 UTC (может быть отрицательным); для значений date и timestamp это число секунд с 1970-01-01 00:00:00 по местному времени, а для interval – общая длительность интервала в секундах.

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE  
'2001-02-16 20:38:40.12-08');
```

Результат:982384720.12

```
SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
```

Результат:442800

Преобразовать время эпохи назад, в значение дата/время можно так:

```
SELECT to_timestamp(982384720.12);
```

Результат:2001-02-17 04:38:40.12+00

hour

Час (0 - 23).

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16  
20:38:40');
```

Результат:20

isodow

День недели, считая с понедельника (1) до воскресенья (7).

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18  
20:38:40');
```

Результат:7

Результат отличается от dow только для воскресенья. Такая нумерация соответствует ISO 8601.

isoyear

Год по недельному календарю ISO 8601, в который попадает дата (неприменимо к интервалам).

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
```

Результат:2005

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
```

Результат:2006

Год по недельному календарю ISO начинается с понедельника недели, в которой оказывается 4 января, так что в начале января или в конце декабря год по ISO может отличаться от года по григорианскому календарю. Подробнее об этом рассказывается в описании поля `week`.

`microseconds`

Значение секунд с дробной частью, умноженное на 1 000 000; заметьте, что оно включает и целые секунды.

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
```

Результат:28500000

`millennium`

Тысячелетие.

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16  
20:38:40');
```

Результат:3

Годы 20 века относятся ко второму тысячелетию. Третье тысячелетие началось 1 января 2001 г.

`milliseconds`

Значение секунд с дробной частью, умноженное на 1 000; заметьте, что оно включает и целые секунды.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
```

Результат:28500

`minute`

Минуты (0 - 59).

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16  
20:38:40');
```

Результат:38

`month`

Для значений `timestamp` это номер месяца в году (1 - 12), а для `interval` – остаток от деления числа месяцев на 12 (в интервале 0 - 11).

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16  
20:38:40');
```

Результат:2

```
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
```

Результат:3

```
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
```

Результат:1

quarter

Квартал года (1 - 4), к которому относится дата.

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16  
20:38:40');
```

Результат:1

second

Секунды, включая дробную часть (0 - 59).

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16  
20:38:40');
```

Результат:40

```
SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
```

Результат:28.5

timezone

Смещение часового пояса от UTC, представленное в секундах.

Положительные значения соответствуют часовым поясам к востоку от UTC, а отрицательные – к западу. Строго говоря, в Синергия-БД используется не UTC, так как секунды координации не учитываются.

timezone_hour

Поле часов в смещении часового пояса.

timezone_minute

Поле минут в смещении часового пояса.

week

Номер недели в году по недельному календарю ISO 8601. По определению, недели ISO 8601 начинаются с понедельника, а первая неделя года включает 4

января этого года. Другими словами, первый четверг года всегда оказывается в 1 неделе этого года.

В системе нумерации недель ISO первые числа января могут относиться к 52-ой или 53-ей неделе предыдущего года, а последние числа декабря – к первой неделе следующего года. Например, 2005-01-01 относится к 53-ей неделе 2004 г., а 2006-01-01 – к 52-ей неделе 2005 г., тогда как 2012-12-31 включается в первую неделю 2013 г. Поэтому для получения согласованных результатов рекомендуется использовать поле `isoyear` в паре с `week`.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16  
20:38:40');
```

Результат:7

`year`

Поле года. Учтите, что года «0» не было, и это следует иметь в виду, вычитая из годов нашей эры годы до нашей эры.

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16  
20:38:40');
```

Результат:2001

Примечание. С аргументом +/-бесконечность `extract` возвращает +/-бесконечность для монотонно увеличивающихся полей (`epoch`, `julian`, `year`, `isoyear`, `decade`, `century` и `millennium`). Для других полей возвращается NULL. В версии СУБД «Синергия-БД 1» возвращался ноль для всех случаев с бесконечными аргументами.

Функция `extract` в основном предназначена для вычислительных целей. Функции форматирования даты/времени были описаны в разделе 1.8.

Функция `date_part` эмулирует традиционный для Ingres эквивалент стандартной SQL-функции `extract`:

```
date_part('поле', источник)
```

Заметьте, что здесь параметр поле должен быть строковым значением, а не именем. Функция `date_part` воспринимает те же поля, что и `extract`.

```
SELECT date_part('day', TIMESTAMP '2001-02-16  
20:38:40');
```

Результат:16

```
SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
```

Результат: 4

1.10.2. date_trunc

Функция `date_trunc` работает подобно `trunc` для чисел.

```
date_trunc('поле', значение)
```

Здесь значение – это выражение типа `timestamp` или `interval`. Значения типов `date` и `time` автоматически приводятся к типам `timestamp` и `interval`, соответственно. Параметр `поле` определяет, до какой точности обрезать переданное значение. Возвращаемое значение будет иметь тип `timestamp` или `interval` и все его значения, менее значимые, чем заданное поле, будут равны нулю (или единице, если это номер дня или месяца).

Параметр `поле` может принимать следующие значения:

`microseconds`

`milliseconds`

`second`

`minute`

`hour`

`day`

`week`

`month`

`quarter`

`year`

`decade`

`century`

`millennium`

Примеры:

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16  
20:38:40');
```

Результат: 2001-02-16 20:00:00

```
SELECT date_trunc('year', TIMESTAMP '2001-02-16
```

20:38:40');

Результат: 2001-01-01 00:00:00

1.10.3. AT TIME ZONE

Указание AT TIME ZONE позволяет переводить дату/время *без часового пояса* в дату/время *с часовым поясом* и обратно, а также пересчитывать значения *времени* для различных часовых поясов. Все разновидности этого указания проиллюстрированы в таблице 1.31.

Таблица 1.31 – Разновидности AT TIME ZONE

Выражение	Тип результата	Описание
timestamp without time zone AT TIME ZONE часовой_пояс	timestamp with time zone	Воспринимает заданное время без указания часового пояса как время в указанном часовом поясе
timestamp with time zone AT TIME ZONE часовой_пояс	timestamp without time zone	Переводит данное значение timestamp с часовым поясом в другой часовой пояс, но не сохраняет информацию о нем в результате
time with time zone AT TIME ZONE часовой_пояс	time with time zone	Переводит данное время с часовым поясом в другой часовой пояс

В этих выражениях желаемый часовой_пояс можно задать либо в виде текстовой строки (например, 'America/Los_Angeles'), либо как интервал (например, INTERVAL '-08:00').

Примеры (в предположении, что местный часовой пояс America/Los_Angeles):

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE
'America/Denver';
```

Результат: 2001-02-16 19:38:40-08

```
SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05'
AT TIME ZONE 'America/Denver';
```

Результат: 2001-02-16 18:38:40

```
SELECT TIMESTAMP '2001-02-16 20:38:40-05' AT TIME ZONE
'Asia/Tokyo' AT TIME ZONE 'America/Chicago';
```

Результат: 2001-02-16 05:38:40

В первом примере для значения, заданного без часового пояса, указывается часовой пояс и полученное время выводится в текущем часовом поясе (заданном параметром TimeZone). Во втором примере значение времени смещается в заданный часовой пояс и выдаётся без указания часового пояса. Этот вариант позволяет

хранить и выводить значения с часовым поясом, отличным от текущего. В третьем примере время в часовом поясе Токио пересчитывается для часового пояса Чикаго. При переводе значений времени без даты в другие часовые пояса используются определения часовых поясов, действующие в данный момент.

Функция `timezone` (часовой_пояс, время) равнозначна SQL-совместимой конструкции `время AT TIME ZONE часовой_пояс`.

1.10.4. Текущая дата/время

Синергия-БД предоставляет набор функций, результат которых зависит от текущей даты и времени. Все следующие функции соответствуют стандарту SQL и возвращают значения, отражающие время начала текущей транзакции:

```
CURRENT_DATE  
CURRENT_TIME  
CURRENT_TIMESTAMP  
CURRENT_TIME (точность)  
CURRENT_TIMESTAMP (точность)  
LOCALTIME  
LOCALTIMESTAMP  
LOCALTIME (точность)  
LOCALTIMESTAMP (точность)
```

`CURRENT_TIME` и `CURRENT_TIMESTAMP` возвращают время с часовым поясом. В результатах `LOCALTIME` и `LOCALTIMESTAMP` нет информации о часовом поясе.

`CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME` и `LOCALTIMESTAMP` могут принимать необязательный параметр точности, определяющий, до какого знака после запятой следует округлять поле секунд. Если этот параметр отсутствует, результат будет иметь максимально возможную точность.

Несколько примеров:

```
SELECT CURRENT_TIME;
```

Результат: 14:39:53.662522-05

```
SELECT CURRENT_DATE;
```

Результат: 2001-12-23

```
SELECT CURRENT_TIMESTAMP;
```

Результат: 2001-12-23 14:39:53.662522-05

```
SELECT CURRENT_TIMESTAMP(2);
```

Результат: 2001-12-23 14:39:53.66-05

```
SELECT LOCALTIMESTAMP;
```

Результат: 2001-12-23 14:39:53.662522

Так как эти функции возвращают время начала текущей транзакции, во время транзакции эти значения не меняются. Это считается не ошибкой, а особенностью реализации: цель такого поведения в том, чтобы в одной транзакции «текущее» время было одинаковым, и для разных изменений в одной транзакции записывалась одна отметка времени.

В Синергия-БД есть также функции, возвращающие время начала текущего оператора, а также текущее время в момент вызова функции. Таким образом, в Синергия-БД есть следующие функции, не описанные в стандарте SQL:

```
transaction_timestamp()  
statement_timestamp()  
clock_timestamp()  
timeofday()  
now()
```

Функция `transaction_timestamp()` равнозначна конструкции `CURRENT_TIMESTAMP`, но в её названии явно отражено, что она возвращает. Функция `statement_timestamp()` возвращает время начала текущего оператора (более точно, время получения последнего командного сообщения от клиента). Функции `statement_timestamp()` и `transaction_timestamp()` возвращают одно и то же значение в первой команде транзакции, но в последующих их показания будут расходиться. Функция `clock_timestamp()` возвращает фактическое текущее время, так что её значение меняется в рамках одной команды SQL. Функция `timeofday()` существует в Синергия-БД по историческим

причинам и, подобно `clock_timestamp()`, она возвращает фактическое текущее время, но представленное в виде форматированной строки типа `text`, а не значения `timestamp with time zone`. Функция `now()` – традиционный для Синергия-БД эквивалент функции `transaction_timestamp()`.

Все типы даты/времени также принимают специальное буквальное значение `now`, подразумевающее текущую дату и время (тоже на момент начала транзакции). Таким образом, результат следующих трех операторов будет одинаковым:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';  -- не подходит для DEFAULT
```

Примечание. Третья форма не подходит для указания в качестве значения `DEFAULT` при создании таблицы. Система преобразует `now` в значение `timestamp` в момент разбора константы, поэтому, когда будет вставляться значение по умолчанию, в соответствующей колонке окажется время создания таблицы! Первые две формы не будут вычисляться, пока не потребуется значение по умолчанию, так как это вызовы функции. Поэтому они дадут желаемый результат при добавлении строки в таблицу.

1.10.5. Задержка выполнения

В случае необходимости вы можете приостановить выполнение серверного процесса, используя следующие функции:

```
pg_sleep(сек)
pg_sleep_for(interval)
pg_sleep_until(timestamp with time zone)
```

Функция `pg_sleep` переводит процесс текущего сеанса в спящее состояние на указанное число секунд (`сек`). Параметр `сек` имеет тип `double precision`, так что в нем можно указать и дробное число. Функция `pg_sleep_for` введена для удобства, ей можно передать большие значения задержки в типе `interval`. А `pg_sleep_until` удобнее использовать, когда необходимо задать определенное время выхода из спящего состояния, например:

```
SELECT pg_sleep(1.5);
SELECT pg_sleep_for('5 minutes');
SELECT pg_sleep_until('tomorrow 03:00');
```

Примечание. Действительное разрешение интервала задержки зависит от

платформы; обычно это 0.01. Фактическая длительность задержки не будет меньше указанного времени, но может быть больше, в зависимости, например от нагрузки на сервер. В частности, не гарантируется, что `pg_sleep_until` проснется именно в указанное время, но она точно не проснется раньше.

1.11. Функции для перечислений

Для типов перечислений предусмотрено несколько функций, которые позволяют сделать код чище, не «зашивая» в нем конкретные значения перечисления. Эти функции перечислены в таблице 1.32. В этих примерах подразумевается, что перечисление создано так:

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow',
    'green',
    'blue', 'purple');
```

Таблица 1.32 – Функции для перечислений

Функция	Описание	Пример	Результат примера
<code>enum_first(anyenum)</code>	Возвращает первое значение заданного перечисления	<code>enum_first(null::rainbow)</code>	red
<code>enum_last(anyenum)</code>	Возвращает последнее значение заданного перечисления	<code>enum_last(null::rainbow)</code>	purple
<code>enum_range(anyenum)</code>	Возвращает все значения заданного перечисления в упорядоченном массиве	<code>enum_range(null::rainbow)</code>	{red, orange, yellow, green, blue, purple}
<code>enum_range(anyenum, anyenum)</code>	Возвращает набор значений, лежащих между двумя заданными, в виде упорядоченного массива. Эти значения должны принадлежать одному перечислению. Если первый параметр равен NULL, функция возвращает первое значение перечисления, а если NULL второй – последнее.	<code>enum_range('orange'::rainbow, 'green'::rainbow)</code>	{orange, yellow, green}
		<code>enum_range(NULL, 'green'::rainbow)</code>	{red, orange, yellow, green}
		<code>enum_range('orange'::rainbow, NULL)</code>	{orange, yellow, green, blue, purple}

Заметьте, что за исключением варианта `enum_range` с двумя аргументами, эти функции не обращают внимание на конкретное переданное им значение; их интересует только объявленный тип. Они возвращают один и тот же результат, когда им передается NULL или любое другое значение типа. Обычно эти функции применяются к колонкам таблицы или аргументам внешних функций, а не к предопределенным типам, как показано в этих примерах.

1.12.Геометрические функции и операторы

Для геометрических типов `point`, `box`, `lseg`, `line`, `path`, `polygon` и `circle` разработан большой набор встроенных функций и операторов, представленный в таблице 1.33, таблице 1.34 и таблице 1.35.

Примечание. Заметьте, что оператор «идентичности», `~=`, представляет обычное сравнение на равенство значений `point`, `box`, `polygon` и `circle`. Для некоторых из этих типов определен также оператор `=`, но `=` проверяет только равенство площадей. Другие скалярные операторы сравнения (`<=` и т.д.) так же сравнивают площади значений этих типов.

Таблица 1.33 – Геометрические операторы

Оператор	Описание	Пример
<code>+</code>	Сдвиг	<code>box '((0,0),(1,1))' + point '(2.0,0)'</code>
<code>-</code>	Сдвиг	<code>box '((0,0),(1,1))' - point '(2.0,0)'</code>
<code>*</code>	Масштабирование/поворот	<code>box '((0,0),(1,1))' * point '(2.0,0)'</code>
<code>/</code>	Масштабирование/поворот	<code>box '((0,0),(2,2))' / point '(2.0,0)'</code>
<code>#</code>	Точка или прямоугольник в пересечении	<code>'((1,-1),(-1,1))' # '((1,1),(-1,-1))'</code>
<code>#</code>	Число точек в пути или вершин в многоугольнике	<code># '((1,0),(0,1),(-1,0))'</code>
<code>@-@</code>	Длина, периметр или длина окружности	<code>@-@ path '((0,0),(1,0))'</code>
<code>@@</code>	Центр	<code>@@ circle '((0,0),10)'</code>
<code>##</code>	Точка, ближайшая к первому операнду и принадлежащая второму	<code>point '(0,0)' ## lseg '((2,0),(0,2))'</code>
<code><-></code>	Расстояние между операндами	<code>circle '((0,0),1)' <-> circle '((5,0),1)'</code>
<code>&&</code>	Пересекаются ли операнды? (Для положительного ответа достаточно одной общей точки.)	<code>box '((0,0),(1,1))' && box '((0,0),(2,2))'</code>
<code><<</code>	Строго слева?	<code>circle '((0,0),1)' << circle '((5,0),1)'</code>
<code>>></code>	Строго справа?	<code>circle '((5,0),1)' >> circle '((0,0),1)'</code>
<code>&<</code>	Не простирается правее?	<code>box '((0,0),(1,1))' &< box '((0,0),(2,2))'</code>
<code>&></code>	Не простирается левее?	<code>box '((0,0),(3,3))' &> box '((0,0),(2,2))'</code>
<code><< </code>	Строго ниже?	<code>box '((0,0),(3,3))' << box '((3,4),(5,5))'</code>
<code> >></code>	Строго выше?	<code>box '((3,4),(5,5))' >> box '((0,0),(3,3))'</code>
<code>&< </code>	Не простирается выше?	<code>box '((0,0),(1,1))' &< box '((0,0),(2,2))'</code>
<code> &></code>	Не простирается ниже?	<code>box '((0,0),(3,3))' &> box '((0,0),(2,2))'</code>
<code><^</code>	Ниже (может касаться)?	<code>circle '((0,0),1)' <^ circle '((0,5),1)'</code>
<code>>^</code>	Выше (может касаться)?	<code>circle '((0,5),1)' >^ circle '((0,0),1)'</code>
<code>?#</code>	Пересекает?	<code>lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'</code>
<code>?-</code>	Горизонтальный объект?	<code>?- lseg '((-1,0),(1,0))'</code>
<code>?-</code>	Выворочены по горизонтали?	<code>point '(1,0)' ?- point '(0,0)'</code>

Окончание таблицы 1.33

Оператор	Описание	Пример
?	Вертикальный объект?	? lseg '((-1,0),(1,0))'
?	Выровнены по вертикали?	point '(0,1)' ? point '(0,0)'
?-	Перпендикулярны?	lseg '((0,0),(0,1))' ?- lseg '((0,0),(1,0))'
?	Параллельны?	lseg '((-1,0),(1,0))' ? lseg '((-1,2),(1,2))'
@>	Первый объект включает второй?	circle '((0,0),2)' @> point '(1,1)'
<@	Первый объект включен во второй?	point '(1,1)' <@ circle '((0,0),2)'
~=	Одинаковы?	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

Таблица 1.34 – Геометрические функции

Функция	Тип результата	Описание	Пример
area(объект)	double precision	площадь	area(box '((0,0),(1,1))')
center(объект)	point	центр	center(box '((0,0),(1,2))')
diameter(circle)	double precision	диаметр круга	diameter(circle '((0,0),2.0)')
height(box)	double precision	вертикальный размер прямоугольника	height(box '((0,0),(1,1))')
isclosed(path)	boolean	замкнутый путь?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	открытый путь?	isopen(path '[(0,0),(1,1),(2,0)]')
length(объект)	double precision	длина	length(path '((-1,0),(1,0))')
npoints(path)	int	число точек	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	int	число точек	npoints(polygon '((1,1),(0,0))')
pclose(path)	path	преобразует путь в замкнутый	pclose(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	преобразует путь в открытый	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	радиус окружности	radius(circle '((0,0),2.0)')
width(box)	double precision	горизонтальный размер прямоугольника	width(box '((0,0),(1,1))')

Таблица 1.35 – Функции преобразования геометрических типов

Функция	Тип результата	Описание	Пример
box(circle)	box	окружность в прямоугольник	box(circle '((0,0),2.0)')
box(point)	box	точка в пустой прямоугольник	box(point '(0,0)')
box(point, point)	box	точки в прямоугольник	box(point '(0,0)', point '(1,1)')
box(polygon)	box	многоугольник в прямоугольник	box(polygon '((0,0),(1,1),(2,0))')
bound_box(box, box)	box	прямоугольники в окружающий прямоугольник	bound_box(box '((0,0),(1,1))', box '((3,3),(4,4))')
circle(box)	circle	прямоугольник в окружность	circle(box '((0,0),(1,1))')

Окончание таблицы 1.35

Функция	Тип результата	Описание	Пример
circle(point, double precision)	circle	окружность из центра и радиуса	circle(point '(0,0)', 2.0)
circle(polygon)	circle	многоугольник в окружность	circle(polygon '((0,0),(1,1),(2,0))')
line(point, point)	line	точки в прямую	line(point '(-1,0)', point '(1,0)')
lseg(box)	lseg	диагональ прямоугольника в отрезок	lseg(box '((-1,0),(1,0))')
lseg(point, point)	lseg	точки в отрезок	lseg(point '(-1,0)', point '(1,0)')
path(polygon)	path	многоугольник в путь	path(polygon '((0,0),(1,1),(2,0))')
point(double precision, double precision)	point	образует точку	point(23.4, -44.5)
point(box)	point	центр прямоугольника	point(box '((-1,0),(1,0))')
point(circle)	point	центр окружности	point(circle '((0,0),2.0)')
point(lseg)	point	центр отрезка	point(lseg '((-1,0),(1,0))')
point(polygon)	point	центр многоугольника	point(polygon '((0,0),(1,1),(2,0))')
polygon(box)	polygon	прямоугольник в многоугольник с 4 вершинами	polygon(box '((0,0),(1,1))')
polygon(circle)	polygon	круг в многоугольник с 12 вершинами	polygon(circle '((0,0),2.0)')
polygon(число_точек, circle)	polygon	окружность с заданным числом точек	polygon(12, circle '((0,0),2.0)')
polygon(path)	polygon	путь в многоугольник	polygon(path '((0,0),(1,1),(2,0))')

К двум компонентам типа point (точка) можно обратиться, как к элементам массива с индексами 0 и 1. Например, если t.p – колонка типа point, SELECT p[0] FROM t вернет координату X, а UPDATE t SET p[1] = ... изменит координату Y. Таким же образом, значение типа box или lseg можно воспринимать как массив двух значений типа point.

Функция area работает с типами box, circle и path. При этом для типа path заданный путь не должен быть самопересекающимся. Например, эта функция не примет значение типа path '((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))'::PATH, но примет визуально идентичный путь '((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))'::PATH. Если вы не вполне поняли, что здесь подразумевается под самопересечением пути, нарисуйте на бумаге две фигуры по приведенным координатам.

1.13. Функции и операторы для работы с сетевыми адресами

В таблице 1.36 показаны операторы, работающие с типами `cidr` и `inet`. Операторы `<<`, `<=<`, `>>`, `>>=` и `&&` проверяют включения подсетей, рассматривая только биты сети в обоих адресах (игнорируя биты узлов) и определяя, идентична ли одна сеть другой или её подсети.

Таблица 1.36 – Операторы для типов `cidr` и `inet`

Оператор	Описание	Пример
<code><</code>	меньше	<code>inet '192.168.1.5' < inet '192.168.1.6'</code>
<code><=</code>	меньше или равно	<code>inet '192.168.1.5' <= inet '192.168.1.5'</code>
<code>=</code>	равно	<code>inet '192.168.1.5' = inet '192.168.1.5'</code>
<code>>=</code>	больше или равно	<code>inet '192.168.1.5' >= inet '192.168.1.5'</code>
<code>></code>	больше	<code>inet '192.168.1.5' > inet '192.168.1.4'</code>
<code><></code>	не равно	<code>inet '192.168.1.5' <> inet '192.168.1.4'</code>
<code><<</code>	содержится в	<code>inet '192.168.1.5' << inet '192.168.1/24'</code>
<code><=<</code>	первая сеть равна или содержится во второй	<code>inet '192.168.1/24' <=< inet '192.168.1/24'</code>
<code>>></code>	первая сеть содержит вторую	<code>inet '192.168.1/24' >> inet '192.168.1.5'</code>
<code>>>=</code>	первая сеть содержит вторую или равна ей	<code>inet '192.168.1/24' >>= inet '192.168.1/24'</code>
<code>&&</code>	первая сеть содержит вторую или содержится в ней	<code>inet '192.168.1/24' && inet '192.168.1.80/28'</code>
<code>~</code>	битовый NOT	<code>~ inet '192.168.1.6'</code>
<code>&</code>	битовый AND	<code>inet '192.168.1.6' & inet '0.0.0.255'</code>
<code> </code>	битовый OR	<code>inet '192.168.1.6' inet '0.0.0.255'</code>
<code>+</code>	сложение	<code>inet '192.168.1.6' + 25</code>
<code>-</code>	вычитание	<code>inet '192.168.1.43' - 36</code>
<code>-</code>	вычитание	<code>inet '192.168.1.43' - inet '192.168.1.19'</code>

В таблице 1.37 перечислены функции, работающие с типами `cidr` и `inet`. Функции `abbrev`, `host` и `text` предназначены в основном для вывода данных в альтернативных форматах.

Таблица 1.37 – Функции для типов `cidr` и `inet`

Функция	Тип результата	Описание	Пример	Результат
<code>abbrev(inet)</code>	<code>text</code>	вывод адрес в кратком текстовом виде	<code>abbrev(inet '10.1.0.0/16')</code>	<code>10.1.0.0/16</code>
<code>abbrev(cidr)</code>	<code>text</code>	вывод адрес в кратком текстовом виде	<code>abbrev(cidr '10.1.0.0/16')</code>	<code>10.1/16</code>
<code>broadcast(inet)</code>	<code>inet</code>	широковещательный адрес сети	<code>broadcast('192.168.1.5/24')</code>	<code>192.168.1.255/24</code>

Окончание таблицы 1.37

Функция	Тип результата	Описание	Пример	Результат
family(inet)	int	возвращает семейство адреса; 4 для адреса IPv4, 6 для IPv6	family('::1')	6
host(inet)	text	извлекает IP-адрес в виде текста	host('192.168.1.5/24')	192.168.1.5
hostmask(inet)	inet	вычисляет маску узла для сетевого адреса	hostmask('192.168.23.20/30')	0.0.0.3
masklen(inet)	int	выдает длину маски сети	masklen('192.168.1.5/24')	24
netmask(inet)	inet	вычисляет маску сети для сетевого адреса	netmask('192.168.1.5/24')	255.255.255.0
network(inet)	cidr	извлекает компонент сети из адреса	network('192.168.1.5/24')	192.168.1.0/24
set_masklen(inet, int)	inet	задает размер маски для значения inet	set_masklen('192.168.1.5/24', 16)	192.168.1.5/16
set_masklen(cidr, int)	cidr	задает размер маски для значения cidr	set_masklen('192.168.1.0/24'::cidr, 16)	192.168.0.0/16
text(inet)	text	выводит в текстовом виде IP-адрес и длину маски	text(inet '192.168.1.5')	192.168.1.5/32
inet_same_family(inet, inet)	boolean	адреса относятся к одному семейству?	inet_same_family('192.168.1.5/24', ':::1')	false
inet_merge(inet, inet)	cidr	наименьшая сеть, включающая обе заданные сети	inet_merge('192.168.1.5/24', '192.168.2.5/24')	192.168.0.0/22

Любое значение `cidr` можно привести к типу `inet`, явно или нет; поэтому все функции, показанные выше с типом `inet`, также будут работать со значениями `cidr`. Некоторые из функций указаны отдельно для типов `inet` и `cidr`, потому что их поведение с разными типами различается. Кроме того, значение `inet` тоже можно привести к типу `cidr`. При этом все биты справа от сетевой маски просто обнуляются, чтобы значение стало допустимым для типа `cidr`. К типам `inet` и `cidr` можно привести и обычные текстовые значения, используя обычный синтаксис, например: `inet(выражение)` или колонка `::cidr`.

В таблице 1.38 приведена функция, предназначенная для работы с типом `macaddr`. Функция `trunc(macaddr)` возвращает MAC-адрес, последние 3 байта в котором равны «0». Это может быть полезно для вычисления префикса, определяющего производителя.

Таблица 1.38 – Функции `macaddr`

Функция	Тип результата	Описание	Пример	Результат
<code>trunc(macaddr)</code>	<code>macaddr</code>	обнуляет последние 3 байта	<code>trunc(macaddr '12:34:56:78:90:ab')</code>	12:34:56:00:00:00

Тип `macaddr` также поддерживает стандартные реляционные операторы лексической сортировки (`>`, `<=` и т.д.) и операторы битовой арифметики (`~`, `&` и `|`), соответствующие операциям NOT, AND и OR.

1.14. Функции и операторы текстового поиска

В таблице 1.39, таблице 1.40 и таблице 1.41 собраны все существующие функции и операторы, предназначенные для полнотекстового поиска.

Таблица 1.39 – Операторы текстового поиска

Оператор	Описание	Пример	Результат
<code>@@</code>	<code>tsvector</code> соответствует <code>tsquery</code> ?	<code>to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat')</code>	t
<code>@@@</code>	устаревший синоним для <code>@@</code>	<code>to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat')</code>	t
<code> </code>	объединяет два значения <code>tsvector</code>	<code>'a:1 b:2'::tsvector 'c:1 d:2 b:3'::tsvector</code>	'a':1 'b':2,5 'c':3 'd':4
<code>&&</code>	логическое И (AND) двух запросов <code>tsquery</code>	<code>'fat rat'::tsquery && 'cat'::tsquery</code>	('fat' 'rat') & 'cat'
<code> </code>	логическое ИЛИ (OR) двух запросов <code>tsquery</code>	<code>'fat rat'::tsquery 'cat'::tsquery</code>	('fat' 'rat') 'cat'
<code>!!</code>	отрицание запроса <code>tsquery</code>	<code>!! 'cat'::tsquery</code>	!'cat'
<code>@></code>	запрос <code>tsquery</code> включает другой?	<code>'cat'::tsquery @> 'cat & rat'::tsquery</code>	f
<code><@</code>	запрос <code>tsquery</code> включен в другой?	<code>'cat'::tsquery <@ 'cat & rat'::tsquery</code>	t

Примечание. Операторы включения `tsquery` рассматривают только лексемы двух запросов, игнорируя операторы их сочетания.

В дополнение к операторам, перечисленным в этой таблице, для типов `tsvector` и `tsquery` определены обычные операторы сравнения для В-дерева (`=`, `<` и т.д.). Они не очень полезны для поиска, но позволяют, в частности, создавать индексы для колонок этих типов.

Таблица 1.40 – Функции текстового поиска

Функция	Тип результата	Описание	Пример	Результат
<code>array_to_tsvector(text[])</code>	<code>tsvector</code>	преобразует массив лексем в <code>tsvector</code>	<code>array_to_tsvector('{fat,cat,rat}'::text[])</code>	'cat' 'fat' 'rat'

Продолжение таблицы 1.40

Функция	Тип результата	Описание	Пример	Результат
<code>get_current_ts_config()</code>	<code>regconfig</code>	получает конфигурацию текстового поиска по умолчанию	<code>get_current_ts_config()</code>	<code>english</code>
<code>length(tsvector)</code>	<code>integer</code>	число лексем в значении <code>tsvector</code>	<code>length('fat:2,4 cat:3 rat:5A'::tsvector)</code>	<code>3</code>
<code>numnode(tsquery)</code>	<code>integer</code>	число лексем и операторов в запросе <code>tsquery</code>	<code>numnode('(fat & rat) cat'::tsquery)</code>	<code>5</code>
<code>plainto_tsquery([конфигурацияregconfig,] запрос text)</code>	<code>tsquery</code>	выдаёт значение <code>tsquery</code> , игнорируя пунктуацию	<code>plainto_tsquery('english', 'The Fat Rats')</code>	<code>'fat' & 'rat'</code>
<code>phraseto_tsquery([конфигурацияregconfig,] запрос text)</code>	<code>tsquery</code>	выдаёт значение <code>tsquery</code> для поиска фразы, игнорируя пунктуацию	<code>phraseto_tsquery('english', 'The Fat Rats')</code>	<code>'fat' <-> 'rat'</code>
<code>websearch_to_tsquery([конфигурацияregconfig,] запрос text)</code>	<code>tsquery</code>	преобразует в значение <code>tsquery</code> поисковый запрос в веб-стиле	<code>websearch_to_tsquery('english', '"fat rat" or rat')</code>	<code>'fat' <-> 'rat' 'rat'</code>
<code>querytree(запрос tsquery)</code>	<code>text</code>	получает индексируемую часть запроса <code>tsquery</code>	<code>querytree('foo & ! bar'::tsquery)</code>	<code>'foo'</code>
<code>setweight(вектор tsvector, вес "char")</code>	<code>tsvector</code>	назначает вес каждому элементу вектора	<code>setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A')</code>	<code>'cat':3A 'fat':2A,4A 'rat':5A</code>
<code>setweight(вектор tsvector, вес "char", лексемы text[])</code>	<code>tsvector</code>	назначает вес элементам вектора, перечисленным в массиве лексемы	<code>setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A', '{cat,rat}')</code>	<code>'cat':3A 'fat':2,4 'rat':5A</code>
<code>strip(tsvector)</code>	<code>tsvector</code>	убирает позиции и веса из значения <code>tsvector</code>	<code>strip('fat:2,4 cat:3 rat:5A'::tsvector)</code>	<code>'cat' 'fat' 'rat'</code>
<code>to_tsquery([конфигурацияregconfig,] запрос text)</code>	<code>tsquery</code>	нормализует слова и переводит их в <code>tsquery</code>	<code>to_tsquery('english', 'The & Fat & Rats')</code>	<code>'fat' & 'rat'</code>
<code>to_tsvector([конфигурацияregconfig,] документ text)</code>	<code>tsvector</code>	сокращает текст документа до значения <code>tsvector</code>	<code>to_tsvector('english', 'The Fat Rats')</code>	<code>'fat':2 'rat':3</code>
<code>to_tsvector([конфигурацияregconfig,] документ json(b))</code>	<code>tsvector</code>	сокращает каждое строковое значение в документе до значения <code>tsvector</code> , а затем складывает эти значения по порядку в документе и выдаёт один <code>tsvector</code>	<code>to_tsvector('english', '{"a": "The Fat Rats"}'::json)</code>	<code>'fat':2 'rat':3</code>

Продолжение таблицы 1.40

Функция	Тип результата	Описание	Пример	Результат
<code>json(b)_to_tsvector([конфигурациярег config,] документ json(b), фильтр json(b))</code>	<code>tsvector</code>	сокращает каждое значение в документе, заданное фильтром, до вектора <code>tsvector</code> , а затем соединяет их в порядке следования в документе и получает один <code>tsvector</code> . Фильтр представляет собой массив <code>jsonb</code> , в котором перечисляются типы элементов, которые должны войти в результирующий <code>tsvector</code> . Возможные значения для элементов фильтра: "string" (включить все строковые значения), "numeric" (включить все числовые значения в виде строк), "boolean" (включить все логические значения в виде "true"/"false"), "key" (включить все ключи) или "all" (включить всё вышеперечисленное). Эти значения можно комбинировать для включения, например, всех строковых и числовых значений.	<code>json_to_tsvector('english', '{"a": "The Fat Rats", "b": 123} '::json, ["string", "numeric"]')</code>	<code>'123':5 'fat':2 'rat':3</code>
<code>ts_delete(вектор tsvector, лексема text)</code>	<code>tsvector</code>	удаляет заданную лексему из вектора	<code>ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, 'fat')</code>	<code>'cat':3 'rat':5A</code>
<code>ts_delete(вектор tsvector, лексемы text[])</code>	<code>tsvector</code>	удаляет все вхождения лексем, перечисленных в массиве лексемы, из вектора	<code>ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, ARRAY['fat', 'rat'])</code>	<code>'cat':3</code>
<code>ts_filter(вектор tsvector, веса "char"[])</code>	<code>tsvector</code>	выбирает из вектора только элементы с заданным весом	<code>ts_filter('fat:2,4 cat:3b rat:5A'::tsvector, '{a,b}')</code>	<code>'cat':3B 'rat':5A</code>

Продолжение таблицы 1.40

Функция	Тип результата	Описание	Пример	Результат
<code>ts_headline([конфигурацияregconfig,] документ text, запрос tsquery [, параметры text])</code>	text	выводит фрагмент, соответствующий запросу	<code>ts_headline('x y z', 'z'::tsquery)</code>	<code>x y z</code>
<code>ts_headline([конфигурацияregconfig,] документ json(b), запрос tsquery [, параметры text])</code>	text	выводит фрагмент, соответствующий запросу	<code>ts_headline('{ "a": "x y z" }'::json, 'z'::tsquery)</code>	<code>{ "a": "x y z" }</code>
<code>ts_rank([весafloat4[,] вектор tsvector, запрос tsquery [, нормализация integer])</code>	float4	вычисляет ранг документа по отношению к запросу	<code>ts_rank(textsearch, query)</code>	0.818
<code>ts_rank_cd([весafloat4[,] вектор tsvector, запрос tsquery [, нормализация integer])</code>	float4	вычисляет ранг документа по отношению к запросу, используя плотность покрытия (CDR)	<code>ts_rank_cd('{0.1, 0.2, 0.4, 1.0}', textsearch, query)</code>	2.01317
<code>ts_rewrite(запрос tsquery, цель tsquery, замена tsquery)</code>	tsquery	подставляет в запросе вместо цели замену	<code>ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo bar'::tsquery)</code>	<code>'b' & ('foo' 'bar')</code>
<code>ts_rewrite(запрос tsquery, выборка text)</code>	tsquery	заменяет элементы запроса, выбирая цели и подстановки командой SELECT	<code>SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases')</code>	<code>'b' & ('foo' 'bar')</code>
<code>tsquery_phrase(запрос1 tsquery, запрос2 tsquery)</code>	tsquery	создаёт запрос, который ищет запрос1, за которым идёт запрос2 (как делает оператор <->)	<code>tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'))</code>	<code>'fat' <-> 'cat'</code>
<code>tsquery_phrase(запрос1 tsquery, запрос2 tsquery, расстояние integer)</code>	tsquery	создаёт запрос, который ищет запрос1, за которым идёт запрос2 на заданном расстоянии	<code>tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10)</code>	<code>'fat' <10> 'cat'</code>
<code>tsvector_to_array(tsvector)</code>	text[]	преобразует tsvector в массив лексем	<code>tsvector_to_array('fat:2,4 cat:3 rat:5A'::tsvector)</code>	<code>{cat,fat,rat}</code>
<code>tsvector_update_trigger()</code>	trigger	триггерная функция для автоматического изменения столбца типа tsvector	<code>CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swedish', title, body)</code>	

Окончание таблицы 1.40

Функция	Тип результата	Описание	Пример	Результат
<code>tsvector_update_trigger_column()</code>	trigger	триггерная функция для автоматического изменения столбца типа <code>tsvector</code>	<code>CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, configcol, title, body)</code>	
<code>unnest(tsvector, OUT лексема text, OUT позиции smallint[], OUT веса text)</code>	setof record	разворачивает <code>tsvector</code> в набор строк	<code>unnest('fat:2,4 cat:3 rat:5A'::tsvector)</code>	<code>(cat,{3},{D}) ...</code>
Примечание. Все функции текстового поиска, принимающие необязательный аргумент <code>regconfig</code> , будут использовать конфигурацию, указанную в параметре <code>default_text_search_config</code> , когда этот аргумент опущен.				

Функции в таблице 1.41 перечислены отдельно, так как они не очень полезны в традиционных операциях поиска. Они предназначены в основном для разработки и отладки новых конфигураций текстового поиска.

Таблица 1.41 – Функции отладки текстового поиска

Функция	Тип результата	Описание	Пример	Результат
<code>ts_debug([конфигурация regconfig,] документ text, OUT псевдоним text, OUT описание text, OUT фрагмент text, OUT словари regdictionary[], OUT словарь regdictionary, OUT лексемы text[])</code>	setof record	проверяет конфигурацию	<code>ts_debug('english', 'The Brightest supernovaes')</code>	<code>(asciiword, »Word, all ASCII», The, {english_stem, english_stem, {})) ...</code>
<code>ts_lexize(словарь regdictionary, фрагмент text)</code>	text[]	проверяет словарь	<code>ts_lexize('english_stem', 'stars')</code>	<code>{star}</code>
<code>ts_parse(имя_анализатора text, документ text, OUT код фрагмента integer, OUT фрагмент text)</code>	setof record	проверяет анализатор	<code>ts_parse('default', 'foo - bar')</code>	<code>(1,foo) ...</code>
<code>ts_parse(oid_анализатора oid, документ text, OUT код фрагмента integer, OUT фрагмент text)</code>	setof record	проверяет анализатор	<code>ts_parse(3722, 'foo - bar')</code>	<code>(1,foo) ...</code>
<code>ts_token_type(имя_анализатора text, OUT код фрагмента integer, OUT псевдоним text, OUT описание text)</code>	setof record	получает типы фрагментов, определенные анализатором	<code>ts_token_type('default')</code>	<code>(1,asciiword, »Word, all ASCII») ...</code>

Окончание таблицы 1.41

Функция	Тип результата	Описание	Пример	Результат
<code>ts_token_type(oid_анализатора oid, OUT код_фрагмента integer, OUT псевдоним text, OUT описание text)</code>	setof record	получает типы фрагментов, определенные анализатором	<code>ts_token_type(3722)</code>	(1,asciword, »Word, all ASCII») ...
<code>ts_stat(sql_запрос text, [веса text,] OUT слово text, OUT число_док integer, OUT число_вхожд integer)</code>	setof record	получает статистику колонки tsvector	<code>ts_stat('SELECT vector from apod')</code>	(foo,10,15) ...

1.15.XML-функции

Функции и подобные им выражения, описанные в этом разделе, работают со значениями типа `xml`. Информацию о типе `xml` вы можете найти в разделе 6.13 (Часть 1. Руководство оператора). Выражения `xmlparse` и `xmlserialize`, преобразующие значения `xml` в текст и обратно, здесь повторно не рассматриваются. Для использования большинства этих функций дистрибутив должен быть собран с ключом `configure --with-libxml`.

1.15.1. Создание XML-контента

Для получения XML-контента из данных SQL существует целый набор функций и функциональных выражений, особенно полезных для выдачи клиентским приложениям результатов запроса в виде XML-документов.

1.15.1.1. `xmlcomment`

`xmlcomment(текст)`

Функция `xmlcomment` создает XML-значение, содержащее XML-комментарий с заданным текстом. Этот текст не должен содержать «--» или заканчиваться знаком «-», чтобы результирующая конструкция была допустимой в XML. Если аргумент этой функции NULL, результатом её тоже будет NULL.

Пример.

```
SELECT xmlcomment('hello');
```

```
xmlcomment
```

```
-----
```

```
<!--hello-->
```

1.15.1.2. xmlconcat

```
xmlconcat(xml[, ...])
```

Функция `xmlconcat` объединяет несколько XML-значений и выдает в результате один фрагмент XML-контента. Значения NULL отбрасываются, так что результат будет равен NULL, только если все аргументы равны NULL.

Пример.

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');
```

```
      xmlconcat
-----
<abc/><bar>foo</bar>
```

XML-объявления, если они присутствуют, обрабатываются следующим образом. Если во всех аргументах содержатся объявления одной версии XML, эта версия будет выдана в результате; в противном случае версии не будет. Если во всех аргументах определен атрибут `standalone` со значением «yes», это же значение будет выдано в результате. Если во всех аргументах есть объявление `standalone`, но минимум в одном со значением «no», в результате будет это значение. В противном случае в результате не будет объявления `standalone`. Если же окажется, что в результате должно присутствовать объявление `standalone`, а версия не определена, тогда в результате будет выведена версия 1.0, так как XML-объявление не будет допустимым без указания версии. Указания кодировки игнорируются и будут удалены в любых случаях.

Пример.

```
SELECT xmlconcat('<?xml version=«1.1»?><foo/>',
  '<?xml version=«1.1» standalone=«no»?><bar/>');
```

```
      xmlconcat
-----
<?xml version=«1.1»?><foo/><bar/>
```

1.15.1.3. xmlelement

```
xmlelement(name имя [, xmlattributes(значение [AS
```



```
атрибут] [, ...]))  
    [, содержимое, ...])
```

Выражение `xmlelement` создает XML-элемент с заданным именем, атрибутами и содержимым.

Примеры:

```
SELECT xmlelement(name foo);
```

```
xmlelement  
-----  
<foo/>
```

```
SELECT xmlelement(name foo, xmlattributes('xyz' as  
bar));
```

```
xmlelement  
-----  
<foo bar=«xyz»/>
```

```
SELECT xmlelement(name foo, xmlattributes(current_date  
as bar),  
    'cont', 'ent');
```

```
xmlelement  
-----  
<foo bar=«2007-01-26»>content</foo>
```

Если имена элементов и атрибутов содержат символы, недопустимые в XML, эти символы заменяются последовательностями `_xNNNN_`, где `NNNN` – шестнадцатеричный код символа в Unicode, например:

```
SELECT xmlelement(name «foo$bar», xmlattributes('xyz' as  
«a&b»));
```

```
xmlelement  
-----  
<foo_x0024_bar a_x0026_b=«xyz»/>
```

Если в качестве значения атрибута используется колонка таблицы, имя атрибута можно не указывать явно, этим именем станет имя колонки. Во всех остальных случаях имя атрибута должно быть определено явно. Таким образом, это выражение допустимо:

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM
test;
```

А следующие варианты – нет:

```
SELECT xmlelement(name test, xmlattributes('constant'),
a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b)))
FROM test;
```

Содержимое элемента, если оно задано, будет форматировано согласно его типу данных. Когда оно имеет тип xml, из него можно конструировать сложные XML-документы. Например:

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
xmlelement(name abc),
xmlcomment('test'),
xmlelement(name xyz));
```

xmlelement

```
-----
<foo bar=«xyz»><abc/><!--test--><xyz/></foo>
```

Содержимое других типов будет оформлено в виде блока символьных данных XML. Это, в частности, означает, что символы <, > и & будут преобразованы в сущности XML. Двоичные данные (данные типа bytea) представляются в кодировке base64 или в шестнадцатеричном виде, в зависимости от значения параметра xmlbinary. Следует ожидать, что конкретные представления отдельных типов данных могут быть изменены для приведения типов SQL и Синергия-БД в соответствие со стандартом XML Schema, когда появится его более полное описание.

1.15.1.4. xmlforest

```
xmlforest(содержимое [AS имя] [, ...])
```

Выражение `xmlforest` создает последовательность XML-элементов с заданными именами и содержимым.

Примеры:

```
SELECT xmlforest('abc' AS foo, 123 AS bar);
```

```
          xmlforest
-----
<foo>abc</foo><bar>123</bar>
```

```
SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
WHERE table_schema = 'pg_catalog';
```

```
          xmlforest
-----
<table_name>pg_authid</table_name><column_name>rolname</column_name>

<table_name>pg_authid</table_name><column_name>rolsuper</column_name>

...
```

Как показано во втором примере, имя элемента можно опустить, если источником содержимого служит колонка (в этом случае именем элемента по умолчанию будет имя колонки). В противном случае имя элемента необходимо указывать.

Имена элементов с символами, недопустимыми для XML, преобразуются так же, как и для `xmlelement`. Данные содержимого тоже приводятся к виду, допустимому для XML (кроме данных, которые уже имеют тип `xml`).

Заметьте, что такие XML-последовательности не являются допустимыми XML-документами, если они содержат больше одного элемента на верхнем уровне, поэтому может иметь смысл вложить выражения `xmlforest` в `xmlelement`.

1.15.1.5. xmlpi

xmlpi(name цель [, содержимое])

Выражение xmlpi создает инструкцию обработки XML. Содержимое, если оно задано, не должно содержать последовательность символов ?>.

Пример.

```
SELECT xmlpi(name php, 'echo «hello world»;');
```

xmlpi

<?php echo «hello world»;?>

1.15.1.6. xmlroot

xmlroot(xml, version текст | нет значения [, standalone yes|no|нет значения])

Выражение xmlroot изменяет свойства корневого узла XML-значения. Если в нем указывается версия, она заменяет значение в объявлении версии корневого узла; также в корневой узел переносится значение свойства standalone.

```
SELECT xmlroot(xmlparse(document '<?xml version=«1.1»?>
<content>abc</content>'),
            version '1.0', standalone yes);
```

xmlroot

<?xml version=«1.0» standalone=«yes»?>
<content>abc</content>

1.15.1.7. xmlagg

xmlagg(xml)

Функция xmlagg, в отличие от других описанных здесь функций, является агрегатной. Она соединяет значения, поступающие на вход агрегатной функции, подобно функции xmlconcat, но делает это, обрабатывая множество строк, а не несколько выражений в одной строке. Дополнительно агрегатные функции описаны в разделе 1.21.

Пример.

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
      xmlagg
-----
<foo>abc</foo><bar/>
```

Чтобы задать порядок объединения элементов, в агрегатный вызов можно добавить предложение ORDER BY, описанное в подразделе 2.2.7. (Часть 1. Руководство оператора), например:

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;
      xmlagg
-----
<bar/><foo>abc</foo>
```

Следующий нестандартный подход рекомендовался в предыдущих версиях и может быть по-прежнему полезен в некоторых случаях:

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y
DESC) AS tab;
      xmlagg
-----
<bar/><foo>abc</foo>
```

1.15.2. Условия с XML

Описанные в этом разделе выражения проверяют свойства значений xml.

1.15.2.1. IS DOCUMENT

xml IS DOCUMENT

Выражение IS DOCUMENT возвращает true, если аргумент представляет собой правильный XML-документ, false в противном случае (т.е. если это фрагмент содержимого) и NULL, если его аргумент также NULL. Чем документы отличаются от фрагментов содержимого, вы можете узнать в разделе 6.13 (Часть 1. Руководство оператора).

1.15.2.2. IS NOT DOCUMENT

`xml IS NOT DOCUMENT`

Выражение `IS NOT DOCUMENT` возвращает `false`, если аргумент представляет собой правильный XML-документ, `true` в противном случае (т. е. если это фрагмент содержимого) и `NULL`, если его аргумент – `NULL`.

1.15.2.3. XMLEXISTS

`XMLEXISTS (текст`

`PASSING [BY REF] xml [BY REF])`

Функция `xmlexists` возвращает `true`, если выражение XPath в первом аргументе возвращает какие либо узлы, и `false` – в противном случае. Если один из аргументов равен `NULL`, результатом также будет `NULL`.

Пример.

```
SELECT xmlexists('//town[text() = ''Toronto'']' PASSING  
BY REF
```

```
'<towns><town>Toronto</town><town>Ottawa</town></towns>'  
);
```

```
xmlexists  
-----  
t  
(1 row)
```

Указания `BY REF` не несут смысловой нагрузки в Синергия-БД, но могут присутствовать для соответствия стандарту SQL и совместимости с другими реализациями. По стандарту SQL первое указание `BY REF` является обязательным, а второе – нет. Также заметьте, что, согласно стандарту SQL, конструкция `xmlexists` должна принимать в первом аргументе выражение XQuery, но Синергия-БД в настоящее время поддерживает только XPath, подмножество XQuery.

1.15.2.4. xml_is_well_formed

`xml_is_well_formed(текст)`

`xml_is_well_formed_document(текст)`

`xml_is_well_formed_content(текст)`

Эти функции проверяют, является ли текст правильно оформленным XML, и

возвращают соответствующее логическое значение. Функция `xml_is_well_formed_document` проверяет аргумент как правильно оформленный документ, а `xml_is_well_formed_content` – правильно оформленное содержание. Функция `xml_is_well_formed` может делать первое или второе, в зависимости от значения параметра конфигурации `xmloption` (`DOCUMENT` или `CONTENT`, соответственно). Это значит, что `xml_is_well_formed` помогает понять, будет ли успешным простое приведение к типу `xml`, тогда как две другие функции проверяют, будут ли успешны соответствующие варианты `XMLPARSE`.

Примеры:

```
SET xmloption TO DOCUMENT;
SELECT xml_is_well_formed('<>');
  xml_is_well_formed
-----
f
(1 row)

SELECT xml_is_well_formed('<abc/>');
  xml_is_well_formed
-----
t
(1 row)

SET xmloption TO CONTENT;
SELECT xml_is_well_formed('abc');
  xml_is_well_formed
-----
t
(1 row)

SELECT xml_is_well_formed_document(
  '<pg:foo
xmlns:pg=«http://postgresql.org/stuff»>bar</pg:foo>');

```

```
xml_is_well_formed_document
-----
t
(1 row)
```

```
SELECT xml_is_well_formed_document(
    '<pg:foo
xmlns:pg=«http://postgresql.org/stuff»>bar</my:foo>');
xml_is_well_formed_document
-----
f
(1 row)
```

Последний пример показывает, что при проверке также учитывается, совпадают ли пространства имен.

1.15.3. Обработка XML

Для обработки значений типа `xml` в Синергия-БД представлены функции `xpath` и `xpath_exists`, вычисляющие выражения XPath 1.0, а также табличная функция `XMLTABLE`.

1.15.3.1. xpath

```
xpath(xpath, xml [, nsarray])
```

Функция `xpath` вычисляет выражение XPath (аргумент `xpath` типа `text`) для заданного `xml`. Она возвращает массив XML-значений с набором узлов, полученных в результате выражения XPath. Если выражение XPath выдает не набор узлов, а скалярное значение, возвращается массив из одного элемента.

Вторым аргументом должен быть правильно оформленный XML-документ. В частности, в нем должен быть единственный корневой элемент.

В необязательном третьем аргументе функции передаются сопоставления пространств имен. Эти сопоставления должны определяться в двумерном массиве типа `text`, во второй размерности которого 2 элемента (т.е. это должен быть массив массивов, состоящих из 2 элементов). В первом элементе каждого массива определяется псевдоним (префикс) пространства имен, а во втором – его URI.

Псевдонимы, определенные в этом массиве, не обязательно должны совпадать с префиксами пространств имен в самом XML-документе (другими словами, для XML-документа и функции `xpath` псевдонимы имеют локальный характер).

Пример.

```
SELECT xpath('/my:a/text()',
            '<my:a
xmlns:my=«http://example.com»>test</my:a>',
            ARRAY[ARRAY['my', 'http://example.com']] );
```

```
xpath
-----
{test}
(1 row)
```

Для пространства имен по умолчанию (анонимного) это выражение можно записать так:

```
SELECT xpath('//mydefns:b/text()',
            '<a
xmlns=«http://example.com»><b>test</b></a>',
            ARRAY[ARRAY['mydefns',
            'http://example.com']] );
```

```
xpath
-----
{test}
(1 row)
```

1.15.3.2. `xpath_exists`

```
xpath_exists(xpath, xml [, nsarray])
```

Функция `xpath_exists` представляет собой специализированную форму функции `xpath`. Она возвращает не весь набор XML-узлов, удовлетворяющих выражению XPath, а только одно логическое значение, показывающее, есть ли такие узлы. Эта функция равнозначна стандартному условию `XMLEXISTS`, за исключением того, что она также поддерживает сопоставления пространств имен.

```
SELECT xpath_exists('/my:a/text()',
```

```
'<my:a
xmlns:my=«http://example.com»>test</my:a>',
ARRAY[ARRAY['my',
'http://example.com']]);
```

```
xpath_exists
-----
```

```
t
(1 row)
```

1.15.3.3. xmltable

```
xmltable( [XMLNAMESPACES (URI пространства имён AS имя
пространства имён[, ...]),]
        выражение_строки PASSING [BY REF]
выражение_документа [BY REF]
        COLUMNS имя { тип [PATH выражение_столбца]
[DEFAULT выражение_по_умолчанию] [NOT NULL | NULL]
        | FOR ORDINALITY }
        [, ...]
)
```

Функция `xmltable` строит таблицу из данного XML-значения по фильтру XPath, извлекающему строки, с необязательным набором определений столбцов.

Необязательное предложение `XMLNAMESPACES` задаёт разделённый запятыми список пространств имён. В нём определяются пространства имён XML и их псевдонимы. Определение пространства по умолчанию в настоящее время не поддерживается.

Обязательный аргумент `выражение_строки` задаёт выражение XPath, вычисляемое в данном XML-документе и выдающее упорядоченную последовательность XML-узлов. Эту последовательность `xmltable` преобразует в выходные строки.

В аргументе `выражение_документа` передаётся XML-документ для обработки. Предложения `BY REF` не действуют в Синергия-БД, но допускаются для соответствия стандарту SQL и совместимости с другими реализациями. Передаваемый XML-документ должен быть правильно оформленным;

фрагменты/леса не принимаются.

В предложении COLUMNS задаётся список столбцов в выходной таблице. Если предложение COLUMNS опускается, строки в результирующем наборе содержат один столбец типа `xml` с данными, которые отобрало выражение `строки`. Если COLUMNS присутствует, каждый его элемент описывает один столбец. Формат этого предложения показан выше в примере синтаксиса. Имя столбца и его тип должны задаваться обязательно: путь, значение по умолчанию и признак допустимости NULL могут опускаться.

Столбец с признаком `FOR ORDINALITY` будет заполняться номерами строк, соответствующими порядку, в котором строки находились в исходном документе XML. Признак `FOR ORDINALITY` может быть не больше чем у одного столбца.

В указании выражение `столбца` для столбца задаётся выражение `XPath`, вычисляемое для каждой строки относительно результата выражение `строки` и выдающее значение столбца. Если выражение `столбца` отсутствует, в качестве неявного пути используется имя столбца.

Если выражение `XPath` для столбца выдаёт несколько элементов, происходит ошибка. Если выражению соответствует пустой элемент, результатом будет пустая строка (не NULL). Атрибуты `xsi:nil` при поиске игнорируются.

Значением столбца становится текстовое содержимое XML-узла, отобранного выражением `столбца`. Несколько узлов `text()` внутри целевого элемента складываются вместе по порядку. Все дочерние элементы, инструкции обработки и комментарии игнорируются, но текстовое содержимое дочерних элементов добавляется к результату. Заметьте, что исключительно пробельные узлы `text()` между двумя не текстовыми элементами сохраняются, как не убираются и ведущие пробелы в узле `text()`.

Если выражение пути не находит элемент строки, но задано выражение `по_умолчанию`, значением поля становится результат вычисления этого выражения. Если предложение `DEFAULT` для этого столбца отсутствует, в это поле записывается NULL. В выражении `по_умолчанию` можно обращаться к значениям выходных столбцов, описанных ранее в списке столбцов, так что

значение по умолчанию одного столбца можно получить из значения другого.

Столбцы могут иметь признак NOT NULL. Если выражение `выражение_столбца` для столбца с признаком NOT NULL не соответствует ничему и при этом отсутствует указание DEFAULT или выражение `по_умолчанию` также выдаёт NULL, происходит ошибка.

В отличие от обычных функций Синергия-БД, выражение `выражение_столбца` и выражение `по_умолчанию` не сводятся к единственному простому значению перед вызовом функции. Так, выражение `выражение_столбца` обычно вычисляется ровно один раз для каждой входной строки, а выражение `по_умолчанию` вычисляется каждый раз, когда для поля требуется значение по умолчанию. Если выражение оказывается стабильным или постоянным, повторное вычисление может не выполняться. По сути `xmltable` действует больше как подзапрос, а не как вызов функции. Это значит, что вы можете с пользой применять изменчивые функции, такие как `nextval` в выражении `по_умолчанию`, а выражение `выражение_столбца` может зависеть от других частей XML-документа.

Примеры:

```
CREATE TABLE xmldata AS SELECT
xml $$
<ROWS>
  <ROW id="1">
    <COUNTRY_ID>AU</COUNTRY_ID>
    <COUNTRY_NAME>Australia</COUNTRY_NAME>
  </ROW>
  <ROW id="5">
    <COUNTRY_ID>JP</COUNTRY_ID>
    <COUNTRY_NAME>Japan</COUNTRY_NAME>
    <PREMIER_NAME>Shinzo Abe</PREMIER_NAME>
    <SIZE unit="sq_mi">145935</SIZE>
  </ROW>
  <ROW id="6">
    <COUNTRY_ID>SG</COUNTRY_ID>
    <COUNTRY_NAME>Singapore</COUNTRY_NAME>
```

```

    <SIZE unit="sq_km">697</SIZE>
  </ROW>
</ROWS>
$$ AS data;

SELECT xmltable.*
  FROM xmldata,
       XMLTABLE ('//ROWS/ROW'
                 PASSING data
                 COLUMNS id int PATH '@id',
                          ordinality FOR ORDINALITY,
                          "COUNTRY_NAME" text,
                          country_id text PATH
'        COUNTRY_ID',
                          size_sq_km float PATH
'        SIZE[@unit = "sq_km"]',
                          size_other text PATH
'        concat(SIZE[@unit!="sq_km"], " ",
SIZE[@unit!="sq_km"]/@unit)',
                          premier_name text PATH
'        PREMIER_NAME' DEFAULT 'not specified') ;

```

Следующий пример иллюстрирует сложение нескольких узлов `text()`, использование имени столбца в качестве фильтра `XPath` и обработку пробельных символов, XML-комментариев и инструкций обработки:

```

CREATE TABLE xmlelements AS SELECT
xml $$
  <root>
    <element> Hello<!-- xyxxz -->2a2<?aaaaa?> <!--x-->
bbb<x>xxx</x>CC  </element>
  </root>
$$ AS data;

SELECT xmltable.*

```

```
FROM xmlelements, XMLTABLE('/root' PASSING data
COLUMNS element text);
```

```
element
```

```
-----
```

```
Hello2a2    bbbCC
```

Следующий пример показывает, как с помощью предложения XMLNAMESPACES можно задать список пространств имён, используемых в XML-документе и в выражениях XPath:

```
WITH xmldata(data) AS (VALUES ('
<example xmlns="http://example.com/myns"
xmlns:B="http://example.com/b">
  <item foo="1" B:bar="2"/>
  <item foo="3" B:bar="4"/>
  <item foo="4" B:bar="5"/>
</example>':::xml)
)

SELECT xmltable.*
FROM XMLTABLE(XMLNAMESPACES('http://example.com/myns'
AS x,
                                'http://example.com/b' AS
                                "B"),
              '/x:example/x:item'
              PASSING (SELECT data FROM xmldata)
              COLUMNS foo int PATH '@foo',
                        bar int PATH '@B:bar');
```

```
foo | bar
-----+-----
1   |    2
3   |    4
4   |    5
(3 rows)
```

1.15.4. Отображение таблиц в XML

Следующие функции отображают содержимое реляционных таблиц в значения

XML. Их можно рассматривать как средства экспорта в XML:

```
table_to_xml(tbl regclass, nulls boolean, tableforest  
boolean, targetns text)
```

```
query_to_xml(query text, nulls boolean, tableforest  
boolean, targetns text)
```

```
cursor_to_xml(cursor refcursor, count int, nulls  
boolean,
```

```
tableforest boolean, targetns text)
```

Результат всех этих функций имеет тип xml.

`table_to_xml` отображает в xml содержимое таблицы, имя которой задается в параметре `tbl`. Тип `regclass` принимает идентификаторы строк в обычной записи, которые могут содержать указание схемы и кавычки. Функция `query_to_xml` выполняет запрос, текст которого передается в параметре `query`, и отображает в xml результирующий набор. Последняя функция, `cursor_to_xml` выбирает указанное число строк из курсора, переданного в параметре `cursor`. Этот вариант рекомендуется использовать с большими таблицами, так как все эти функции создают результирующий xml в памяти.

Если параметр `tableforest` имеет значение `false`, результирующий XML-документ выглядит так:

```
<имя_таблицы>  
  <row>  
    <имя_колонки1>данные</имя_колонки1>  
    <имя_колонки2>данные</имя_колонки2>  
  </row>  
  
  <row>  
    ...  
  </row>  
  
  ...  
</имя_таблицы>
```

А если `tableforest` равен `true`, в результате будет выведен следующий фрагмент XML:

```
<имя_таблицы>
  <имя_колонки1>data</имя_колонки1>
  <имя_колонки2>data</имя_колонки2>
</имя_таблицы>
```

```
<имя_таблицы>
  ...
</имя_таблицы>
```

...

Если имя таблицы неизвестно, например, при отображении результатов запроса или курсора, вместо него в первом случае вставляется `table`, а во втором — `row`.

Выбор между этими форматами остается за пользователем. Первый вариант позволяет создать готовый XML-документ, что может быть полезно для многих приложений, а второй удобно применять с функцией `cursor_to_xml`, если её результаты будут собираться в документ позже. Полученный результат можно изменить по вкусу с помощью рассмотренных выше функций создания XML-содержимого, в частности `xmlelement`.

Значения данных эти функции отображают так же, как и ранее описанная функция `xmlelement`.

Параметр `nulls` определяет, нужно ли включать в результат значения `NULL`. Если он установлен, значения `NULL` в колонках представляются так:

```
<имя_колонки xsi:nil=«true»/>
```

Здесь `xsi` — это префикс пространства имен XML Schema Instance. При этом в результирующий XML будет добавлено соответствующее объявление пространства имен. Если же данный параметр имеет значение `false`, колонки со значениями `NULL` просто не будут выводиться.

Параметр `targetns` определяет целевое пространство имен для

результатирующего XML. Если пространство имен не нужно, значением этого параметра должна быть пустая строка.

Следующие функции выдают документы XML Schema, которые содержат схемы отображений, выполняемых соответствующими ранее рассмотренными функциями:

```
table_to_xmlschema(tbl regclass, nulls boolean,  
tableforest boolean,  
targetns text)  
query_to_xmlschema(query text, nulls boolean,  
tableforest boolean,  
targetns text)  
cursor_to_xmlschema(cursor refcursor, nulls boolean,  
tableforest boolean,  
targetns text)
```

Чтобы результаты отображения данных в XML соответствовали XML-схемам, важно, чтобы паре функций передавались одинаковые параметры.

Следующие функции выдают отображение данных в XML и соответствующую XML-схему в одном документе (или фрагменте), объединяя их вместе. Это может быть полезно там, где желательно получить самодостаточные результаты с описанием:

```
table_to_xml_and_xmlschema(tbl regclass, nulls boolean,  
tableforest boolean,  
targetns text)  
query_to_xml_and_xmlschema(query text, nulls boolean,  
tableforest boolean,  
targetns text)
```

В дополнение к ним есть следующие функции, способные выдать аналогичные представления для целых схем в базе данных или даже всей текущей базы данных:

```
schema_to_xml(schema name, nulls boolean, tableforest  
boolean,  
targetns text)  
schema_to_xmlschema(schema name, nulls boolean,  
tableforest boolean,  
targetns text)
```

```
schema_to_xml_and_xmlschema(schema name, nulls boolean,
tableforest boolean,
    targetns text)
```

```
database_to_xml(nulls boolean, tableforest boolean,
targetns text)
```

```
database_to_xmlschema(nulls boolean, tableforest
boolean, targetns text)
```

```
database_to_xml_and_xmlschema(nulls boolean, tableforest
boolean,
    targetns text)
```

Заметьте, что объем таких данных может быть очень большим, а XML будет создаваться в памяти. Поэтому, вместо того, чтобы пытаться отобразить в XML сразу все содержимое больших схем или баз данных, лучше делать это по таблицам, возможно даже используя курсор.

Результат отображения содержимого схемы будет выглядеть так:

```
<имя_схемы>
```

```
отображение_таблицы1
```

```
отображение_таблицы2
```

```
...
```

```
</имя_схемы>
```

Формат отображения таблицы определяется параметром `tableforest`, описанным выше.

Результат отображения содержимого базы данных будет таким:

```
<имя_БД>
```

```
<имя_схемы1>
```

```
...
```

```
</имя_схемы1>
```

```
<имя_схемы2>
...
</имя_схемы2>
```

...

```
</имя_БД>
```

Здесь отображение схемы имеет вид, показанный выше.

1.16. Функции и операторы JSON

В таблице 1.42 перечислены функции, позволяющие создавать данные в формате JSON.

Таблица 1.42 – Операторы для типов json и jsonb

Оператор	Тип правого операнда	Описание	Пример	Результат примера
->	int	Выдает элемент массива JSON (по индексу с 0)	'[{«a»:»foo»},{«b»:»bar»},{«c»:»baz»}]'::json->2	{«c»:»baz»}
->	text	Выдает поле объекта JSON по ключу	'{«a»: {«b»:»foo»}}'::json->'a'	{«b»:»foo»}
->>	int	Выдает элемент массива JSON в типе text	'[1,2,3]'::json->>2	3
->>	text	Выдает поле объекта JSON в типе text	'{«a»:1,»b»:2}'::json->>'b'	2
#>	text[]	Выдает объект JSON по заданному пути	'{«a»: {«b»: {«c»: «foo»}}} '::json#>'a,b'	{«c»: «foo»}
#>>	text[]	Выдает объект JSON по заданному пути в типе text	'{«a»: [1,2,3], »b»: [4,5,6]} '::json#>>'a,2'	3

Приечание. Эти операторы существуют в двух вариациях для типов json и jsonb. Операторы извлечения поля/элемента/пути возвращают тот же тип, что у операнда слева (json или jsonb), за исключением тех, что возвращают тип text (они возвращают значение как текстовое). Если входные данные JSON не содержат структуры, удовлетворяющей запросу, например в них нет искомого элемента, то операторы извлечения поля/элемента/пути не выдают ошибку, а возвращают NULL. Все операторы извлечения поля/элемента/пути, принимающие целочисленные позиции в массивах JSON, поддерживают и отсчёт от конца массива по отрицательной позиции.

Стандартные операторы сравнения есть для типа jsonb, но не для json. Они следуют правилам сортировки для операций В-дерева.

Некоторые из следующих операторов существуют только для jsonb, как

показано в таблице 1.43. Многие из этих операторов могут быть проиндексированы с помощью классов операторов `jsonb`. Полное описание проверок на вхождение и существование для `jsonb` приведено в подразделе 6.14.3 (Часть 1. Руководство оператора). Как эти операторы могут использоваться для эффективного индексирования данных `jsonb`, описано в подразделе 6.14.4 (Часть 1. Руководство оператора).

Таблица 1.43 – Дополнительные операторы `jsonb`

Оператор	Тип правого операнда	Описание	Пример
@>	<code>jsonb</code>	Левое значение JSON содержит на верхнем уровне путь/значение JSON справа?	<code>'{"a":1, "b":2}':::jsonb @> '{"b":2}':::jsonb</code>
<@	<code>jsonb</code>	Путь/значение JSON слева содержится на верхнем уровне в правом значении JSON?	<code>'{"b":2}':::jsonb <@ '{"a":1, "b":2}':::jsonb</code>
?	<code>text</code>	Присутствует ли <i>строка</i> в качестве ключа верхнего уровня в значении JSON?	<code>'{"a":1, "b":2}':::jsonb ? 'b'</code>
?	<code>text[]</code>	Какие-либо <i>строки</i> массива присутствуют в качестве ключей верхнего уровня?	<code>'{"a":1, "b":2, "c":3}':::jsonb ? array['b', 'c']</code>
?&	<code>text[]</code>	Все <i>строки</i> массива присутствуют в качестве ключей верхнего уровня?	<code>'["a", "b"]':::jsonb ?& array['a', 'b']</code>
	<code>jsonb</code>	Соединяет два значения <code>jsonb</code> в новое значение <code>jsonb</code>	<code>'["a", "b"]':::jsonb '{"c", "d"}':::jsonb</code>
-	<code>text</code>	Удаляет пару ключ/значение или элемент- <i>строку</i> из левого операнда. Пары ключ/значение выбираются по значению ключа.	<code>'{"a": "b"}':::jsonb - 'a'</code>
-	<code>text[]</code>	Удаляет множество пар ключ/значение или элементы- <i>строки</i> из левого операнда. Пары ключ/значение выбираются по значению ключа.	<code>'{"a": "b", "c": "d"}':::jsonb - '{a,c}':::text[]</code>
-	<code>integer</code>	Удаляет из массива элемент в заданной позиции (отрицательные номера позиций отсчитываются от конца). Выдаёт ошибку, если контейнер верхнего уровня – не массив.	<code>'["a", "b"]':::jsonb - 1</code>
#-	<code>text[]</code>	Удаляет поле или элемент с заданным путём (для массивов JSON отрицательные номера позиций отсчитываются от конца)	<code>'["a", {"b":1}]':::jsonb #- '{1,b}'</code>

Примечание. Оператор `||` соединяет элементы верхнего уровня обоих операндов. Вложенность при этом не учитывается. Например, если оба операнда содержат поле с одноимённым ключом, значением поля в результате будет значение, взятое из правого операнда.

В таблице 1.44 показаны функции, позволяющие создавать значения типов

json и jsonb. Для типа jsonb нет аналогов функций row_to_json и array_to_json, но практически тот же результат можно получить с помощью to_jsonb.

Таблица 1.44 – Функции для создания JSON

Функция	Описание	Пример	Результат примера
to_json(anyelement) to_jsonb(anyelement)	Возвращает значение в виде json или jsonb. Массивы и составные структуры преобразуются (рекурсивно) в массивы и объекты; для других типов, для которых определено приведение к json, применяется эта функция приведения, а для всех остальных выдаётся скалярное значение. Значения всех скалярных типов, кроме числового, логического и NULL, представляются в текстовом виде, в стиле, допустимом для значений json или jsonb.	to_json('Fred said "Hi."':text)	"Fred said \"Hi.\""
array_to_json(anyarray [, pretty_bool])	Возвращает массив в виде массива JSON. Многомерный массив Синергия-БД становится массивом массивов JSON. Если параметр pretty_bool равен true, между элементами 1-ой размерности вставляются разрывы строк.	array_to_json('{{1,5},{99,100}}':int[])	[[1,5],[99,100]]
row_to_json(record [, pretty_bool])	Возвращает кортеж в виде объекта JSON. Если параметр pretty_bool равен true, между элементами 1-ой размерности вставляются разрывы строк.	row_to_json(row(1,'foo'))	{"f1":1,"f2":"foo"}
json_build_array(VARIADIC "any") jsonb_build_array(VARIADIC "any")	Формирует массив JSON (возможно, разнородный) из переменного списка аргументов.	json_build_array(1,2,'3',4,5)	[1, 2, "3", 4, 5]
json_build_object(VARIADIC "any") jsonb_build_object(VARIADIC "any")	Формирует объект JSON из переменного списка аргументов. По соглашению в этом списке перечисляются по очереди ключи и значения.	json_build_object('foo',1,'bar',2)	{"foo": 1, "bar": 2}
json_object(text[]) jsonb_object(text[])	Формирует объект JSON из текстового массива. Этот массив должен иметь либо одну размерность с чётным числом элементов (в этом случае они воспринимаются как чередующиеся ключи/значения), либо две размерности и при этом каждый внутренний массив содержит ровно два элемента, которые воспринимаются как пара ключ/значение.	json_object('{a, 1, b, "def", c, 3.5}')	{"a": "1", "b": "def", "c": "3.5"}
json_object(keys text[], values text[]) jsonb_object(keys text[], values text[])	Эта форма json_object принимает ключи и значения по парам из двух отдельных массивов. Во всех остальных отношениях она не отличается от формы с одним аргументом.	json_object('{a, b}', '{1,2}')	{"a": "1", "b": "2"}

Примечание. Функции array_to_json и row_to_json подобны to_json, но предлагают возможность улучшенного вывода. Действие to_json, описанное выше, распространяется на каждое отдельное значение, преобразуемое этими функциями.

В таблице 1.45 показаны функции, предназначенные для работы со значениями `json` и `jsonb`.

Таблица 1.45 – Функции для обработки JSON

Функция	Тип результата	Описание	Пример	Результат примера
<code>json_array_length(json)</code> <code>jsonb_array_length(jsonb)</code>	<code>int</code>	Возвращает число элементов во внешнем массиве JSON.	<code>json_array_length(' [1,2,3, {"f1":1,"f2": [5,6]},4]')</code>	5
<code>json_each(json)</code> <code>jsonb_each(jsonb)</code>	<code>setof key text, value json</code> <code>setof key text, value jsonb</code>	Разворачивает внешний объект JSON в набор пар ключ/значение (key/value).	<code>select * from json_each('{"a":"foo", "b":"bar"}')</code>	key value -----+----- a "foo" b "bar"
<code>json_each_text(json)</code> <code>jsonb_each_text(jsonb)</code>	<code>setof key text, value text</code>	Разворачивает внешний объект JSON в набор пар ключ/значение (key/value). Возвращаемые значения будут иметь тип <code>text</code> .	<code>select * from json_each_text('{"a":"foo", "b":"bar"}')</code>	key value -----+----- a foo b bar
<code>json_extract_path(from_json json, VARIADIC path_elems text[])</code> <code>jsonb_extract_path(from_json jsonb, VARIADIC path_elems text[])</code>	<code>json</code> <code>jsonb</code>	Возвращает значение JSON по пути, заданному элементами пути (<code>path_elems</code>) (равнозначно оператору <code>#></code> operator).	<code>json_extract_path('{"f2":{"f3":1,"f4":{"f5":99,"f6":"foo"}}','f4')</code>	<code>{"f5":99,"f6":"foo"}</code>
<code>json_extract_path_text(from_json json, VARIADIC path_elems text[])</code> <code>jsonb_extract_path_text(from_json jsonb, VARIADIC path_elems text[])</code>	<code>text</code>	Возвращает значение JSON по пути, заданному элементами пути <code>path_elems</code> , как <code>text</code> (равнозначно оператору <code>#>></code>).	<code>json_extract_path_text('{"f2":{"f3":1,"f4":{"f5":99,"f6":"foo"}}','f4','f6')</code>	foo
<code>json_object_keys(json)</code> <code>jsonb_object_keys(jsonb)</code>	<code>setof text</code>	Возвращает набор ключей во внешнем объекте JSON.	<code>json_object_keys('{"f1":"abc","f2":{"f3":"a","f4":"b"}}')</code>	json_object_keys ----- f1 f2

Продолжение таблицы 1.45

Функция	Тип результата	Описание	Пример	Результат примера
json_populate_record(base anyelement, from_json json) jsonb_populate_record(base anyelement, from_json jsonb)	anyelement	Разворачивает объект из from_json в табличную строку, в которой столбцы соответствуют типу строки, заданному параметром base	select * from json_populate_record(null::myrowtype, '{"a": 1, "b": ["2", "a b"], "c": {"d": 4, "e": "a b c"}}')	a b c ----+-----+----- 1 {2, "a b"} (4, "a b c")
json_populate_recordset(base anyelement, from_json json) jsonb_populate_recordset(base anyelement, from_json jsonb)	setof anyelement	Разворачивает внешний массив объектов из from_json в набор табличных строк, в котором столбцы соответствуют типу строки, заданному параметром base	select * from json_populate_recordset(null::myrowtype, '[[{"a":1,"b":2},{ "a":3,"b":4}]')	a b ----+--- 1 2 3 4
json_array_elements(json) jsonb_array_elements(jsonb)	setof json setof jsonb	Разворачивает массив JSON в набор значений JSON.	select * from json_array_elements('[1,true,[2,false]]')	value ----- 1 true [2,false]
json_array_elements_text(json) jsonb_array_elements_text(jsonb)	setof text	Разворачивает массив JSON в набор значений text.	select * from json_array_elements_text('["foo","bar"]')	value ----- foo bar
json_typeof(json) jsonb_typeof(jsonb)	text	Возвращает тип внешнего значения JSON в виде текстовой строки. Возможные типы: object, array, string, number, boolean и null.	json_typeof('-123.4')	number
json_to_record(json) jsonb_to_record(jsonb)	record	Формирует обычную запись из объекта JSON. Как и со всеми функциями, возвращающими record, при вызове необходимо явно определить структуру записи с помощью предложения AS.	select * from json_to_record('{ "a":1,"b": [1,2,3], "c": [1,2,3], "e": "bar", "r": {"a": 123, "b": "a b c"}}') as x(a int, b text, c int[], d text, r myrowtype)	a b c d r ----+-----+----- +----+-----+----- 1 [1,2,3] {1,2,3} (123, "a b c")

Продолжение таблицы 1.45

Функция	Тип результата	Описание	Пример	Результат примера
<code>json_to_recordset(json)</code> <code>jsonb_to_recordset(jsonb)</code>	<code>setof record</code>	Формирует обычный набор записей из массива объекта JSON. Как и со всеми функциями, возвращающими <code>record</code> , при вызове необходимо явно определить структуру записи с помощью предложения AS.	<pre>select * from json_to_recordset(' [{"a":1, "b":"foo"}, {"a":2, "c":"bar"}]') as x(a int, b text);</pre>	<pre>a b ---+----- 1 foo 2 </pre>
<code>json_strip_nulls(from_json(json))</code> <code>jsonb_strip_nulls(from_json(jsonb))</code>	<code>json</code> <code>jsonb</code>	Возвращает значение <code>from_json</code> , из которого исключаются все поля объекта, содержащие значения NULL. Другие значения NULL остаются нетронутыми.	<code>json_strip_nulls(' [{"f1":1, "f2":null}, 2, null, 3]')</code>	<pre>[{"f1":1}, 2, null, 3]</pre>
<code>jsonb_set(target jsonb, path text[], new_value jsonb[, create_missing boolean])</code>	<code>jsonb</code>	Возвращает значение <code>target</code> , в котором раздел с заданным путём (<code>path</code>) заменяется новым значением (<code>new_value</code>), либо в него добавляется значение <code>new_value</code> , если аргумент <code>create_missing</code> равен <code>true</code> (это значение по умолчанию) и элемент, на который ссылается <code>path</code> , не существует. Как и с операторами, рассчитанными на пути, отрицательные числа в пути (<code>path</code>) обозначают отсчёт от конца массивов JSON.	<code>jsonb_set(' [{"f1":1, "f2":null}, 2, null, 3] ', '{0,f1}', '[2, 3, 4]', false)</code> <code>jsonb_set(' [{"f1":1, "f2":null}, 2] ', '{0,f3}', '[2, 3, 4]')</code>	<pre>[{"f1": [2, 3, 4], "f2": null}, 2, null, 3] [{"f1": 1, "f2": null, "f3": [2, 3, 4]}, 2]</pre>

Окончание таблицы 1.45

Функция	Тип результата	Описание	Пример	Результат примера
<code>jsonb_insert(target jsonb, path text[], new_value jsonb, [insert_after boolean])</code>	jsonb	Возвращает значение target с вставленным в него новым значением new_value. Если место в target, выбранное путём path, оказывается в массиве JSONB, new_value будет вставлен до (по умолчанию) или после (если параметр insert_after равен true) выбранной позиции. Если место в target, выбранное путём path, оказывается в объекте JSONB, значение new_value будет вставлено в него, только если заданный путь path не существует.	<code>jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"')</code> <code>jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"', true)</code>	<pre>{ "a": [0, "new_value", 1, 2] } { "a": [0, 1, "new_value", 2] }</pre>
<code>jsonb_pretty(from_json jsonb)</code>	text	Возвращает значение from_json в виде текста JSON с отступами.	<code>jsonb_pretty(' [{"f1":1,"f2":null},2,null,3]')</code>	<pre>[{ "f1": 1, "f2": null }, 2, null, 3]</pre>

Примечание. Многие из этих функций и операторов преобразуют спецпоследовательности Unicode в JSON-строках в соответствующие одиночные символы. Для входных данных типа jsonb это ничем не грозит, так как преобразование уже выполнено; однако для типа json в результате может произойти ошибка, как отмечено в разделе 6.14. (Часть 1. Руководство оператора).

Хотя в примерах использования перечисленных функций применяются константы, обычно эти функции обращаются к таблицам в предложении FROM, а в качестве аргумента указывается один из столбцов типа json или jsonb. Извлечённые значения затем могут использоваться в других частях запроса, например, в предложениях WHERE и результирующих списках. Извлекая множество значений подобным образом, можно значительно увеличить производительность по

сравнению с использованием операторов, работающих с отдельными ключами.

Примечания:

1. В `target` должны присутствовать все элементы пути, заданного параметром `path` функций `jsonb_set` и `jsonb_insert`, за исключением последнего. Если `create_missing` равен `false`, должны присутствовать абсолютно все элементы пути `path`, переданного функции `jsonb_set`. Если это условие не выполняется, значение `target` возвращается неизменённым.

2. Если последним элементом пути оказывается ключ объекта, он будет создан в случае отсутствия и получит новое значение. Если последний элемент пути – позиция в массиве, то когда она положительна, целевой элемент отсчитывается слева, а когда отрицательна – справа, то есть `-1` указывает на самый правый элемент и т. д. Если позиция лежит вне диапазона `-длина_массива .. длина_массива -1`, и параметр `create_missing` равен `true`, новое значение добавляется в начало массива, если позиция отрицательна, и в конец, если положительна.

3. Значение `null`, возвращаемое функцией `json_typeof`, не следует путать с SQL `NULL`. Тогда как при вызове `json_typeof('null'::json)` возвращается `null`, при вызове `json_typeof(NULL::json)` будет возвращено значение SQL `NULL`.

4. Если аргумент функции `json_strip_nulls` содержит повторяющиеся имена полей в любом объекте, в результате могут проявиться семантические различия, в зависимости от порядка этих полей. Это не проблема для функции `jsonb_strip_nulls`, так как в значениях `jsonb` имена полей не могут дублироваться.

В разделе 1.21 вы также можете узнать о агрегатной функции `json_agg`, которая агрегирует значения записи в виде JSON, и агрегатной функции `json_object_agg`, которая агрегирует пары значений в один объект JSON.

1.17. Функции для работы с последовательностями

В этом разделе описаны функции для работы с объектами, представляющими последовательности. Такие объекты (также называемыми генераторами последовательностей или просто последовательностями) являются специальными таблицами из одной строки и создаются командой `CREATE SEQUENCE`. Используются они обычно для получения уникальных идентификаторов строк таблицы. Функции, перечисленные в таблице 1.46, предоставляют простые и безопасные для параллельного использования методы получения очередных значений таких последовательностей.

Таблица 1.46 – Функции для работы с последовательностями

Функция	Тип результата	Описание
<code>currval(regclass)</code>	<code>bigint</code>	Выдает значение заданной последовательности, которое было возвращено при последнем вызове функции <code>nextval</code>
<code>lastval()</code>	<code>bigint</code>	Выдает значение любой последовательности, которое было возвращено при последнем вызове функции <code>nextval</code>
<code>nextval(regclass)</code>	<code>bigint</code>	Продвигает последовательность к следующему значению и возвращает его
<code>setval(regclass, bigint)</code>	<code>bigint</code>	Устанавливает текущее значение последовательности
<code>setval(regclass, bigint, boolean)</code>	<code>bigint</code>	Устанавливает текущее значение последовательности и флаг <code>is_called</code> , указывающий на то, что это значение использовалось

Последовательность, к которой будет обращаться одна из этих функций, определяется аргументом `regclass`, задающим просто OID последовательности в системном каталоге `pg_class`. Вычислять этот OID вручную не нужно, так как процедура ввода данных `regclass` автоматически выполнит эту работу за вас. Просто запишите имя последовательности в апострофах, чтобы оно выглядело как строковая константа. Для совместимости с обычными именами SQL эта строка будет переведена в нижний регистр, если только она не заключена в кавычки. Например:

```
nextval('foo')      обращается к последовательности foo
nextval('FOO')      обращается к последовательности foo
nextval('«Foo»')    обращается к последовательности Foo
```

При необходимости имя последовательности можно дополнить именем схемы:

```
nextval('myschema.foo')    обращается к myschema.foo
nextval('«myschema».foo')  то же самое
nextval('foo')             ищет foo в пути поиска
```

Подробнее тип `regclass` описан в разделе 6.18. (Часть 1. Руководство оператора).

Ниже описаны все функции, предназначенные для работы с последовательностями:

- `nextval`

Продвигает последовательность к следующему значению и возвращает его. Это атомарная операция: если `nextval` вызывается одновременно в нескольких сеансах, в результате каждого вызова будут гарантированно получены разные значения.

Если последовательность создается с параметрами по умолчанию, успешные вызовы `nextval` получают очередные значения по возрастанию, начиная с 1. Другое поведение можно получить с помощью специальных параметров в команде `CREATE SEQUENCE`; подробнее это описано на странице описания команды.

Примечание. Во избежание блокирования параллельных транзакций, пытающихся получить значения одной последовательности, операция `nextval` никогда не откатывается; то есть, как только значение было выбрано, оно считается использованным и не будет возвращено снова. Это утверждение верно, даже когда окружающая транзакция впоследствии прерывается или вызывающий запрос никак не использует это значение. Например, команда `INSERT` с предложением `ON CONFLICT` вычислит кортеж, претендующий на добавление, произведя все требуемые вызовы `nextval`, прежде чем выявит конфликты, которые могут привести к отработке правил `ON CONFLICT` вместо добавления. В таких ситуациях в последовательности задействованных значений могут образовываться «дыры». Таким образом, объекты последовательностей Синергия-БД *не годятся для получения непрерывных последовательностей*.

Этой функции требуется право `USAGE` или `UPDATE` для последовательности.

- `currval`

Возвращает значение, выданное при последнем вызове `nextval` для этой последовательности в текущем сеансе. Если в данном сеансе `nextval` ни разу не вызывалась для данной последовательности, возвращается ошибка. Так как это значение ограничено рамками сеанса, эта функция выдает предсказуемый результат вне зависимости от того, вызывалась ли впоследствии `nextval` в других сеансах или нет.

Этой функции требуется право `USAGE` или `SELECT` для последовательности.

- `lastval`

Возвращает значение, выданное при последнем вызове `nextval` в текущем сеансе. Эта функция подобна `currval`, но она не принимает в параметрах имя последовательности, а выдает значение для той последовательности, для которой `nextval` вызывалась в текущем сеансе в последний раз. Если в текущем сеансе функция `nextval` ещё не вызывалась, при вызове `lastval` произойдет ошибка.

Этой функции требуется право `USAGE` или `SELECT` для последней использованной последовательности.

- `setval`

Сбрасывает счетчик последовательности. В форме с двумя параметрами устанавливает для последовательности заданное значение поля `last_value` и значение `true` для флага `is_called`, показывающего, что при следующем вызове `nextval` последовательность должна сначала продвинуться к очередному значению, которое будет возвращено. При этом `currval` также возвратит заданное значение. В форме с тремя параметрами флагу `is_called` можно присвоить `true` или `false`. Со значением `true` она действует так же, как и форма с двумя параметрами. Если же присвоить этому флагу значение `false`, первый вызов `nextval` после этого вернет именно заданное значение, а продвижение последовательности произойдет при последующем вызове `nextval`. Кроме того, значение, возвращаемое `currval` в этом случае, не меняется. Например:

<code>SELECT setval('foo', 42);</code>	Следующий вызов
<code>nextval</code> вернет 43	
<code>SELECT setval('foo', 42, true);</code>	То же самое
<code>SELECT setval('foo', 42, false);</code>	Следующий вызов
<code>nextval</code> вернет 42	

Результатом самой функции `setval` будет просто значение её второго аргумента.

Примечание. Так как значения последовательностей изменяются вне транзакций, действие функции `setval` не отменяется при откате транзакции. Этой функции требуется право `UPDATE` для последовательности.

1.18. Условные выражения

В этом разделе описаны SQL-совместимые условные выражения, которые поддерживаются в Синергия-БД.

Примечание. Если возможностей этих условных выражений оказывается недостаточно, вероятно, имеет смысл перейти к написанию серверных функций на более мощном языке программирования.

1.18.1. CASE

Выражение CASE в SQL представляет собой общее условное выражение, напоминающее операторы if/else в других языках программирования:

```
CASE WHEN условие THEN результат
      [WHEN ...]
      [ELSE результат]
END
```

Предложения CASE можно использовать везде, где допускаются выражения. Каждое условие в нем представляет собой выражение, возвращающее результат типа boolean. Если результатом выражения оказывается true, значением выражения CASE становится результат, следующий за условием, а остальная часть выражения CASE не вычисляется. Если же условие не выполняется, за ним таким же образом проверяются все последующие предложения WHEN. Если не выполняется ни одно из условий WHEN, значением CASE становится результат, записанный в предложении ELSE. Если при этом предложение ELSE отсутствует, результатом выражения будет NULL.

Пример.

```
SELECT * FROM test;
```

```
  a
---
  1
  2
  3
```

```
SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
FROM test;
```

```
a | case
---+-----
1 | one
2 | two
3 | other
```

Типы данных всех выражений результатов должны приводиться к одному выходному типу.

Существует также «простая» форма выражения CASE, разновидность вышеприведенной общей формы:

```
CASE выражение
  WHEN значение THEN результат
  [WHEN ...]
  [ELSE результат]
END
```

В такой форме сначала вычисляется первое выражение, а затем его результат сравнивается с выражениями значений в предложениях WHEN, пока не будет найдено равное ему. Если такого значения не находится, возвращается результат предложения ELSE (или NULL). Эта форма больше похожа на оператор switch, существующий в языке C.

Показанный ранее пример можно записать по-другому, используя простую форму CASE:

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
             WHEN 2 THEN 'two'
             ELSE 'other'
       END
FROM test;
```

```
a | case
---+-----
1 | one
```

```
2 | two
3 | other
```

В выражении CASE вычисляются только те подвыражения, которые необходимы для получения результата. Например, так можно избежать ошибки деления на ноль:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE
false END;
```

Примечание. Как было описано в подразделе 2.2.14. (Часть 1. Руководство оператора), все же возможны ситуации, когда подвыражения вычисляются на разных этапах, так что железной гарантии, что в «CASE вычисляются только необходимые подвыражения», в принципе нет. Например, константное подвыражение 1/0 обычно вызывает ошибку деления на ноль на этапе планирования, хотя эта ветвь CASE может вовсе не вычисляться во время выполнения.

1.18.2. COALESCE

COALESCE(значение [, ...])

Функция COALESCE возвращает первый попавшийся аргумент, отличный от NULL. Если же все аргументы равны NULL, результатом тоже будет NULL. Это часто используется при отображении данных для подстановки некоторого значения по умолчанию вместо значений NULL:

```
SELECT COALESCE(description, short_description,
' (none) ') ...
```

Этот запрос вернет значение description, если оно не равно NULL, либо short_description, если оно не NULL, и строку (none), если оба эти значения равны NULL.

Как и выражение CASE, COALESCE вычисляет только те аргументы, которые необходимы для получения результата; то есть, аргументы правее первого отличного от NULL аргумента не вычисляются. Эта функция соответствует стандарту SQL, а в некоторых других СУБД её аналоги называются NVL и IFNULL.

1.18.3. NULLIF

NULLIF(значение1, значение2)

Функция NULLIF возвращает значение NULL, если значение1 равно значение2; в противном случае она возвращает значение1. Это может быть

полезно для реализации обратной операции к COALESCE. В частности, для примера, показанного выше:

```
SELECT NULLIF(value, ' (none) ') ...
```

Если аргумент `value` равен `(none)`, результатом выражения будет `NULL`, а в противном случае – значение аргумента `value`.

1.18.4. GREATEST и LEAST

```
GREATEST(значение [, ...])
```

```
LEAST(значение [, ...])
```

Функции `GREATEST` и `LEAST` выбирают наибольшее или наименьшее значение из списка выражений. Все эти выражения должны приводиться к общему типу данных, который станет типом результата. Значения `NULL` в этом списке игнорируются, так что результат выражения будет равен `NULL`, только если все его аргументы равны `NULL`.

Заметьте, что функции `GREATEST` и `LEAST` не описаны в стандарте SQL, но часто реализуются в СУБД как расширения. В некоторых других СУБД они могут возвращать `NULL`, когда не все, а любой из аргументов равен `NULL`.

1.19. Функции и операторы для работы с массивами

В таблице 1.47 показаны операторы, предназначенные для работы с массивами.

Таблица 1.47 – Операторы для работы с массивами

Оператор	Описание	Пример	Результат
<code>=</code>	равно	<code>ARRAY[1.1, 2.1, 3.1]::int[] = ARRAY[1, 2, 3]</code>	t
<code><></code>	не равно	<code>ARRAY[1, 2, 3] <> ARRAY[1, 2, 4]</code>	t
<code><</code>	меньше	<code>ARRAY[1, 2, 3] < ARRAY[1, 2, 4]</code>	t
<code>></code>	больше	<code>ARRAY[1, 4, 3] > ARRAY[1, 2, 4]</code>	t
<code><=</code>	меньше или равно	<code>ARRAY[1, 2, 3] <= ARRAY[1, 2, 3]</code>	t
<code>>=</code>	больше или равно	<code>ARRAY[1, 4, 3] >= ARRAY[1, 4, 3]</code>	t
<code>@></code>	содержит	<code>ARRAY[1, 4, 3] @> ARRAY[3, 1]</code>	t
<code><@</code>	содержится в	<code>ARRAY[2, 7] <@ ARRAY[1, 7, 4, 2, 6]</code>	t
<code>&&</code>	пересечение (есть общие элементы)	<code>ARRAY[1, 4, 3] && ARRAY[2, 1]</code>	t
<code> </code>	соединение массива с массивом	<code>ARRAY[1, 2, 3] ARRAY[4, 5, 6]</code>	{1, 2, 3, 4, 5, 6}

Окончание таблицы 1.47

Оператор	Описание	Пример	Результат
	соединение массива с массивом	ARRAY[1,2,3] ARRAY[[4,5,6], [7,8,9]]	{{1,2,3},{4,5,6}, {7,8,9}}
	соединение элемента с массивом	3 ARRAY[4,5,6]	{3,4,5,6}
	соединение массива с элементом	ARRAY[4,5,6] 7	{4,5,6,7}

Операторы сравнения массивов сравнивают содержимое массивов по элементам, используя при этом функцию сравнения для В-дерева, определенную для типа данного элемента по умолчанию. В многомерных массивах элементы просматриваются по строкам (индекс последней размерности меняется в первую очередь). Если содержимое двух массивов совпадает, а размерности различаются, результат их сравнения будет определяться первым отличием в размерностях.

В таблице 1.48 перечислены функции, предназначенные для работы с массивами. Дополнительная информация о них и примеры использования приведены в разделе 6.15. (Часть 1. Руководство оператора).

Таблица 1.48 – Функции для работы с массивами

Функция	Тип результата	Описание	Пример	Результат
array_append(anyarray, anyelement)	anyarray	добавляет элемент в конец массива	array_append(ARRAY[1,2], 3)	{1,2,3}
array_cat(anyarray, anyarray)	anyarray	соединяет два массива	array_cat(ARRAY[1,2,3], ARRAY[4,5])	{1,2,3,4,5}
array_ndims(anyarray)	int	возвращает число размерностей массива	array_ndims(ARRAY[[1,2,3], [4,5,6]])	2
array_dims(anyarray)	text	возвращает текстовое представление размерностей массива	array_dims(ARRAY[[1,2,3], [4,5,6]])	[1:2][1:3]
array_fill(anyelement, int[], [, int[]])	anyarray	возвращает массив, заполненный заданным значением и имеющий указанные размерности, в которых нижняя граница может быть отлична от 1	array_fill(7, ARRAY[3], ARRAY[2])	[2:4]={7,7,7 }
array_length(anyarray, int)	int	возвращает длину указанной размерности массива	array_length(array[1,2,3], 1)	3
array_lower(anyarray, int)	int	возвращает нижнюю границу указанной размерности массива	array_lower(''[0:2]= {1,2,3}''::int[], 1)	0

Окончание таблицы 1.48

Функция	Тип результата	Описание	Пример	Результат
<code>array_position(anyarray, anyelement [, int])</code>	<code>int</code>	возвращает позицию первого вхождения второго аргумента в массиве, начиная с элемента, выбираемого третьим аргументом, либо с первого элемента (массив должен быть одномерным)	<code>array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon')</code>	2
<code>array_positions(anyarray, anyelement)</code>	<code>int[]</code>	возвращает массив с позициями всех вхождений второго аргумента в массиве, задаваемым первым аргументом (массив должен быть одномерным)	<code>array_positions(ARRAY['A','A','B','A'], 'A')</code>	{1,2,4}
<code>array_prepend(anyelement, anyarray)</code>	<code>anyarray</code>	вставляет элемент в начало массива	<code>array_prepend(1, ARRAY[2,3])</code>	{1,2,3}
<code>array_remove(anyarray, anyelement)</code>	<code>anyarray</code>	удаляет из массива все элементы, равные заданному значению (массив должен быть одномерным)	<code>array_remove(ARRAY[1,2,3,2], 2)</code>	{1,3}
<code>array_replace(anyarray, anyelement, anyelement)</code>	<code>anyarray</code>	заменяет в массиве все элементы, равные заданному значению, другим значением	<code>array_replace(ARRAY[1,2,5,4], 5, 3)</code>	{1,2,3,4}
<code>array_to_string(anyarray, text [, text])</code>	<code>text</code>	выводит элементы массива через заданный разделитель и позволяет определить замену для значения NULL	<code>array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*')</code>	1,2,3,*,5
<code>array_upper(anyarray, int)</code>	<code>int</code>	возвращает верхнюю границу указанной размерности массива	<code>array_upper(ARRAY[1,8,3,7], 1)</code>	4
<code>cardinality(anyarray)</code>	<code>int</code>	возвращает общее число элементов в массиве, либо 0, если массив пуст	<code>cardinality(ARRAY[[1,2],[3,4]])</code>	4
<code>string_to_array(text, text [, text])</code>	<code>text[]</code>	разбивает строку на элементы массива, используя заданный разделитель и, возможно, замену для значений NULL	<code>string_to_array('xx~^~yy~^~zz', '~^~', 'yy')</code>	{xx,NULL,zz}
<code>unnest(anyarray)</code>	<code>setof anyelement</code>	разворачивает массив в набор строк	<code>unnest(ARRAY[1,2])</code>	1 2 (2 строки)
<code>unnest(anyarray, anyarray [, ...])</code>	<code>setof anyelement, anyelement [, ...]</code>	разворачивает массивы (возможно разных типов) в набор строк. Это допускается только в предложении FROM.	<code>unnest(ARRAY[1,2], ARRAY['foo','bar','baz'])</code>	1 foo 2 bar NULL baz (3 строки)

В функциях `array_position` и `array_positions` каждый элемент массива сравнивается с искомым значением по принципу `IS NOT DISTINCT FROM`.

Функция `array_position` возвращает `NULL`, если искомое значение не находится.

Функция `array_positions` возвращает `NULL`, только если в качестве массива передаётся `NULL`; если же в массиве не находится значение, она возвращает пустой массив.

Если для функции `string_to_array` в качестве разделителя задан `NULL`, каждый символ входной строки станет отдельным элементов в полученном массиве. Если разделитель пустая строка, строка будет возвращена целиком в массиве из одного элемента. В противном случае входная строка разбивается по вхождением подстроки, указанной в качестве разделителя.

Если для функции `string_to_array` параметр замены значения `NULL` опущен или равен `NULL`, никакие подстроки во входных данных не будут заменяться на `NULL`. Если же параметр замены `NULL` опущен или равен `NULL` для функции `array_to_string`, все значения `NULL` просто пропускаются и никак не представляются в выходной строке.

Вы также можете узнать об агрегатной функции, работающей с массивами, `array_agg` в разделе 1.20.

1.20. Диапазонные функции и операторы

Диапазонные типы данных рассматриваются в разделе 6.17. (Часть 1. Руководство оператора).

В таблице 1.49 показаны операторы, предназначенные для работы с диапазонами.

Таблица 1.49 – Диапазонные операторы

Оператор	Описание	Пример	Результат
=	равно	<code>int4range(1,5) = '[1,4]':::int4range</code>	t
<>	не равно	<code>numrange(1.1,2.2) <> numrange(1.1,2.3)</code>	t
<	меньше	<code>int4range(1,10) < int4range(2,3)</code>	t

Окончание таблицы 1.49

Оператор	Описание	Пример	Результат
>	больше	<code>int4range(1,10) > int4range(1,5)</code>	t
<=	меньше или равно	<code>numrange(1.1,2.2) <= numrange(1.1,2.2)</code>	t
>=	больше или равно	<code>numrange(1.1,2.2) >= numrange(1.1,2.0)</code>	t
@>	содержит диапазон	<code>int4range(2,4) @> int4range(2,3)</code>	t
@>	содержит элемент	<code>'[2011-01-01,2011-03-01)':'::tsrange @> '2011-01-10'::timestamp</code>	t
<@	диапазон содержится в	<code>int4range(2,4) <@ int4range(1,7)</code>	t
<@	элемент содержится в	<code>42 <@ int4range(1,7)</code>	f
&&	пересекает (есть общие точки)	<code>int8range(3,7) && int8range(4,12)</code>	t
<<	строго слева от	<code>int8range(1,10) << int8range(100,110)</code>	t
>>	строго справа от	<code>int8range(50,60) >> int8range(20,30)</code>	t
&<	не простирается правее	<code>int8range(1,20) &< int8range(18,20)</code>	t
&>	не простирается левее	<code>int8range(7,20) &> int8range(5,10)</code>	t
- -	примыкает к	<code>numrange(1.1,2.2) - - numrange(2.2,3.3)</code>	t
+	union	<code>numrange(5,15) + numrange(10,20)</code>	[5,20)
*	пересечение	<code>int8range(5,15) * int8range(10,20)</code>	[10,15)
-	вычитание	<code>int8range(5,15) - int8range(10,20)</code>	[5,10)

Простые операторы сравнения <, >, <= и >= сначала сравнивают нижние границы, и только если они равны, сравнивают верхние. Эти операторы сравнения обычно не очень полезны для диапазонов; основное их предназначение – сделать возможным построение индексов B-деревьев по диапазонам.

Операторы слева/справа/примыкает всегда возвращают false, если один из диапазонов пуст; то есть, считается, что пустой диапазон находится не слева и не справа от какого-либо другого диапазона.

Операторы сложения и вычитания вызывают ошибку, если получающийся в результате диапазон оказывается состоящим из двух разделенных поддиапазонов, так как его нельзя представить в этом типе данных.

В таблице 1.50 перечислены функции, предназначенные для работы с диапазонными типами.

Таблица 1.50 – Диапазонные функции

Функция	Тип результата	Описание	Пример	Результат
<code>lower(anyrange)</code>	тип элемента диапазона	нижняя граница диапазона	<code>lower(numrange(1.1, 2.2))</code>	1.1
<code>upper(anyrange)</code>	тип элемента диапазона	верхняя граница диапазона	<code>upper(numrange(1.1, 2.2))</code>	2.2
<code>isempty(anyrange)</code>	boolean	диапазон пуст?	<code>isempty(numrange(1.1, 2.2))</code>	false
<code>lower_inc(anyrange)</code>	boolean	нижняя граница включается?	<code>lower_inc(numrange(1.1, 2.2))</code>	true

Окончание таблицы 1.50

Функция	Тип результата	Описание	Пример	Результат
<code>upper_inc(anyrange)</code>	boolean	верхняя граница включается?	<code>upper_inc(numrange(1.1, 2.2))</code>	false
<code>lower_inf(anyrange)</code>	boolean	нижняя граница равна бесконечности?	<code>lower_inf('(',')'::daterange)</code>	true
<code>upper_inf(anyrange)</code>	boolean	верхняя граница равна бесконечности?	<code>upper_inf('(',')'::daterange)</code>	true
<code>range_merge(anyrange, anyrange)</code>	anyrange	наименьший диапазон, включающий оба заданных диапазона	<code>range_merge('[1,2]'::int4range, '[3,4]'::int4range)</code>	[1,4)

Функции `lower` и `upper` возвращают NULL, если диапазон пуст или указанная граница равна бесконечности. Если же пустой диапазон передается функциям `lower_inc`, `upper_inc`, `lower_inf` и `upper_inf`, все они возвращают false.

1.21.Агрегатные функции

Агрегатные функции получают единственный результат из набора входных значений. Встроенные обычные агрегатные функции перечислены в таблице 1.51 и таблице 1.52, а сортирующие агрегатные функции – в таблице 1.53 и таблице 1.54. Особенности синтаксиса агрегатных функций разъясняются в подразделе 2.2.7. (Часть 1. Руководство оператора).

Таблица 1.51 – Агрегатные функции общего назначения

Функция	Типы аргумента	Тип результата	Частичный режим	Описание
<code>array_agg(выражение)</code>	любой тип не массива	массив элементов с типом аргумента	Нет	входные значения, включая NULL, объединяются в массив
<code>array_agg(выражение)</code>	любой тип массива	тот же, что и тип аргумента	Нет	входные массивы собираются в массив большей размерности (они должны иметь одну размерность и не могут быть пустыми или равны NULL)
<code>avg(выражение)</code>	smallint, int, bigint, real, double precision, numeric или interval	numeric для любых целочисленных аргументов, double precision для аргументов с плавающей точкой, в противном случае тип данных аргумента	Да	арифметическое среднее для всех входных значений

Окончание таблицы 1.51

Функция	Типы аргумента	Тип результата	Частичный режим	Описание
bit_and(выражение)	smallint, int, bigint или bit	тот же, что и тип аргумента	Да	побитовое И для всех входных значений, не равных NULL, или NULL, если таких нет
bit_or(выражение)	smallint, int, bigint или bit	тот же, что и тип аргумента	Да	побитовое ИЛИ для всех входных значений, не равных NULL, или NULL, если таких нет
bool_and(выражение)	bool	bool	Да	true, если все входные значения равны true, и false в противном случае
bool_or(выражение)	bool	bool	Да	true, если хотя бы одно входное значение равно true, и false в противном случае
count(*)		bigint	Да	количество входных строк
count(выражение)	any	bigint	Да	количество входных строк, для которых значение выражения не равно NULL
every(выражение)	bool	bool	Да	синоним bool_and
json_agg(выражение)	any	json	Нет	агрегирует значения в виде массива JSON
jsonb_agg(выражение)	any	jsonb	Нет	агрегирует значения в виде массива JSON
json_object_agg(имя, значение)	(any, any)	json	Нет	агрегирует пары имя/значение в виде объекта JSON
jsonb_object_agg(имя, значение)	(any, any)	jsonb	Нет	агрегирует пары имя/значение в виде объекта JSON
max(выражение)	любой числовой, строковый, сетевой тип или тип даты/времени, либо массив этих типов	тот же, что и тип аргумента	Да	максимальное значение выражения среди всех входных данных
min(выражение)	любой числовой, строковый, сетевой тип или тип даты/времени, либо массив этих типов	тот же, что и тип аргумента	Да	минимальное значение выражения среди всех входных данных
string_agg(выражение, разделитель)	(text, text) или (bytea, bytea)	тот же, что и типы аргументов	Нет	входные данные складываются в строку через заданный разделитель
sum(выражение)	smallint, int, bigint, real, double precision, numeric, interval или money	bigint для аргументов smallint или int, numeric для аргументов bigint, и тип аргумента в остальных случаях	Да	сумма значений выражения по всем входным данным
xmlagg(выражение)	xml	xml	Нет	соединение XML-значений (см. также подраздел 1.15.1.7)

Следует заметить, что за исключением `count`, все эти функции возвращают `NULL`, если для них не была выбрана ни одна строка. В частности, функция `sum`, не получив строк, возвращает `NULL`, а не 0, как можно было бы ожидать, и `array_agg` в этом случае возвращает `NULL`, а не пустой массив. Если необходимо, подставить в результат 0 или пустой массив вместо `NULL` можно с помощью функции `coalesce`.

Агрегатные функции, поддерживающие *частичный режим*, являются кандидатами на участие в различных оптимизациях, например, в параллельном агрегировании.

Примечания:

1. Булевы агрегатные функции `bool_and` и `bool_or` соответствуют стандартным SQL-агрегатам `every` и `any` или `some`. Что касается `any` и `some`, по стандарту их синтаксис допускает некоторую неоднозначность:

```
SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
```

Здесь `ANY` можно рассматривать и как объявление подзапроса, и как агрегатную функцию, если этот подзапрос возвращает одну строку с булевым значением. Таким образом, этим агрегатным функциям нельзя было дать стандартные имена.

2. Пользователи с опытом использования других СУБД SQL могут быть недовольны скоростью агрегатной функции `count`, когда она применяется ко всей таблице. Подобный запрос:

```
SELECT count(*) FROM sometable;
```

потребуется затрат в количестве, пропорциональном размеру таблицы: Синергия-БД придется полностью просканировать либо всю таблицу, либо один из индексов, включающий все её строки.

Агрегатные функции `array_agg`, `json_agg`, `jsonb_agg`, `json_object_agg`, `jsonb_object_agg`, `string_agg` и `xmlagg` так же, как и подобные пользовательские агрегатные функции, выдают разные по содержанию результаты в зависимости от порядка входных значений. По умолчанию порядок не определён, но его можно задать, дополнив вызов агрегатной функции предложением `ORDER BY`. Обычно нужного результата также можно добиться, передав для агрегирования результат подзапроса с сортировкой. Например:

```
SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y  
DESC) AS tab;
```

Но учтите, что этот подход может не работать, если на внешнем уровне запроса выполняется дополнительная обработка, например, соединение, так как при этом результат подзапроса может быть переупорядочен перед вычислением

агрегатной функции.

В таблице 1.52 перечислены агрегатные функции, обычно применяемые в статистическом анализе. Они выделены просто для того, чтобы не загромождать список наиболее популярных агрегатных функций. В их описании под N подразумевается число входных строк, для которых входные выражения не равны NULL. Все эти функции возвращают NULL во всех случаях, когда вычисление бессмысленно, например, когда N равно 0.

Таблица 1.52 – Агрегатные функции для статистических вычислений

Функция	Тип аргумента	Тип результата	Частичный режим	Описание
<code>corr(Y, X)</code>	double precision	double precision	Да	коэффициент корреляции
<code>covar_pop(Y, X)</code>	double precision	double precision	Да	ковариация совокупности
<code>covar_samp(Y, X)</code>	double precision	double precision	Да	ковариация выборки
<code>regr_avgx(Y, X)</code>	double precision	double precision	Да	среднее независимой переменной ($\text{sum}(X) / N$)
<code>regr_avgy(Y, X)</code>	double precision	double precision	Да	среднее зависимой переменной ($\text{sum}(Y) / N$)
<code>regr_count(Y, X)</code>	double precision	bigint	Да	число входных строк, в которых оба выражения не NULL
<code>regr_intercept(Y, X)</code>	double precision	double precision	Да	пересечение с осью OY линии, полученной методом наименьших квадратов по данным (X, Y)
<code>regr_r2(Y, X)</code>	double precision	double precision	Да	квадрат коэффициента корреляции
<code>regr_slope(Y, X)</code>	double precision	double precision	Да	наклон линии, полученной методом наименьших квадратов по данным (X, Y)
<code>regr_sxx(Y, X)</code>	double precision	double precision	Да	$\text{sum}(X^2) - \text{sum}(X)^2 / N$ («сумма квадратов» независимой переменной)
<code>regr_sxy(Y, X)</code>	double precision	double precision	Да	$\text{sum}(X*Y) - \text{sum}(X) * \text{sum}(Y) / N$ («сумма произведений» независимых и зависимых переменных)
<code>regr_syy(Y, X)</code>	double precision	double precision	Да	$\text{sum}(Y^2) - \text{sum}(Y)^2 / N$ («сумма квадратов» зависимой переменной)

Окончание таблицы 1.52

Функция	Тип аргумента	Тип результата	Частичный режим	Описание
stddev (выражение)	smallint, int, bigint, real, double precision или numeric	double precision для аргументов с плавающей точкой, numeric для остальных	Да	сохранившийся синоним stddev_samp
stddev_pop (выражение)	smallint, int, bigint, real, double precision или numeric	double precision для аргументов с плавающей точкой, numeric для остальных	Да	стандартное отклонение по генеральной совокупности входных значений
stddev_samp (выражение)	smallint, int, bigint, real, double precision или numeric	double precision для аргументов с плавающей точкой, numeric для остальных	Да	стандартное отклонение по выборке входных значений
variance(выражение)	smallint, int, bigint, real, double precision или numeric	double precision для аргументов с плавающей точкой, numeric для остальных	Да	сохранившийся синоним var_samp
var_pop(выражение)	smallint, int, bigint, real, double precision или numeric	double precision для аргументов с плавающей точкой, numeric для остальных	Да	дисперсия для генеральной совокупности входных значений (квадрат стандартного отклонения)
var_samp(выражение)	smallint, int, bigint, real, double precision или numeric	double precision для аргументов с плавающей точкой, numeric для остальных	Да	дисперсия по выборке для входных значений (квадрат отклонения по выборке)

В таблице 1.53 показаны некоторые агрегатные функции, использующие синтаксис сортирующих агрегатных функций. Иногда такие функции функциями называют функциями «обратного распределения».

Таблица 1.53 – Сортирующие агрегатные функции

Функция	Тип непосредственного аргумента	Тип агрегированного аргумента	Тип результата	Частичный режим	Описание
mode() WITHIN GROUP (ORDER BY выражение_сортировки)		любой сортируемый тип	тот же, что у выражения сортировки	Нет	возвращает значение, наиболее часто встречающееся во входных данных (если одинаково часто встречаются несколько значений, произвольно выбирается первое из них)

Окончание таблицы 1.53

Функция	Тип непосредственного аргумента	Тип агрегированного аргумента	Тип результата	Частичный режим	Описание
percentile_cont (дробь) WITHIN GROUP (ORDER BY выражение_сортировки)	double precision	double precision или interval	тот же, что у выражения сортировки	Нет	непрерывный процентиль: возвращает значение, соответствующее заданной дроби по порядку, интерполируя соседние входные значения, если необходимо
percentile_cont (дробь) WITHIN GROUP (ORDER BY выражение_сортировки)	double precision[]	double precision или interval	массив типа выражения сортировки	Нет	множественный непрерывный процентиль: возвращает массив результатов, соответствующих форме параметра дроби (для каждого элемента не NULL подставляется значение, соответствующее данному процентиллю)
percentile_disc (дробь) WITHIN GROUP (ORDER BY выражение_сортировки)	double precision	любой сортируемый тип	тот же, что у выражения сортировки	Нет	дискретный процентиль: возвращает первое значение из входных данных, позиция которого по порядку равна или превосходит указанную дробь
percentile_disc (дробь) WITHIN GROUP (ORDER BY выражение_сортировки)	double precision[]	любой сортируемый тип	массив типа выражения сортировки	Нет	множественный дискретный процентиль: возвращает массив результатов, соответствующих форме параметра дроби (для каждого элемента не NULL подставляется входное значение, соответствующее данному процентиллю)

Все агрегатные функции, перечисленные в таблице 1.53, игнорируют значения NULL при сортировке данных. Для функций, принимающих параметр дробь, значение этого параметра должно быть от 0 до 1; в противном случае возникает

ошибка. Однако если в этом параметре передается NULL, эти функции просто выдают NULL.

Все агрегатные функции, перечисленные в таблице 1.54, связаны с одноименными оконными функциями, определенными в разделе 1.21. В каждом случае их результат – значение, которое вернула бы связанная оконная функция для «гипотетической» строки, полученной из аргументов, если бы такая строка была добавлена в сортированную группу строк, которую образуют сортированные_аргументы.

Таблица 1.54 – Гипотезирующие агрегатные функции

Функция	Тип непосредственного аргумента	Тип агрегированного аргумента	Тип результата	Частичный режим	Описание
rank (аргументы) WITHIN GROUP (ORDER BY сортированные_аргументы)	VARIADIC «any»	VARIADIC «any»	bigint	Нет	ранг гипотетической строки, с пропусками повторяющихся строк
dense_rank (аргументы) WITHIN GROUP (ORDER BY сортированные_аргументы)	VARIADIC «any»	VARIADIC «any»	bigint	Нет	ранг гипотетической строки, без пропусков
percent_rank (аргументы) WITHIN GROUP (ORDER BY сортированные_аргументы)	VARIADIC «any»	VARIADIC «any»	double precision	Нет	относительный ранг гипотетической строки, от 0 до 1
cume_dist (аргументы) WITHIN GROUP (ORDER BY сортированные_аргументы)	VARIADIC «any»	VARIADIC «any»	double precision	Нет	относительный ранг гипотетической строки, от 1/N до 1

Для всех этих гипотезирующих агрегатных функций непосредственные аргументы должны соответствовать (по количеству и типу) сортированным_аргументам. В отличие от встроенных агрегатных функций, они не являются строгими, то есть не отбрасывают входные строки, содержащие NULL. Значения NULL сортируются согласно правилу, указанному в предложении ORDER BY.

Таблица 1.55 – Операции группировки

Функция	Тип результата	Описание
GROUPING (<i>аргументы...</i>)	integer	Целочисленная битовая маска, показывающая, какие аргументы не вошли в текущий набор группирования

Операции группировки применяются в сочетании с наборами для различения результирующих строк. Аргументы операции GROUPING на самом деле не вычисляются, но они должны в точности соответствовать выражениям, заданным в предложении GROUP BY на их уровне запроса. Биты назначаются справа налево (правый аргумент отражается в младшем бите); бит равен 0, если соответствующее выражение вошло в критерий группировки набора группирования, для которого сформирована строка результата, или 1 в противном случае. Например:

```
=>SELECT * FROM items_sold;
```

```
  make  | model | sales
-----+-----+-----
  Foo   |  GT   |    10
  Foo   |  Tour  |    20
  Bar   |  City  |    15
  Bar   |  Sport |     5
(4 rows)
```

```
=> SELECT make, model, GROUPING(make,model), sum(sales)
FROM items_sold GROUP BY ROLLUP(make,model);
```

```
  make  | model | grouping | sum
-----+-----+-----+-----
  Foo   |  GT   |          |  0 | 10
  Foo   |  Tour  |          |  0 | 20
  Bar   |  City  |          |  0 | 15
  Bar   |  Sport |          |  0 |  5
  Foo   |        |          |  1 | 30
  Bar   |        |          |  1 | 20
         |        |          |  3 | 50
(7 rows)
```

1.22.Оконные функции

Оконные функции дают возможность выполнять вычисления с набором строк, каким-либо образом, связанным с текущей строкой запроса. Подробнее узнать о синтаксисе можно в подразделе 2.2.8. (Часть 1. Руководство оператора).

Встроенные оконные функции перечислены в таблице 1.56. Заметьте, что эти функции должны вызываться именно как оконные, т.е. при вызове необходимо использовать предложение OVER.

В дополнение к этим функциям в качестве оконных можно использовать любые встроенные или пользовательские универсальные или статистические агрегатные функции (но не сортирующие и не гипотезирующие). Агрегатные функции работают как оконные, только когда за их вызовом следует предложение OVER; в противном случае они работают как обычные, не оконные функции и выдают для всего набора единственную строку.

Таблица 1.56 – Оконные функции общего назначения

Функция	Тип результата	Описание
row_number()	bigint	номер текущей строки в её разделе, начиная с 1
rank()	bigint	ранг текущей строки с пропусками; то же, что и row_number для первой родственной ей строки
dense_rank()	bigint	ранг текущей строки без пропусков; эта функция считает группы родственных строк
percent_rank()	double precision	относительный ранг текущей строки: (rank - 1) / (общее число строк раздела - 1)
cume_dist()	double precision	кумулятивное распределение: (число строк раздела, предшествующих или родственных текущей строке) / общее число строк раздела
ntile(число_групп integer)	integer	ранжирование по целым числам от 1 до значения аргумента так, чтобы размеры групп были максимально близки
lag(значение anyelement [, смещение integer [, по_умолчанию anyelement]])	тип аргумента значение	возвращает значение для строки, положение которой задаётся смещением от текущей строки к началу раздела; если такой строки нет, возвращается значение по_умолчанию (оно должно иметь тот же тип, что и значение). Оба параметра смещение и по_умолчанию вычисляются для текущей строки. Если они не указываются, то смещение считается равным 1, а по_умолчанию – NULL
lead(значение anyelement [, смещение integer [, по_умолчанию anyelement]])	тип аргумента значение	возвращает значение для строки, положение которой задаётся смещением от текущей строки к концу раздела; если такой строки нет, возвращается значение по_умолчанию (оно должно иметь тот же тип, что и значение). Оба параметра смещение и по_умолчанию вычисляются для текущей строки. Если они не указываются, то смещение считается равным 1, а по_умолчанию – NULL
first_value(значение any)	тип аргумента значение	возвращает значение, вычисленное для первой строки в рамке окна

Окончание таблицы 1.56

Функция	Тип результата	Описание
<code>last_value</code> (значение any)	тип аргумента значение	возвращает значение, вычисленное для последней строки в рамке окна
<code>nth_value</code> (значение any, <code>n integer</code>)	тип аргумента значение	возвращает значение, вычисленное в <i>n</i> -ой строке в рамке окна (считая с 1), или NULL, если такой строки нет

Результат всех функций, перечисленных в таблице 1.56, зависит от порядка сортировки, заданного предложением `ORDER BY` в определении соответствующего окна. Строки, которые являются одинаковыми с точки зрения сортировки `ORDER BY`, считаются родственными; четыре функции, вычисляющие ранг, реализованы так, что их результат будет одинаковым для любых двух родственных строк.

Заметьте, что функции `first_value`, `last_value` и `nth_value` рассматривают только строки в «рамке окна», которая по умолчанию содержит строки от начала раздела до последней родственной строки для текущей. Поэтому результаты `last_value` и иногда `nth_value` могут быть не очень полезны. В таких случаях можно переопределить рамку, добавив в предложение `OVER` подходящее указание (`RANGE` или `ROWS`). Подробнее эти указания описаны в подразделе 2.2.8 (Часть 1. Руководства оператора).

Когда в качестве оконной функции используется агрегатная, она обрабатывает строки в рамке текущей строки. Агрегатная функция с `ORDER BY` и определением рамки окна по умолчанию будет вычисляться как «бегущая сумма», что может не соответствовать желаемому результату. Чтобы агрегатная функция работала со всем разделом, следует опустить `ORDER BY` или использовать `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`. Используя другие указания в определении рамки, можно получить и другие эффекты.

Примечание. В стандарте SQL определены параметры `RESPECT NULLS` или `IGNORE NULLS` для функций `lead`, `lag`, `first_value`, `last_value` и `nth_value`. В Синергия-БД такие параметры не реализованы: эти функции ведут себя так, как положено в стандарте по умолчанию (или с подразумеваемым параметром `RESPECT NULLS`). Также функция `nth_value` не поддерживает предусмотренные стандартом параметры `FROM FIRST` и `FROM LAST`: реализовано только поведение по умолчанию (с подразумеваемым параметром `FROM FIRST`). Получить эффект параметра `FROM LAST` можно, изменив порядок `ORDER BY` на

обратный.

Функция `cume_dist` вычисляет процент строк раздела, которые меньше или равны текущей строке или родственным ей строкам, тогда как `percent_rank` вычисляет процент строк раздела, которые меньше текущей строки, в предположении, что текущая строка не относится к разделу.

1.23.Выражения подзапросов

В этом разделе описаны выражения подзапросов, которые реализованы в Часть Синергия-БД в соответствии со стандартом SQL. Все рассмотренные здесь формы выражений возвращает булевы значения (`true/false`).

1.23.1. EXISTS

`EXISTS` (подзапрос)

Аргументом `EXISTS` является обычный оператор `SELECT`, т.е. подзапрос. Выполнив запрос, система проверяет, возвращает ли он строки в результате. Если он возвращает минимум одну строку, результатом `EXISTS` будет «`true`», а если не возвращает ни одной – «`false`».

Подзапрос может обращаться к переменным внешнего запроса, которые в рамках одного вычисления подзапроса считаются константами.

Вообще говоря, подзапрос может выполняться не полностью, а завершаться, как только будет возвращена хотя бы одна строка. Поэтому в подзапросах следует избегать побочных эффектов (например, обращений к генераторам последовательностей); проявление побочного эффекта может быть непредсказуемым.

Так как результат этого выражения зависит только от того, возвращаются строки или нет, но не от их содержимого, список выходных значений подзапроса обычно не имеет значения. Как следствие, широко распространена практика, когда проверки `EXISTS` записываются в форме `EXISTS (SELECT 1 WHERE ...)`. Однако из этого правила есть и исключения, например с подзапросами с предложением `INTERSECT`.

Этот простой пример похож на внутреннее соединение по колонке `col2`, но он

выдает максимум одну строку для каждой строки в `tab1`, даже если в `tab2` ей соответствуют несколько строк:

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 =
tab1.col2);
```

1.23.2. IN

выражение `IN` (подзапрос)

В правой стороне этого выражения в скобках задается подзапрос, который должен возвращать ровно одну колонку. Вычисленное значение левого выражения сравнивается со значениями во всех строках, возвращенных подзапросом. Результатом всего выражения `IN` будет «true», если строка с таким значением находится, и «false» в противном случае (в том числе, когда подзапрос вообще не возвращает строк).

Заметьте, что если результатом выражения слева оказывается `NULL` или равных значений справа не находится, а хотя бы одно из значений справа равно `NULL`, конструкция `IN` возвращает `NULL`, а не `false`. Это соответствует принятым в SQL правилам сравнения переменных со значениями `NULL`.

Так же, как и с `EXISTS`, здесь не следует рассчитывать на то, что подзапрос будет всегда выполняться полностью.

конструктор_строки `IN` (подзапрос)

В левой части этой формы `IN` записывается конструктор строки. Справа в скобках записывается подзапрос, который должен вернуть ровно столько колонок, сколько содержит строка в выражении слева. Вычисленные значения левого выражения сравниваются построчно со значениями во всех строках, возвращенных подзапросом. Результатом всего выражения `IN` будет «true», если строка с такими значениями находится, и «false» в противном случае (в том числе, когда подзапрос вообще не возвращает строк).

Как обычно, значения `NULL` в строках обрабатываются при этом по принятым в SQL правилам сравнения. Две строки считаются равными, если все их соответствующие элементы не равны `NULL`, но равны между собой; неравными они

считаются, когда в них находятся элементы, не равные NULL, и не равные друг другу; в противном случае результат сравнения строк не определен (равен NULL). Если в результатах сравнения строк нет ни одного положительного, но есть хотя бы один NULL, результатом IN будет NULL.

1.23.3. NOT IN

выражение NOT IN (подзапрос)

Справа в скобках записывается подзапрос, который должен возвращать ровно одну колонку. Вычисленное значение левого выражения сравнивается со значением во всех строках, возвращенных подзапросом. Результатом всего выражения NOT IN будет «true», если находятся только несовпадающие строки (в том числе, когда подзапрос вообще не возвращает строк). Если же находится хотя бы одна подходящая строка, результатом будет «false».

Заметьте, что если результатом выражения слева оказывается NULL или равных значений справа не находится, а хотя бы одно из значений справа равно NULL, конструкция NOT IN возвращает NULL, а не true. Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

Так же, как и с EXISTS, здесь не следует рассчитывать на то, что подзапрос будет всегда выполняться полностью.

конструктор_строки NOT IN (подзапрос)

В левой части этой формы NOT IN записывается конструктор строки. Справа в скобках записывается подзапрос, который должен вернуть ровно столько колонок, сколько содержит строка в выражении слева. Вычисленные значения левого выражения сравниваются построчно со значениями во всех строках, возвращенных подзапросом. Результатом всего выражения NOT IN будет «true», если равных строк не найдется (в том числе, и когда подзапрос не возвращает строк), и «false», если такие строки есть.

Как обычно, значения NULL в строках обрабатываются при этом по принятым в SQL правилам сравнения. Две строки считаются равными, если все их соответствующие элементы не равны NULL, но равны между собой; неравными они

считаются, когда в них находятся элементы, не равные NULL, и не равные друг другу; в противном случае результат сравнения строк не определен (равен NULL). Если в результатах сравнения строк нет ни одного положительного, но есть хотя бы один NULL, результатом NOT IN будет NULL.

1.23.4. ANY/SOME

выражение оператор ANY (подзапрос)

выражение оператор SOME (подзапрос)

В правой части конструкции в скобках записывается подзапрос, который должен возвращать ровно одну колонку. Вычисленное значение левого выражения сравнивается со значением в каждой строке результата подзапроса с помощью заданного оператора условия, который должен выдавать логическое значение. Результатом ANY будет «true», если хотя бы для одной строки условие истинно, и «false» в противном случае (в том числе, и когда подзапрос не возвращает строк).

Ключевое слово SOME является синонимом ANY. Конструкцию IN можно записать также записать как = ANY.

Заметьте, что если условие не выполняется ни для одной из строк, а хотя бы для одной строки условный оператор выдает NULL, конструкция ANY возвращает NULL, а не false. Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

Так же, как и с EXISTS, здесь не следует рассчитывать на то, что подзапрос будет всегда выполняться полностью.

конструктор_строки оператор ANY (подзапрос)

конструктор_строки оператор SOME (подзапрос)

В левой части этой формы ANY записывается конструктор строки. Справа в скобках записывается подзапрос, который должен возвращать ровно столько колонок, сколько содержит строка в выражении слева. Вычисленные значения левого выражения сравниваются построчно со значениями во всех строках, возвращенных подзапросом, с применением заданного оператора. Результатом всего выражения ANY будет «true», если для какой-либо из строк результатом сравнения будет true, и «false», если для всех строк результатом сравнения оказывается false (в том числе, и

когда подзапрос не возвращает строк). Результат будет равен NULL, если сравнение не возвращает true ни для одной из строк, но как минимум для одной результат сравнения NULL.

1.23.5. ALL

выражение оператор ALL (подзапрос)

В правой части конструкции в скобках записывается подзапрос, который должен возвращать ровно одну колонку. Вычисленное значение левого выражения сравнивается со значением в каждой строке результата подзапроса с помощью заданного оператора условия, который должен выдавать логическое значение. Результатом ALL будет «true», если условие истинно для всех строк (и когда подзапрос не возвращает строк), и «false», если находятся строки, для которых оно ложно. Результат будет равен NULL, если сравнение не возвращает false ни для одной из строк, но как минимум для одной результат сравнения NULL.

Конструкция NOT IN равнозначна <> ALL.

Так же, как и с EXISTS, здесь не следует рассчитывать на то, что подзапрос будет всегда выполняться полностью.

конструктор_строки оператор ALL (подзапрос)

В левой части этой формы ALL записывается конструктор строки. Справа в скобках записывается подзапрос, который должен возвращать ровно столько колонок, сколько содержит строка в выражении слева. Вычисленные значения левого выражения сравниваются построчно со значениями во всех строках, возвращенных подзапросом, с применением заданного оператора. Результатом всего выражения ALL будет «true», если для всех строк подзапроса результатом сравнения будет true (или если подзапрос не возвращает строк), и «false», если результат сравнения равен false для любой из строк подзапроса. Результат будет равен NULL, если сравнение не возвращает false ни для одной из строк, но как минимум для одной результат сравнения NULL.

1.23.6. Сравнение единичных строк

конструктор_строки оператор (подзапрос)

В левой части конструкции записывается конструктор строки. Справа в скобках записывается подзапрос, который должен возвращать ровно столько колонок, сколько содержит строка в выражении слева. Более того, подзапрос может вернуть максимум одну строку. Если он не вернет строк, результатом будет NULL. Конструкция возвращает результат сравнения строки слева с этой одной строкой результата подзапроса.

1.24. Сравнение табличных строк и массивов

В этом разделе описываются несколько специальных конструкций, позволяющих сравнивать группы значений. Синтаксис этих конструкций связан с формами выражений с подзапросами, описанными в предыдущем разделе, а отличаются они отсутствием подзапросов. Конструкции, в которых в качестве подвыражений используются массивы, являются расширениями Синергия-БД; все остальные формы соответствуют стандарту SQL. Все описанные здесь выражения возвращают логические значения (true/false).

1.24.1. IN

выражение IN (значение [, ...])

Справа в скобках записывается список скалярных выражений. Результатом будет «true», если значение левого выражения равняется одному из значений выражений в правой части. Эту конструкцию можно считать краткой записью условия:

```
выражение = значение1  
OR  
выражение = значение2  
OR  
...
```

Заметьте, что если результатом выражения слева оказывается NULL или равных значений справа не находится, а хотя бы одно из значений справа равно NULL, конструкция IN возвращает NULL, а не false. Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

1.24.2. NOT IN

выражение NOT IN (значение [, ...])

Справа в скобках записывается список скалярных выражений. Результатом будет «true», если значение левого выражения не равно ни одному из значений выражений в правой части. Эту конструкцию можно считать краткой записью условия:

выражение <> значение1

AND

выражение <> значение2

AND

...

Заметьте, что если результатом выражения слева оказывается NULL или равных значений справа не находится, а хотя бы одно из значений справа равно NULL, конструкция NOT IN возвращает NULL, а не true, как можно было бы наивно полагать. Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

Примечание. Выражения $x \text{ NOT IN } y$ и $\text{NOT } (x \text{ IN } y)$ полностью равнозначны. Учитывая, что значения NULL могут ввести в заблуждение начинающих скорее в конструкции NOT IN, чем в IN, лучше формулировать условия так, чтобы в них было как можно меньше отрицаний.

1.24.3. ANY/SOME (с массивом)

выражение оператор ANY (выражение массива)

выражение оператор SOME (выражение массива)

Справа в скобках записывается выражение, результатом которого является массив. Вычисленное значение левого выражения сравнивается с каждым элементом этого массива с применением заданного оператора условия, который должен выдавать логическое значение. Результатом ANY будет «true», если для какого-либо элемента условие истинно, и «false» в противном случае (в том числе, и когда массив оказывается пустым).

Если значением массива оказывается NULL, результатом ANY также будет NULL. Если NULL получен в левой части, результатом ANY обычно тоже будет NULL

(хотя оператор нестрогого сравнения может выдать другой результат). Кроме того, если массив в правой части содержит элементы NULL и ни с одним из элементов условие не выполняется, результатом ANY будет NULL, а не false (опять же, если используется оператор строгого сравнения). Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

Ключевое слово SOME является синонимом ANY.

1.24.4. ALL (с массивом)

выражение оператор ALL (выражение массива)

Справа в скобках записывается выражение, результатом которого является массив. Вычисленное значение левого выражения сравнивается с каждым элементом этого массива с применением заданного оператора условия, который должен выдавать логическое значение. Результатом ALL будет «true», если для всех элементов условие истинно (или массив не содержит элементов), и «false», если находятся строки, для которых оно ложно.

Если значением массива оказывается NULL, результатом ALL также будет NULL. Если NULL получен в левой части, результатом ALL обычно тоже будет NULL (хотя оператор нестрогого сравнения может выдать другой результат). Кроме того, если массив в правой части содержит элементы NULL и при этом нет элементов, с которыми условие не выполняется, результатом ALL будет NULL, а не true (опять же, если используется оператор строгого сравнения). Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

1.24.5. Сравнение конструкторов строк

конструктор_строки оператор конструктор_строки

С обеих сторон представлены конструкторы строк. При этом данные строки должны содержать одинаковое число полей. После вычисления каждой стороны они сравниваются по строкам. Сравнения конструкторов строк возможны с оператором =, <>, <, <=, > или >=. Каждый элемент строки должен иметь тип, для которого определен класс операторов В-дерева; в противном случае при попытке сравнения может возникнуть ошибка.

Примечание. Ошибок, связанных с числом или типов элементов, не должно быть, если сравнение выполняется с ранее полученными колонками.

Сравнения = и <> несколько отличаются от других. С этими операторами две строки считаются равными, если все их соответствующие поля не равны NULL и равны между собой, и неравными, если какие-либо соответствующие их поля не NULL и не равны между собой. В противном случае результатом сравнения будет неопределенность (NULL).

С операторами <, <=, > и >= элементы строк сравниваются слева направо до тех пор, пока не будет найдена пара неравных элементов или значений NULL. Если любым из элементов пары оказывается NULL, результатом сравнения будет неопределенность (NULL), в противном случае результат всего выражения определяется результатом сравнения этих двух элементов. Например, результатом ROW(1, 2, NULL) < ROW(1, 3, 0) будет true, а не NULL, так как третья пара элементов не принимается в рассмотрение.

конструктор_строки IS DISTINCT FROM конструктор_строки

Эта конструкция похожа на сравнение строк с оператором <>, но со значениями NULL она выдаёт не NULL. Любое значение NULL для неё считается неравным (отличным от) любому значению не NULL, а два NULL считаются равными (не различными). Таким образом, результатом такого выражения будет true или false, но не NULL.

конструктор_строки IS NOT DISTINCT FROM
конструктор_строки

Эта конструкция похожа на сравнение строк с оператором =, но со значениями NULL она выдаёт не NULL. Любое значение NULL для неё считается неравным (отличным от) любому значению не NULL, а два NULL считаются равными (не различными). Таким образом, результатом такого выражения всегда будет true или false, но не NULL.

1.24.6. Сравнение составных типов

запись оператор запись

Стандарт SQL требует, чтобы при сравнении строк возвращался NULL, если

результат зависит от сравнения двух значений NULL или значения NULL и не NULL. Синергия-БД выполняет это требование только при сравнении строк, созданных конструкторами (как описано в подразделе 1.23.5.), или строки, созданной конструктором, со строкой результата подзапроса (как было описано в разделе 1.22). В других контекстах при сравнении полей составных типов два значения NULL считаются равными, а любое значение не NULL полагается меньшим NULL. Это отклонение от правила необходимо для полноценной реализации сортировки и индексирования составных типов.

После вычисления каждой стороны они сравниваются по строкам. Сравнения составных типов возможны с оператором =, <>, <, <=, > или >=, либо другим подобным. Точнее, оператором сравнения строк может быть любой оператор, входящий в класс операторов В-дерева, либо обратный к оператору =, входящему в класс операторов В-дерева. По умолчанию вышеперечисленные операторы действуют так же, как выражение IS [NOT] DISTINCT FROM для конструкторов строк (см. подраздел 1.24.5).

Для поддержки сравнения строк с элементами, для которых не определен класс операторов В-дерева по умолчанию, введены следующие операторы: *=, *<>, *<, *<=, *> и *>=. Эти операторы сравнивают внутреннее двоичное представление двух строк. Учтите, что две строки могут иметь различное двоичное представление, даже когда при сравнении оператором равенства считаются равными. Порядок строк с такими операторами детерминирован, но не несет смысловой нагрузки. Данные операторы применяются внутри системы для материализованных представлений и могут быть полезны для других специальных целей (например, репликации), но, вообще говоря, не предназначены для использования в обычных запросах.

1.25. Функции, возвращающие множества

В этом разделе описаны функции, которые могут возвращать не одну, а множество строк. Чаще всего из их числа используются функции, генерирующие ряды значений, которые перечислены в таблице 1.57 и таблице 1.58. Другие, более специализированные функции множеств описаны в других разделах этой

документации. Варианты комбинирования нескольких функций, возвращающих множества строк, описаны в подразделе 5.2.1.4. (Часть 1. Руководство оператора).

Таблица 1.57 – Функции, генерирующие ряды значений

Функция	Тип аргумента	Тип результата	Описание
<code>generate_series(start, stop)</code>	<code>int</code> , <code>bigint</code> или <code>numeric</code>	<code>setof int</code> , <code>setof bigint</code> или <code>setof numeric</code> (определяется типом аргумента)	Выдаёт ряд целых чисел от <code>start</code> до <code>stop</code> с шагом 1
<code>generate_series(start, stop, step)</code>	<code>int</code> , <code>bigint</code> или <code>numeric</code>	<code>setof int</code> , <code>setof bigint</code> или <code>setof numeric</code> (определяется типом аргумента)	Выдаёт ряд значений от <code>start</code> до <code>stop</code> с заданным шагом (<code>step</code>)
<code>generate_series(start, stop, step interval)</code>	<code>timestamp</code> или <code>timestamp with time zone</code>	<code>setof timestamp</code> или <code>setof timestamp with time zone</code> (определяется типом аргумента)	Выдаёт ряд значений от <code>start</code> до <code>stop</code> с заданным шагом (<code>step</code>)

Если заданный шаг (`step`) положительный, а `start` оказывается больше `stop`, эти функции возвращают 0 строк. Тот же результат будет, если `step` меньше 0, а `start` меньше `stop`, или если какой-либо аргумент равен `NULL`. Если же `step` будет равен 0, произойдёт ошибка. Несколько примеров:

```
SELECT * FROM generate_series(2,4);
```

```
generate_series
-----
                2
                3
                4
(3 rows)
```

```
SELECT * FROM generate_series(5,1,-2);
```

```
generate_series
-----
                5
                3
                1
(3 rows)
```

```
SELECT * FROM generate_series(4,3);
```

```
generate_series
-----
```

(0 rows)

```
SELECT generate_series(1.1, 4, 1.3);
generate_series
```

```
-----
                1.1
                2.4
                3.7
```

(3 rows)

-- этот пример задействует оператор прибавления к дате
целого числа

```
SELECT current_date + s.a AS dates FROM
generate_series(0,14,7) AS s(a);
dates
```

```
-----
2004-02-05
2004-02-12
2004-02-19
```

(3 rows)

```
SELECT * FROM generate_series('2008-03-01
00:00'::timestamp,
```

```
                                '2008-03-04 12:00', '10
hours');
```

```
generate_series
-----
2008-03-01 00:00:00
2008-03-01 10:00:00
2008-03-01 20:00:00
2008-03-02 06:00:00
2008-03-02 16:00:00
2008-03-03 02:00:00
2008-03-03 12:00:00
2008-03-03 22:00:00
```

2008-03-04 08:00:00

(9 rows)

Таблица 1.58 – Функции, генерирующие индексы массивов

Функция	Тип результата	Описание
<code>generate_subscripts(array anyarray, dim int)</code>	setof int	Выдает ряд значений для использования в качестве индекса данного массива.
<code>generate_subscripts(array anyarray, dim int, reverse boolean)</code>	setof int	Выдает ряд значений для использования в качестве индекса данного массива. Если параметр <code>reverse</code> равен <code>true</code> , значения выдаются от большего к меньшему.

Функция `generate_subscripts` позволяет упростить получение всего набора индексов для указанной размерности заданного массива. Она выдает 0 строк, если в массиве нет указанной размерности или сам массив равен `NULL` (хотя для элементов, равных `NULL`, индексы будут выданы, как и для любых других).

Взгляните на следующие примеры:

```
-- простой пример использования
```

```
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1)
AS s;
```

```
s
```

```
---
```

```
1
```

```
2
```

```
3
```

```
4
```

```
(4 rows)
```

```
-- для показанного массива получение индекса и обращение
-- к элементу по индексу выполняется с помощью
подзапроса
```

```
SELECT * FROM arrays;
```

```
a
```

```
-----
```

```
{-1,-2}
```

```
{100,200,300}
```

```
(2 rows)
```

```
SELECT a AS array, s AS subscript, a[s] AS value
FROM (SELECT generate_subscripts(a, 1) AS s, a FROM
arrays) foo;
```

array	subscript	value
{-1,-2}	1	-1
{-1,-2}	2	-2
{100,200,300}	1	100
{100,200,300}	2	200
{100,200,300}	3	300

(5 rows)

```
-- разворачивание двумерного массива
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
select $1[i][j]
      from generate_subscripts($1,1) g1(i),
           generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
unnest2
-----
      1
      2
      3
      4
(4 rows)
```

Когда после функции в предложении FROM добавляется WITH ORDINALITY, в выходные данные добавляется колонка типа bigint, числа в которой начинаются с 1 и увеличиваются на 1 для каждой строки, выданной функцией. В первую очередь это полезно для функций, возвращающих множества, например, unnest().

```
-- функция, возвращающая множество, с нумерацией
```

```
SELECT * FROM pg_ls_dir('.') WITH ORDINALITY AS t(ls,n);
```

```
ls | n
```

```
-----+-----
```

```
pg_serial | 1
pg_twophase | 2
postmaster.opts | 3
pg_notify | 4
postgresql.conf | 5
pg_tblspc | 6
logfile | 7
base | 8
postmaster.pid | 9
pg_ident.conf | 10
global | 11
pg_xact | 12
pg_snapshots | 13
pg_multixact | 14
PG_VERSION | 15
pg_wal | 16
pg_hba.conf | 17
pg_stat_tmp | 18
pg_subtrans | 19
```

(19 строк)

1.26. Системные информационные функции

В таблице 1.59 перечислен ряд функций, предназначенных для получения информации о текущем сеансе и системе.

В дополнение к перечисленным здесь функциям существуют также функции, связанные с подсистемой статистики, которые тоже предоставляют системную информацию.

Таблица 1.59 – Функции получения информации о сеансе

Имя	Тип результата	Описание
current_catalog	name	имя текущей базы данных (в стандарте SQL она называется «каталогом»)
current_database()	name	имя текущей базы данных

Окончание таблицы 1.59

Имя	Тип результата	Описание
current_query()	text	текст запроса, выполняемого в данный момент, в том виде, в каком его передал клиент (может состоять из нескольких операторов)
current_role	name	синоним current_user
current_schema()	name	имя текущей схемы
current_schemas(boolean)	name[]	имена схем в пути поиска, возможно включая схемы, добавляемые в него неявно
current_user	name	имя пользователя в текущем контексте выполнения
inet_client_addr()	inet	адрес удалённой стороны соединения
inet_client_port()	int	порт удалённой стороны соединения
inet_server_addr()	inet	адрес локальной стороны соединения
inet_server_port()	int	порт локальной стороны соединения
pg_backend_pid()	int	код серверного процесса, обслуживающего текущий сеанс
pg_blocking_pids(int)	int[]	идентификаторы процессов, не дающих серверному процессу с определённым ID получить блокировку
pg_conf_load_time()	timestamp with time zone	время загрузки конфигурации
pg_current_logfile([text])	text	имя файла главного журнала или журнала в заданном формате, который в настоящее время используется сборщиком сообщений
pg_my_temp_schema()	oid	OID временной схемы этого сеанса или 0, если её нет
pg_is_other_temp_schema(oid)	boolean	является ли заданная схема временной в другом сеансе?
pg_jit_available()	boolean	доступна ли JIT-компиляция в данном сеансе? Возвращает false, если параметр jit равен false.
pg_listening_channels()	setof text	имена каналов, по которым текущий сеанс принимает сигналы
pg_notification_queue_usage()	double	занятая доля очереди асинхронных уведомлений (0-1)
pg_postmaster_start_time()	timestamp with time zone	время запуска сервера
pg_safe_snapshot_blocking_pids(int)	int[]	Идентификаторы процессов, не дающих серверному процессу с определённым ID получить безопасный снимок
pg_trigger_depth()	int	текущий уровень вложенности в триггерах Синергия-БД (0, если эта функция вызывается (прямо или косвенно) не из тела триггера)
session_user	name	имя пользователя сеанса
user	name	синоним current_user
version()	text	информация о версии Синергия-БД. Также можно прочитать версию в машинно-ориентированном виде, обратившись к переменной server_version_num.

Примечание. Функции current_catalog, current_role, current_schema, current_user, session_user и user имеют особый синтаксический статус в SQL: они должны вызываться без скобок после имени. Синергия-БД позволяет добавить скобки в вызове current_schema, но не других функций.

Функция session_user обычно возвращает имя пользователя, установившего текущее соединение с базой данных, но суперпользователи могут изменить это имя, выполнив команду SET SESSION AUTHORIZATION. Функция current_user возвращает идентификатор пользователя, по которому будут проверяться его права. Обычно это тот же пользователь, что и пользователь сеанса,

но его можно сменить с помощью `SET ROLE`. Этот идентификатор также меняется при выполнении функций с атрибутом `SECURITY DEFINER`. На языке Unix пользователь сеанса называется «реальным», а текущий – «эффективным». Имена `current_role` и `user` являются синонимами `current_user`. В стандарте SQL `current_role` и `current_user` имеют разное значение, но в Синергия-БД они не различаются, так как пользователи и роли объединены в единую сущность.

Функция `current_schema` возвращает имя схемы, которая стоит первой в пути поиска (или `NULL`, если путь поиска пуст). Эта схема будет задействована при создании таблиц или других именованных объектов, если целевая схема не указана явно. Функция `current_schemas(boolean)` возвращает массив имен всех схем, находящихся в пути поиска. Ее логический параметр определяет, будут ли включаться в результат неявно добавляемые в путь поиска системные схемы, такие как `pg_catalog`.

Примечание. Путь поиска можно изменить во время выполнения следующей командой:

```
SET search_path TO схема [, схема, ...]
```

`pg_listening_channels` возвращает имена каналов, по которым текущий сеанс принимает сигналы. Подробнее это освещается в описании команды `LISTEN`.

Функция `inet_client_addr` возвращает IP-адрес текущего клиента, `inet_client_port` – номер его порта, `inet_server_addr` – IP-адрес сервера, по которому он принял подключение клиента, а `inet_server_port` – соответствующий номер порта. Все эти функции возвращают `NULL`, если текущее соединение устанавливается через сокет Unix-сокет

Функция `pg_blocking_pids` возвращает массив идентификаторов процессов сеансов, которые блокирует серверный процесс с указанным идентификатором, либо пустой массив, если такой серверный процесс не найден или не заблокирован. Один серверный процесс блокирует другой, если он либо удерживает блокировку, конфликтующую с блокировкой, запрашиваемой серверным процессом (жёсткая блокировка), либо ждёт блокировки, которая вызвала бы конфликт с запросом блокировки заблокированного процесса и находится перед ней в очереди ожидания (мягкая блокировка). При распараллеливании запросов эта

функция всегда выдаёт видимые клиентом идентификаторы процессов (то есть, результаты `pg_backend_pid`), даже если фактическая блокировка удерживается или ожидается дочерним рабочим процессом. Вследствие этого, в результатах могут оказаться дублирующие PID. Также заметьте, что когда конфликтующую блокировку удерживает подготовленная транзакция, в выводе этой функции она будет представлена нулевым ID процесса. Частые вызовы этой функции могут отразиться на производительности базы данных, так как ей нужен монопольный доступ к общему состоянию менеджера блокировок, хоть и на короткое время.

Функция `pg_conf_load_time` возвращает время (timestamp with time zone), когда в последний раз сервер загружал файлы конфигурации. Если текущий сеанс начался раньше, она возвращает время, когда эти файлы были перезагружены для данного сеанса, так что в разных сеансах это значение может немного различаться. В противном случае это будет время, когда файлы конфигурации считал главный процесс.

Функция `pg_current_logfile` возвращает в значении `text` путь к файлам журналов, в настоящее время используемым сборщиком сообщений. Этот путь состоит из каталога `log_directory` и имени файла журнала. Если сборщик сообщений отключён, возвращается значение `NULL`. Если ведутся несколько журналов в разных форматах, при вызове функции `pg_current_logfile` без аргументов возвращается путь файла, имеющего первый формат по порядку из следующего списка: `stderr`, `csvlog`. Если файл журнала имеет какой-то иной формат, возвращается `NULL`. Чтобы запросить файл в определённом формате, передайте либо `csvlog`, либо `stderr` в качестве значения необязательного параметра типа `text`. Если запрошенный формат не включён в `log_destination`, будет возвращено значение `NULL`. Функция `pg_current_logfile` отражает содержимое файла `current_logfiles`.

`pg_my_temp_schema` возвращает OID временной схемы текущего сеанса или 0, если такой нет (в рамках сеанса не создавались временные таблицы). `pg_is_other_temp_schema` возвращает `true`, если заданный OID относится к временной схеме другого сеанса. Это может быть полезно, например для

исключения временных таблиц других сеансов из общего списка при просмотре таблиц базы данных.

Функция `pg_listening_channels` возвращает набор имён каналов асинхронных уведомлений, на которые подписан текущий сеанс. Функция `pg_notification_queue_usage` возвращает долю от всего свободного пространства для уведомлений, в настоящее время занятую уведомлениями, ожидающими обработки, в виде значения `double` в диапазоне 0..1. За дополнительными сведениями обратитесь к `LISTEN` и `NOTIFY`.

`pg_postmaster_start_time` возвращает время (timestamp with time zone), когда был запущен сервер.

Функция `pg_safe_snapshot_blocking_pids` возвращает массив идентификаторов процессов сеансов, которые блокируют серверный процесс с указанным идентификатором (не дают получить ему безопасный снимок), либо пустой массив, если такой серверный процесс не найден или не заблокирован. Сеанс, выполняющий транзакцию уровня `SERIALIZABLE`, блокирует транзакцию `SERIALIZABLE READ ONLY DEFERRABLE`, не давая ей получить снимок, пока она не определит, что можно безопасно избежать установления предикатных блокировок. Частые вызовы этой функции могут отразиться на производительности базы данных, так как ей нужен доступ к общему состоянию менеджера предикатных блокировок, хоть и на короткое время.

Функция `version` возвращает строку, описывающую версию сервера Синергия-БД. Эту информацию также можно получить из переменной `server_version` или, в более машинно-ориентированном формате, из переменной `server_version_num`. При разработке программ следует использовать `server_version_num` либо `PQserverVersion`, а не разбирать текстовую версию.

В таблице 1.60 перечислены функции, позволяющую пользователю программно проверить свои права доступа к объектам. Подробнее о правах можно узнать в разделе 3.6. (Часть 1. Руководство оператора).

Таблица 1.60 – Функции для проверки прав доступа

Имя	Тип результата	Описание
<code>has_any_column_privilege(user, table, privilege)</code>	boolean	имеет ли пользователь указанное право для какого-либо столбца таблицы
<code>has_any_column_privilege(table, privilege)</code>	boolean	имеет ли текущий пользователь указанное право для какого-либо столбца таблицы
<code>has_column_privilege(user, table, column, privilege)</code>	boolean	имеет ли пользователь указанное право для столбца
<code>has_column_privilege(table, column, privilege)</code>	boolean	имеет ли текущий пользователь указанное право для столбца
<code>has_database_privilege(user, database, privilege)</code>	boolean	имеет ли пользователь указанное право для базы данных
<code>has_database_privilege(database, privilege)</code>	boolean	имеет ли текущий пользователь указанное право для базы данных
<code>has_foreign_data_wrapper_privilege(user, fdw, privilege)</code>	boolean	имеет ли пользователь указанное право для обёртки сторонних данных
<code>has_foreign_data_wrapper_privilege(fdw, privilege)</code>	boolean	имеет ли текущий пользователь указанное право для обёртки сторонних данных
<code>has_function_privilege(user, function, privilege)</code>	boolean	имеет ли пользователь указанное право для функции
<code>has_function_privilege(function, privilege)</code>	boolean	имеет ли текущий пользователь указанное право для функции
<code>has_language_privilege(user, language, privilege)</code>	boolean	имеет ли пользователь указанное право для языка
<code>has_language_privilege(language, privilege)</code>	boolean	имеет ли текущий пользователь указанное право для языка
<code>has_schema_privilege(user, schema, privilege)</code>	boolean	имеет ли пользователь указанное право для схемы
<code>has_schema_privilege(schema, privilege)</code>	boolean	имеет ли текущий пользователь указанное право для схемы
<code>has_sequence_privilege(user, sequence, privilege)</code>	boolean	имеет ли пользователь указанное право для последовательности
<code>has_sequence_privilege(sequence, privilege)</code>	boolean	имеет ли текущий пользователь указанное право для последовательности
<code>has_server_privilege(user, server, privilege)</code>	boolean	имеет ли пользователь указанное право для стороннего сервера
<code>has_server_privilege(server, privilege)</code>	boolean	имеет ли текущий пользователь указанное право для стороннего сервера
<code>has_table_privilege(user, table, privilege)</code>	boolean	имеет ли пользователь указанное право для таблицы
<code>has_table_privilege(table, privilege)</code>	boolean	имеет ли текущий пользователь указанное право для таблицы
<code>has_tablespace_privilege(user, tablespace, privilege)</code>	boolean	имеет ли пользователь указанное право для табличного пространства
<code>has_tablespace_privilege(tablespace, privilege)</code>	boolean	имеет ли текущий пользователь указанное право для табличного пространства
<code>has_type_privilege(user, type, privilege)</code>	boolean	имеет ли пользователь указанное право для типа
<code>has_type_privilege(type, privilege)</code>	boolean	имеет ли текущий пользователь указанное право для типа
<code>pg_has_role(user, role, privilege)</code>	boolean	имеет ли пользователь указанное право для роли
<code>pg_has_role(role, privilege)</code>	boolean	имеет ли текущий пользователь указанное право для роли
<code>row_security_active(table)</code>	boolean	включена ли для текущего пользователя защита на уровне строк для таблицы

`has_table_privilege` проверяет, может ли пользователь выполнять с таблицей заданные действия. В качестве идентификатора пользователя можно задать

его имя, OID (`pg_authid.oid`) или `public` (это будет указывать на псевдороль `PUBLIC`). Если этот аргумент опущен, подразумевается текущий пользователь (`current_user`). Таблицу можно указать по имени или по OID. Таким образом, фактически есть шесть вариантов функции `has_table_privilege`, различающихся по числу и типу аргументов. Когда указывается имя объекта, его можно дополнить именем схемы, если это необходимо. Интересующее право доступа записывается в виде текста и может быть одним из следующих: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES` и `TRIGGER`. Дополнительно к названию права можно добавить `WITH GRANT OPTION` и проверить, разрешено ли пользователю передавать это право другим. Кроме того, в одном параметре можно перечислить несколько названий прав через запятую, и тогда функция возвратит `true`, если пользователь имеет одно из этих прав. Регистр в названии прав не имеет значения, а между ними (но не внутри) разрешены пробельные символы. Пара примеров:

```
SELECT has_table_privilege('myschema.mytable',  
    'select');  
SELECT has_table_privilege('joe', 'mytable',  
    'INSERT, SELECT WITH GRANT OPTION');
```

`has_sequence_privilege` проверяет, может ли пользователь выполнять заданные действия с последовательностью. В определении аргументов эта функция аналогична `has_table_privilege`. Допустимые для нее права складываются из `USAGE`, `SELECT` и `UPDATE`.

`has_any_column_privilege` проверяет, может ли пользователь выполнять заданные действия с какой-либо колонкой таблицы. В определении аргументов эта функция аналогична `has_table_privilege`, а допустимые права складываются из `SELECT`, `INSERT`, `UPDATE` и `REFERENCES`. Заметьте, что любое из этих прав, назначенное на уровне таблицы, автоматически распространяется на все её колонки, так что `has_any_column_privilege` всегда возвращает `true`, если `has_table_privilege` дает положительный ответ для тех же аргументов. Но `has_any_column_privilege` возвращает `true` ещё и тогда, когда право

назначено только для некоторых колонок.

`has_column_privilege` проверяет, может ли пользователь выполнять заданные действия с колонкой таблицы. В определении аргументов эта функция аналогична `has_table_privilege`, с небольшим дополнением: колонку можно задать по имени или номеру атрибута. Для нее допустимые права складываются из `SELECT`, `INSERT`, `UPDATE` и `REFERENCES`. Заметьте, что любое из этих прав, назначенное на уровне таблицы, автоматически распространяется на все колонки таблицы.

`has_database_privilege` проверяет, может ли пользователь выполнять заданные действия с базой данных. В определении аргументов эта функция аналогична `has_table_privilege`. Для нее допустимые права складываются из `CREATE`, `CONNECT` и `TEMPORARY` (или `TEMP`, что равносильно `TEMPORARY`).

`has_function_privilege` проверяет, может ли пользователь обратиться к заданной функции. В определении аргументов эта функция аналогична `has_table_privilege`. Когда функция определяется не своим `OID`, а текстовой строкой, эта строка должна быть допустимой для вводимого значения типа `regprocedure`. Для этой функции допустимо только право `EXECUTE`. Например:

```
SELECT has_function_privilege('joeuser', 'myfunc(int,  
text)', 'execute');
```

`has_foreign_data_wrapper_privilege` проверяет, может ли пользователь обращаться к обертке сторонних данных. В определении аргументов она аналогична `has_table_privilege`. Для нее допустимо только право `USAGE`.

`has_language_privilege` проверяет, может ли пользователь обращаться к процедурному языку. В определении аргументов эта функция аналогична `has_table_privilege`. Для нее допустимо только право `USAGE`.

`has_schema_privilege` проверяет, может ли пользователь выполнять заданные действия со схемой. В определении аргументов эта функция аналогична `has_table_privilege`. Для нее допустимые права складываются из `CREATE` и `USAGE`.

`has_server_privilege` проверяет, может ли пользователь обращаться к

стороннему серверу. В определении аргументов она аналогична `has_table_privilege`. Для нее допустимо только право `USAGE`.

`has_tablespace_privilege` проверяет, может ли пользователь выполнять заданное действие в табличном пространстве. В определении аргументов эта функция аналогична `has_table_privilege`. Для нее допустимо только право `CREATE`.

`has_type_privilege` проверяет, может ли пользователь обратиться к типу определенным образом. Возможные аргументы аналогичны `has_table_privilege`. При указании типа текстовой строкой, а не по `OID`, допускаются те же входные значения, что и для типа данных `regtype`. Для неё допустимо только право `USAGE`.

`pg_has_role` проверяет, может ли пользователь выполнять заданные действия с ролью. В определении аргументов эта функция аналогична `has_table_privilege`, за исключением того, что именем пользователя не может быть `public`. Для нее допустимые права складываются из `MEMBER` и `USAGE`. `MEMBER` обозначает прямое или косвенное членство в данной роли (то есть наличие права выполнить команду `SET ROLE`), тогда как `USAGE` показывает, что пользователь получает все права роли сразу, без `SET ROLE`.

`row_security_active` проверяет, включена ли защита на уровне строк для указанной таблицы в контексте и окружении текущего пользователя (`current_user`). Таблицу можно задать по имени или `OID`.

В таблице 1.61 перечислены функции, определяющие видимость объекта с текущим путем поиска схем. К примеру, таблица считается видимой, если содержащая её схема включена в путь поиска и нет другой таблицы с тем же именем, которая была бы найдена по пути поиска раньше. Другими словами, к этой таблице можно будет обратиться просто по её имени, без явного указания схемы. Просмотреть список всех видимых таблиц можно так:

```
SELECT relname FROM pg_class WHERE  
pg_table_is_visible(oid);
```

Таблица 1.61 – Функции для определения видимости

Имя	Тип результата	Описание
<code>pg_collation_is_visible(collation_oid)</code>	boolean	видимо ли правило сортировки
<code>pg_conversion_is_visible(conversion_oid)</code>	boolean	видимо ли преобразование
<code>pg_function_is_visible(function_oid)</code>	boolean	видима ли функция
<code>pg_opclass_is_visible(opclass_oid)</code>	boolean	видим ли класс операторов
<code>pg_operator_is_visible(operator_oid)</code>	boolean	видим ли оператор
<code>pg_opfamily_is_visible(opclass_oid)</code>	boolean	видимо ли семейство операторов
<code>pg_statistics_obj_is_visible(stat_oid)</code>	boolean	видим ли объект статистики в пути поиска
<code>pg_table_is_visible(table_oid)</code>	boolean	видима ли таблица
<code>pg_ts_config_is_visible(config_oid)</code>	boolean	видима ли конфигурация текстового поиска
<code>pg_ts_dict_is_visible(dict_oid)</code>	boolean	видим ли словарь текстового поиска
<code>pg_ts_parser_is_visible(parser_oid)</code>	boolean	видим ли анализатор текстового поиска
<code>pg_ts_template_is_visible(template_oid)</code>	boolean	видим ли шаблон текстового поиска
<code>pg_type_is_visible(type_oid)</code>	boolean	видим ли тип (или домен)

Каждая из этих функций проверяет видимость объектов определенного типа. Заметьте, что `pg_table_is_visible` можно также использовать для представлений, индексов и последовательностей, `pg_type_is_visible` можно использовать и для доменов. Для функций и операторов объект считается видимым в пути поиска, если при просмотре пути не находится предшествующий ему другой объект с тем же именем и типами аргументов. Для классов операторов во внимание принимается и имя оператора, и связанный с ним метод доступа к индексу.

Всем этим функциям должен передаваться OID проверяемого объекта. Если вы хотите проверить объект по имени, удобнее использовать типы-псевдонимы OID (`regclass`, `regtype`, `regprocedure`, `regoperator`, `regconfig` или `regdictionary`), например:

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

Заметьте, что проверять таким способом имена без указания схемы не имеет большого смысла – если имя удастся распознать, значит и объект будет видимым.

В таблице 1.62 перечислены функции, извлекающие информацию из системных каталогов.

Таблица 1.62 – Функции для обращения к системным каталогам

Имя	Тип результата	Описание
<code>format_type(type_oid, typemod)</code>	text	получает имя типа данных в формате SQL

Продолжение таблицы 1.62

Имя	Тип результата	Описание
<code>pg_get_constraintdef(constraint_oid)</code>	text	получает определение ограничения
<code>pg_get_constraintdef(constraint_oid, pretty_bool)</code>	text	получает определение ограничения
<code>pg_get_expr(pg_node_tree, relation_oid)</code>	text	декомпилирует внутреннюю форму выражения, в предположении, что все переменные в нём ссылаются на таблицу или отношение, указанное вторым параметром
<code>pg_get_expr(pg_node_tree, relation_oid, pretty_bool)</code>	text	декомпилирует внутреннюю форму выражения, в предположении, что все переменные в нём ссылаются на таблицу или отношение, указанное вторым параметром
<code>pg_get_functiondef(func_oid)</code>	text	получает определение функции или процедуры
<code>pg_get_function_arguments(func_oid)</code>	text	получает список аргументов из определения функции или процедуры (со значениями по умолчанию)
<code>pg_get_function_identity_arguments(func_oid)</code>	text	получает список аргументов, идентифицирующий функцию или процедуру (без значений по умолчанию)
<code>pg_get_function_result(func_oid)</code>	text	получает предложение RETURNS для функции (для процедуры возвращает NULL)
<code>pg_get_indexdef(index_oid)</code>	text	получает команду CREATE INDEX для индекса
<code>pg_get_indexdef(index_oid, column_no, pretty_bool)</code>	text	получает команду CREATE INDEX для индекса или определение одного индексированного столбца, когда column_no не равен 0
<code>pg_get_keywords()</code>	setof record	получает список ключевых слов SQL по категориям
<code>pg_get_ruledef(rule_oid)</code>	text	получает команду CREATE RULE для правила
<code>pg_get_ruledef(rule_oid, pretty_bool)</code>	text	получает команду CREATE RULE для правила
<code>pg_get_serial_sequence(table_name, column_name)</code>	text	получает имя последовательности, используемой столбцом идентификации или столбцом serial
<code>pg_get_statisticsobjdef(statobj_oid)</code>	text	получает команду CREATE STATISTICS для объекта расширенной статистики
<code>pg_get_triggerdef(trigger_oid)</code>	text	получает команду CREATE [CONSTRAINT] TRIGGER для триггера
<code>pg_get_triggerdef(trigger_oid, pretty_bool)</code>	text	получает команду CREATE [CONSTRAINT] TRIGGER для триггера
<code>pg_get_userbyid(role_oid)</code>	name	получает имя роли по заданному OID
<code>pg_get_viewdef(view_name)</code>	text	получает команду SELECT, определяющую представление или материализованное представление (устаревшая функция)
<code>pg_get_viewdef(view_name, pretty_bool)</code>	text	получает команду SELECT, определяющую представление или материализованное представление (устаревшая функция)
<code>pg_get_viewdef(view_oid)</code>	text	получает команду SELECT, определяющую представление или материализованное представление
<code>pg_get_viewdef(view_oid, pretty_bool)</code>	text	получает команду SELECT, определяющую представление или материализованное представление
<code>pg_get_viewdef(view_oid, wrap_column_int)</code>	text	получает команду SELECT, определяющую представление или материализованное представление; при необходимости разбивает строки с полями, выходящие за wrap_int символов, подразумевая форматированный вывод

Окончание таблицы 1.62

Имя	Тип результата	Описание
<code>pg_index_column_has_property(index_oid, column_no, prop_name)</code>	boolean	проверяет, имеет ли столбец индекса заданное свойство
<code>pg_index_has_property(index_oid, prop_name)</code>	boolean	проверяет, имеет ли индекс заданное свойство
<code>pg_indexam_has_property(am_oid, prop_name)</code>	boolean	проверяет, имеет ли метод доступа индекса заданное свойство
<code>pg_options_to_table(rel_options)</code>	setof record	получает набор параметров хранилища в виде имя/значение
<code>pg_tablespace_databases(tablespace_oid)</code>	setof oid	получает или устанавливает OID баз данных, объекты которых содержатся в заданном табличном пространстве
<code>pg_tablespace_location(tablespace_oid)</code>	text	получает путь в файловой системе к местоположению заданного табличного пространства
<code>pg_typeof(any)</code>	regtype	получает тип данных любого значения
<code>collation for (any)</code>	text	получает правило сортировки для аргумента
<code>to_regclass(rel_name)</code>	regclass	получает OID указанного отношения
<code>to_regproc(func_name)</code>	regproc	получает OID указанной функции
<code>to_regprocedure(func_name)</code>	regprocedure	получает OID указанной функции
<code>to_regoper(имя_оператора)</code>	regoper	получает OID указанного оператора
<code>to_regoperator(имя_оператора)</code>	regoperator	получает OID указанного оператора
<code>to_regtype(type_name)</code>	regtype	получает OID указанного типа

`format_type` возвращает в формате SQL имя типа данных, определяемого по OID и, возможно, модификатору типа. Если модификатор неизвестен, вместо него можно передать NULL.

`pg_get_keywords` возвращает таблицу с ключевыми словами SQL, которые воспринимает сервер. Колонка `word` содержит ключевое слово, а `catcode` – код категории: U – не зарезервировано, C – имя колонки, T – имя типа или функции, R – зарезервировано. Колонка `catdesc` содержит возможно локализованное описание категории.

`pg_get_constraintdef`, `pg_get_indexdef`, `pg_get_ruledef`, `pg_get_statisticsobjdef` и `pg_get_triggerdef` восстанавливают команду, создававшую заданное ограничение, индекс, правило или триггер, соответственно. Учтите, что они возвращают не изначальный текст команды, а результат декомпиляции. `pg_get_expr` декомпилирует внутреннюю форму отдельного выражения, например значения по умолчанию для колонки. Это может быть полезно для изучения содержимого системных каталогов. Если выражение может содержать переменные, укажите во втором параметре OID отношения, на

который они ссылаются; если таких переменных нет, вместо OID можно передать 0. `pg_get_viewdef` восстанавливает запрос `SELECT`, определяющий представление. Многие из этих функций имеют две версии, одна из которых позволяет получить форматированный вывод (параметр `pretty_bool`). Форматированный текст легче читается, но нет гарантии, что он будет всегда восприниматься одинаково будущими версиями Синергия-БД, поэтому не следует применять форматирование при выгрузке метаданных. Если параметр `pretty_bool` равен `false`, эта версия функции выдает тот же результат, что и версия без параметров.

`pg_get_functiondef` возвращает полный оператор `CREATE OR REPLACE FUNCTION` для заданной функции. `pg_get_function_arguments` возвращает список аргументов функции, в виде достаточном для включения в команду `CREATE FUNCTION`. `pg_get_function_result` в дополнение возвращает готовое предложение `RETURNS` для функции. `pg_get_function_identity_arguments` возвращает список аргументов, достаточный для однозначной идентификации функции, в форме, допустимой, например для команды `ALTER FUNCTION`. Значения по умолчанию в этой форме опускаются.

`pg_get_serial_sequence` возвращает имя последовательности, связанной со столбцом, либо `NULL`, если такой последовательности нет. Для столбца идентификации это будет последовательность, связанная с ним внутренним образом. Для столбцов, имеющих один из последовательных типов (`serial`, `smallserial`, `bigserial`), это последовательность, созданная объявлением данного столбца. В последнем случае эту связь можно изменить или разорвать, воспользовавшись командой `ALTER SEQUENCE OWNED BY`. Возможно, эту функцию стоило назвать `pg_get_owned_sequence`; её существующее имя отражает тот факт, что она обычно используется со столбцами `serial` или `bigserial`. В первом параметре функции указывается имя таблицы, возможно дополненное схемой, а во втором имя столбца. Так как первый параметр может содержать имя схемы и таблицы, он воспринимается не как идентификатор в кавычках и поэтому по умолчанию приводится к нижнему регистру, тогда как имя столбца во втором параметре

воспринимается как заключённое в кавычки и в нём регистр символов сохраняется. Эта функция возвращает имя в виде, пригодном для передачи функциям, работающим с последовательностями. Обычно она применяется для получения текущего значения последовательности для столбца идентификации или последовательного столбца, например:

```
SELECT currval(pg_get_serial_sequence('sometable',
'id'));
```

pg_get_userbyid получает имя роли по её OID.

Функции pg_index_column_has_property, pg_index_has_property и pg_indexam_has_property показывают, обладает ли указанный столбец индекса, индекс или метод доступа индекса заданным свойством. Они возвращают NULL, если имя свойства неизвестно или неприменимо к конкретному объекту, либо если OID или номер столбца не указывают на действительный объект. Описание свойств столбцов вы можете найти в таблице 1.63, свойства индексов описаны в таблице 1.64, а свойства методов доступа – в таблице 1.65. Заметьте, что методы доступа, реализуемые расширениями, могут определять для своих индексов дополнительные имена свойств.

Таблица 1.63 – Свойства столбца индекса

Имя	Описание
asc	Сортируется ли столбец по возрастанию при сканировании вперёд?
desc	Сортируется ли столбец по убыванию при сканировании вперёд?
nulls_first	Выдаются ли NULL в начале при сканировании вперёд?
nulls_last	Выдаются ли NULL в конце при сканировании вперёд?
orderable	Связан ли со столбцом некоторый порядок сортировки?
distance_orderable	Может ли столбец сканироваться по порядку оператором «расстояния», например, ORDER BY столбец <-> константа ?
returnable	Может ли значение столбца быть получено при сканировании только индекса?
search_array	Поддерживает ли столбец внутренними средствами поиск столбец = ANY (массив) ?
search_nulls	Поддерживает ли столбец поиск IS NULL и IS NOT NULL?

Таблица 1.64 – Свойства индекса

Имя	Описание
clusterable	Может ли индекс использоваться в команде CLUSTER?
index_scan	Поддерживает ли индекс простое сканирование (не по битовой карте)?
bitmap_scan	Поддерживает ли индекс сканирование по битовой карте?
backward_scan	Может ли в процессе сканирования меняться направление (для поддержки перемещения курсора FETCH BACKWARD без необходимости материализации)?

Таблица 1.65 – Свойства метода доступа индекса

Имя	Описание
can_order	Поддерживает ли метод доступа ASC, DESC и связанные ключевые слова в CREATE INDEX?
can_unique	Поддерживает ли метод доступа уникальные индексы?
can_multi_col	Поддерживает ли метод доступа индексы по нескольким столбцам?
can_exclude	Поддерживает ли метод доступа ограничения-исключения?
can_include	Поддерживает ли метод доступа предложение INCLUDE команды CREATE INDEX?

`pg_options_to_table` возвращает набор параметров хранилища в виде (имя_параметра/значение_параметра), когда ей передается `pg_class.reloptions` или `pg_attribute.attoptions`.

`pg_tablespace_databases` позволяет изучить содержимое табличного пространства. Она возвращает набор OID баз данных, объекты которых размещены в этом табличном пространстве. Если эта функция возвращает строки, это означает, что табличное пространство не пустое и удалить его нельзя. Какие именно объекты находятся в табличном пространстве, можно узнать, подключаясь к базам данных, OID которых сообщила `pg_tablespace_databases`, и анализируя их каталоги `pg_class`.

`pg_typeof` возвращает OID типа данных для переданного значения. Это может быть полезно для разрешения проблем или динамического создания SQL-запросов. Эта функция объявлена как возвращающая тип `regtype`, который является псевдонимом типа `OID`; это означает, что значение этого типа можно сравнивать как `OID`, но выводится оно как название типа. Например:

```
SELECT pg_typeof(33);
```

```
pg_typeof
-----
integer
(1 row)
```

```
SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);
 typlen
-----
```

(1 row)

Выражение `collation for` возвращает правило сортировки для переданного значения, например:

```
SELECT collation for (description) FROM pg_description
LIMIT 1;
```

```
pg_collation_for
-----
«default»
```

(1 row)

```
SELECT collation for ('foo' COLLATE «de_DE»);
```

```
pg_collation_for
-----
«de_DE»
```

(1 row)

Это значение может быть заключено в кавычки и дополнено схемой. Если для выражения аргумента нет правила сортировки, возвращается значение `NULL`. Если же правила сортировки не применимы для типа аргумента, происходит ошибка.

Функции `to_regclass`, `to_regproc`, `to_regprocedure`, `to_regoper`, `to_regoperator` и `to_regtype` преобразуют имена отношений, функций, операторов и типов в объекты типа `regclass`, `regproc`, `regprocedure`, `regoper`, `regoperator` и `regtype`, соответственно. Перечисленные функции отличаются от явных приведений к этим типам тем, что они не принимают числовые `OID` и возвращают `NULL`, вместо того, чтобы генерировать ошибку, если имя не найдено (или, в случае с `to_regproc` и `to_regoper`, если данному имени соответствуют несколько объектов).

В таблице 1.66 перечислены функции, связанные с идентификацией и адресацией объектов баз данных.

Таблица 1.66 – Функции получения информации и адресации объектов

Имя	Тип результата	Описание
<code>pg_describe_object(classid oid, objid oid, objsubid integer)</code>	<code>text</code>	получает описание объекта базы данных

Окончание таблицы 1.66

Имя	Тип результата	Описание
<code>pg_identify_object(classid oid, objid oid, objsubid integer)</code>	typetext, schematext, nametext, identitytext	получает идентификатор объекта базы данных
<code>pg_identify_object_as_address(classid oid, objid oid, objsubid integer)</code>	typetext, object_namestext[], object_argstext[]	получает внешнее представление адреса объекта базы данных
<code>pg_get_object_address(type text, object_names text[], object_args text[])</code>	classidoid, objidoid, objsubidinteger	получает адрес объекта базы данных из его внешнего представления

`pg_describe_object` возвращает текстовое описание объекта БД, идентифицируемого по OID каталога, OID объекта и ID подобъекта (может быть нулевым). Это описание предназначено для человека и может переводиться, в зависимости от конфигурации сервера. С помощью этой функции, например, можно узнать, что за объект хранится в каталоге `pg_depend`.

`pg_identify_object` возвращает строку, содержащую достаточно информации для однозначной идентификации объекта БД по OID каталога, OID объекта и ID подобъекта (может быть нулевым). Эта информация предназначена для машины и поэтому никогда не переводится. Параметр `type` задает тип объекта БД; `schema` – имя схемы, к которой относится объект (либо NULL для объектов, не относящихся к схемам); `name` – имя объекта, при необходимости в кавычках, которое присутствует только если оно (возможно, вместе со схемой) однозначно идентифицирует объект (в противном случае NULL); `identity` – полный идентификатор объекта, точный формат которого зависит от типа объекта, а каждая его часть дополняется схемой и заключается в кавычки, если требуется.

`pg_identify_object_as_address` возвращает запись, содержащую достаточно информации для однозначной идентификации объекта БД по OID каталога, OID объекта и ID подобъекта. Выдаваемая информация не зависит от текущего сервера, то есть по ней можно идентифицировать одноимённый объект на другом сервере. Поле `type` содержит тип объекта БД, а `object_names` и `object_args` – текстовые массивы, в совокупности формирующие ссылку на объект. Эти три значения можно передать функции `pg_get_object_address`,

чтобы получить внутренний адрес объекта. Данная функция является обратной к `pg_get_object_address`.

`pg_get_object_address` возвращает запись, содержащую достаточно информации для уникальной идентификации объекта БД по его типу и массивам имён и аргументов. В ней возвращаются значения, которые используются в системных каталогах, например `pg_depend`, и могут передаваться в другие системные функции, например `pg_identify_object` или `pg_describe_object`. Поле `classid` содержит OID системного каталога, к которому относится объект; `objid` – OID самого объекта, а `objsubid` – идентификатор подобъекта или 0 в случае его отсутствия. Эта функция является обратной к `pg_identify_object_as_address`.

Функции, перечисленные в таблице 1.67, извлекают комментарии, заданные для объектов с помощью команды `COMMENT`. Если найти комментарий для заданных параметров не удастся, они возвращают `NULL`.

Таблица 1.67 – Функции получения комментариев

Имя	Тип результата	Описание
<code>col_description(table_oid, column_number)</code>	text	получает комментарий для колонки таблицы
<code>obj_description(object_oid, catalog_name)</code>	text	получает комментарий для объекта базы данных
<code>obj_description(object_oid)</code>	text	получает комментарий для объекта базы данных (устаревшая форма)
<code>shobj_description(object_oid, catalog_name)</code>	text	получает комментарий для разделяемого объекта баз данных

`col_description` возвращает комментарий для колонки с заданным номером в таблице с указанным OID. `obj_description` нельзя использовать для колонок таблицы, так колонки не имеют собственных OID.

Функция `obj_description` с двумя параметрами возвращает комментарий для объекта, имеющего заданный OID и находящегося в указанном системном каталоге. Например, `obj_description(123456, 'pg_class')` вернет комментарий для таблицы с OID 123456. Форма `obj_description` с одним параметром принимает только OID. Она является устаревшей, так как значения OID могут повторяться в разных системных каталогах, и поэтому она может возвращать комментарий для другого объекта.

`shobj_description` работает подобно `obj_description`, но она получает комментарии для разделяемых объектов. Некоторые системные каталоги являются глобальными для всех баз данных в кластере и описания объектов в них также хранятся глобально.

Функции, перечисленные в таблице 1.68, выдают информацию о транзакциях сервера в форме во внешнем представлении. В основном эти функции используются, чтобы определить, какие транзакции были зафиксированы между двумя снимками состояния.

Таблица 1.68 – Идентификаторы транзакций и снимков состояния

Имя	Тип результата	Описание
<code>txid_current()</code>	<code>bigint</code>	получает идентификатор текущей транзакции и присваивает новый, если текущая транзакция его не имеет
<code>txid_current_if_assigned()</code>	<code>bigint</code>	работает аналогично <code>txid_current()</code> , но возвращает <code>NULL</code> , не присваивая новый идентификатор транзакции, если он ещё не присвоен
<code>txid_current_snapshot()</code>	<code>txid_snapshot</code>	получает код текущего снимка
<code>txid_snapshot_xip(txid_snapshot)</code>	<code>setof bigint</code>	возвращает идентификаторы выполняющихся транзакций в снимке
<code>txid_snapshot_xmax(txid_snapshot)</code>	<code>bigint</code>	возвращает значение <code>xmax</code> для заданного снимка
<code>txid_snapshot_xmin(txid_snapshot)</code>	<code>bigint</code>	возвращает значение <code>xmin</code> для заданного снимка
<code>txid_visible_in_snapshot(bigint, txid_snapshot)</code>	<code>boolean</code>	видима ли транзакция с указанным идентификатором в данном снимке? (коды подтранзакций не поддерживаются)
<code>txid_status(bigint)</code>	<code>text</code>	возвращает состояние заданной транзакции – <code>committed</code> (зафиксирована), <code>aborted</code> (прервана), <code>in progress</code> (выполняется) или <code>NULL</code> , если идентификатор транзакции слишком старый

Внутренний тип идентификаторов транзакций (`xid`) имеет размер 32 бита, поэтому они повторяются через 4 миллиарда транзакций. Однако эти функции выдают 64-битные значения, дополненные счетчиком «эпохи», так что эти значения останутся уникальными на протяжении всей жизни сервера. Используемый этими функциями тип данных `txid_snapshot` сохраняет информацию о видимости транзакций в определенный момент времени. Его состав описан в таблице 1.69.

Таблица 1.69 – Состав информации о снимке

Имя	Описание
<code>xmin</code>	Идентификатор самой ранней транзакции (<code>txid</code>) из активных. Все предыдущие транзакции либо зафиксированы и видимы, либо отменены и мертвы.

Окончание таблицы 1.69

Имя	Описание
xmax	Первый txid из ещё не назначенных. На момент снимка не было запущенных (а значит и видимых) транзакций с идентификатором, большим или равным данному.
xip_list	Список идентификаторов транзакций, активных в момент снимка. Он включает только идентификаторы с номерами от xmin до xmax; хотя уже могут быть транзакции с идентификаторами больше xmax. Если в этом списке не оказывается идентификатора транзакции xmin ≤ txid < xmax, это означает, что она уже не выполнялась к моменту снимка и, таким образом, видима или мертва, в зависимости от типа завершения. Идентификаторы подтранзакций в этот список не включаются.

В текстовом виде txid_snapshot представляется как xmin:xmax:xip_list. Например, 10:20:10,14,15 означает xmin=10, xmax=20, xip_list=10, 14, 15.

Функция txid_status(bigint) выдаёт состояние фиксации последней транзакции. Проверять его, приложения могут определить, была ли транзакция зафиксирована или прервана, когда приложение отключается от сервера баз данных в процессе выполнения COMMIT. Состояние транзакции может быть следующим: in progress (выполняется), committed (зафиксирована) и aborted (прервана), если только транзакция не настолько стара, чтобы система удалила информация о её состоянии фиксирования. Если же она так стара, что упоминаний о ней в системе не осталось и информация о фиксации потеряна, эта функция возвращает NULL. Заметьте, что для подготовленных транзакций возвращается состояние in progress; приложения должны проверять pg_prepared_xacts, если им нужно определить, является ли транзакция «txid» подготовленной.

Функции, показанные в таблице 1.70, выдают информацию об уже зафиксированных транзакциях. Они возвращают полезные данные, только когда включён параметр конфигурации track_commit_timestamp и только для транзакций, зафиксированных после его включения.

Таблица 1.70 – Информация о фиксации транзакций

Имя	Тип результата	Описание
pg_xact_commit_timestamp(xid)	timestamp with time zone	выдаёт время фиксации транзакции
pg_last_committed_xact()	xid, timestamp with time zone	выдаёт идентификатор и время фиксации транзакции, зафиксированной последней

Функции, перечисленные в таблице 1.71, выводят свойства, установленные

командой `initdb`, например, версию каталога. Они также выводят сведения о работе журнала предзаписи и контрольных точек. Эта информация относится ко всему кластеру, а не к отдельной базе данных. Данные функции выдают практически всю ту же информацию, и из того же источника, что и `pg_controldata`, но в форме, более подходящей для функций SQL.

Таблица 1.71 – Функции управления данными

Имя	Тип результата	Описание
<code>pg_control_checkpoint()</code>	record	Возвращает информацию о состоянии текущей контрольной точки.
<code>pg_control_system()</code>	record	Возвращает информацию о текущем состоянии управляющего файла.
<code>pg_control_init()</code>	record	Возвращает информацию о состоянии инициализации кластера.
<code>pg_control_recovery()</code>	record	Возвращает информацию о состоянии восстановления.

Функция `pg_control_checkpoint` возвращает запись, показанную в таблице 1.72.

Таблица 1.72 – Столбцы результата `pg_control_checkpoint`

Имя столбца	Тип данных
<code>checkpoint_lsn</code>	<code>pg_lsn</code>
<code>redo_lsn</code>	<code>pg_lsn</code>
<code>redo_wal_file</code>	text
<code>timeline_id</code>	integer
<code>prev_timeline_id</code>	integer
<code>full_page_writes</code>	boolean
<code>next_xid</code>	text
<code>next_oid</code>	oid
<code>next_multixact_id</code>	xid
<code>next_multi_offset</code>	xid
<code>oldest_xid</code>	xid
<code>oldest_xid_dbid</code>	oid
<code>oldest_active_xid</code>	xid
<code>oldest_multi_xid</code>	xid
<code>oldest_multi_dbid</code>	oid
<code>oldest_commit_ts_xid</code>	xid
<code>newest_commit_ts_xid</code>	xid
<code>checkpoint_time</code>	timestamp with time zone

Функция `pg_control_system` возвращает запись, показанную в таблице 1.73.

Таблица 1.73 – Столбцы результата `pg_control_system`

Имя столбца	Тип данных
<code>pg_control_version</code>	integer
<code>catalog_version_no</code>	integer
<code>system_identifier</code>	bigint
<code>pg_control_last_modified</code>	timestamp with time zone

Функция `pg_control_init` возвращает запись, показанную в таблице 1.74.

Таблица 1.74 – Столбцы результата pg_control_init

Имя столбца	Тип данных
max data alignment	integer
database block size	integer
blocks per segment	integer
wal block size	integer
bytes per wal segment	integer
max identifier length	integer
max index columns	integer
max toast chunk size	integer
large object chunk size	integer
float4 pass by value	boolean
float8 pass by value	boolean
data page checksum version	integer

Функция pg_control_recovery возвращает запись, показанную в таблице 1.75.

Таблица 1.75 – Столбцы результата pg_control_recovery

Имя столбца	Тип данных
min_recovery_end_lsn	pg_lsn
min_recovery_end_timeline	integer
backup_start_lsn	pg_lsn
backup_end_lsn	pg_lsn
end_of_backup_record_required	boolean

1.27. Функции для системного администрирования

Функции, описанные в этом разделе, предназначены для контроля и управления сервером Синергия-БД.

1.27.1. Функции для управления конфигурацией

В таблице 1.76 показаны функции, позволяющие получить и изменить значения параметров конфигурации выполнения.

Таблица 1.76 – Функции для управления конфигурацией

Имя	Тип результата	Описание
current_setting(setting_name [, missing_ok])	text	получает текущее значение параметра
set_config(setting_name, new_value, is_local)	text	устанавливает новое значение параметра и возвращает его

Функция current_setting выдает текущее значение параметра setting_name. Она соответствует стандартной SQL-команде SHOW. Пример использования:

```
SELECT current_setting('datestyle');
```

```
current_setting
```

```
-----
ISO, MDY
(1 row)
```

Если параметра с именем `setting_name` нет, функция `current_setting` выдаёт ошибку, если только дополнительно не передан параметр `missing_ok` со значением `true`.

`set_config` устанавливает для параметра `setting_name` значение `new_value`. Если параметр `is_local` равен `true`, новое значение будет действовать только в рамках текущей транзакции. Чтобы это значение действовало на протяжении текущего сеанса, ему нужно присвоить `false`. Эта функция соответствует SQL-команде `SET`. Пример использования:

```
SELECT set_config('log_statement_stats', 'off', false);

set_config
-----
off
(1 row)
```

1.27.2. Функции для передачи сигналов серверу

Функции, перечисленные в таблице 1.77, позволяют передавать управляющие сигналы другим серверным процессам. Вызывать эти функции обычно могут только суперпользователи, кроме явно отмеченных исключений.

Таблица 1.77 – Функции для передачи сигналов серверу

Имя	Тип результата	Описание
<code>pg_cancel_backend(pid int)</code>	boolean	Отменяет текущий запрос в обслуживающем процессе. Это действие разрешается и ролям, являющимся членами роли, обслуживающий процесс которой затрагивается, и ролям, которым дано право <code>pg_signal_backend</code> ; однако только суперпользователям разрешено воздействовать на обслуживающие процессы других суперпользователей.
<code>pg_reload_conf()</code>	boolean	Даёт команду серверным процессам перегрузить конфигурацию
<code>pg_rotate_logfile()</code>	boolean	Прокручивает журнал сообщений сервера
<code>pg_terminate_backend(pid int)</code>	boolean	Завершает обслуживающий процесс. Это действие разрешается и ролям, являющимся членами роли, обслуживающий процесс которой прерывается, и ролям, которым дано право <code>pg_signal_backend</code> ; однако только суперпользователям разрешено прерывать обслуживающие процессы других суперпользователей.

Каждая из этих функций возвращает `true` при успешном завершении и

false в противном случае.

pg_cancel_backend и pg_terminate_backend передают сигналы (SIGINT и SIGTERM, соответственно) серверному процессу с заданным кодом PID. Код активного процесса можно получить из колонки pid представления pg_stat_activity или просмотрев на сервере процессы с именем postgres (используя ps в Unix). Роль пользователя активного процесса можно узнать в колонке username представления pg_stat_activity.

pg_reload_conf отправляет сигнал SIGHUP главному серверному процессу, который командует всем подчиненным процессам перезагрузить файлы конфигурации.

pg_rotate_logfile указывает менеджеру журнала сообщений немедленно переключиться на новый файл. Это имеет смысл, только когда работает встроенный сборщик сообщений, так как без него подпроцесс менеджера журнала не запускается.

1.27.3. Функции управления резервным копированием

Функции, перечисленные в таблице 1.78, предназначены для выполнения резервного копирования «на ходу». Эти функции нельзя выполнять во время восстановления (за исключением pg_is_in_backup, pg_backup_start_time и pg_xlog_location_diff).

Таблица 1.78 – Функции управления резервным копированием

Имя	Тип результата	Описание
pg_create_restore_point(name text)	pg_lsn	Создаёт именованную точку для восстановления (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям)
pg_current_wal_flush_lsn()	pg_lsn	Получает текущую позицию сброса данных в журнале предзаписи
pg_current_wal_insert_lsn()	pg_lsn	Получает текущую позицию добавления в журнале предзаписи
pg_current_wal_lsn()	pg_lsn	Получает текущую позицию записи в журнале предзаписи
pg_start_backup(label text [, fast boolean [, exclusive boolean]])	pg_lsn	Подготавливает сервер к резервному копированию (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям)

Окончание таблицы 1.78

Имя	Тип результата	Описание
<code>pg_stop_backup()</code>	<code>pg_lsn</code>	Завершает монопольное резервное копирование (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям)
<code>pg_stop_backup(exclusive boolean [, wait_for_archive boolean])</code>	<code>setof record</code>	Завершает монопольное или немонопольное резервное копирование (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям)
<code>pg_is_in_backup()</code>	<code>bool</code>	Возвращает true в процессе исключительного резервного копирования
<code>pg_backup_start_time()</code>	<code>timestamp with time zone</code>	Получает время запуска выполняющегося исключительного резервного копирования
<code>pg_switch_wal()</code>	<code>pg_lsn</code>	Иницирует переключение на новый файл журнала предзаписи (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям)
<code>pg_walfile_name(lsn pg_lsn)</code>	<code>text</code>	Получает из позиции в журнале предзаписи имя соответствующего файла
<code>pg_walfile_name_offset(lsn pg_lsn)</code>	<code>text, integer</code>	Получает из позиции в журнале предзаписи имя соответствующего файла и десятичное смещение в нём
<code>pg_wal_lsn_diff(lsn pg_lsn, lsn pg_lsn)</code>	<code>numeric</code>	Вычисляет разницу между двумя позициями в журнале предзаписи

`pg_start_backup` принимает произвольную заданную пользователем метку резервной копии. Обычно это имя файла, в котором будет сохранена резервная копия. При копировании в монопольном режиме эта функция записывает файл метки (`backup_label`) и, если есть ссылки в каталоге `pg_tblspc/`, файл карты табличных пространств (`tablespace_map`) в каталог данных кластера БД, выполняет процедуру контрольной точки, а затем возвращает в текстовом виде начальную позицию в журнале предзаписи для данной резервной копии. Результат этой функции может быть полезен, но если он не нужен, его можно просто игнорировать. При копировании в немонопольном режиме содержимое этих файлов выдаётся функцией `pg_stop_backup` и должно быть записано в архивную копию вызывающим субъектом.

```
postgres=# select pg_start_backup('label_goes_here');
pg_start_backup
-----
0/D4445B8
(1 row)
```

У этой функции есть также второй необязательный, параметр типа `boolean`. Если он равен `true`, `pg_start_backup` начнет работу максимально быстро. При

этом будет немедленно выполнена процедура контрольной точки, что может повлечь массу операций ввода/вывода и затормозить параллельные запросы.

При монопольном копировании функция `pg_stop_backup` удаляет файл метки (и, если существует, файл `tablespace_map`), созданный функцией `pg_start_backup`. При немонопольном копировании содержимое `backup_label` и `tablespace_map` возвращается в результате функции и должно быть записано в файлы в архиве (а не в каталоге данных). Также есть необязательный второй параметр типа `boolean`. Если он равен `false`, `pg_stop_backup` завершится сразу после окончания резервного копирования, не дожидаясь архивации WAL. Это поведение полезно только для программ резервного копирования, которые независимо осуществляют архивацию WAL. Если же WAL не заархивируется, резервная копия может оказаться неполной, а значит, бесполезной. Когда он равен `true`, `pg_stop_backup` будет ждать выполнения архивации WAL, если архивирование включено; для резервного сервера это означает, что ожидание возможно только при условии `archive_mode = always`. Если активность записи на ведущем сервере невысока, может иметь смысл выполнить на нём `pg_switch_wal` для немедленного переключения сегмента.

При выполнении на ведущем эта функция также создаёт файл истории резервного копирования в области архива журнала предзаписи. В этом файле для данной резервной копии сохраняется метка, заданная при вызове `pg_start_backup`, начальная и конечная позиция в журнале предзаписи, а также время начала и окончания. Возвращает данная функция позицию окончания резервной копии в журнале предзаписи (которую тоже можно игнорировать). После записи конечной позиции текущая позиция автоматически перемещается к следующему файлу журнала предзаписи, чтобы файл конечной позиции можно было немедленно архивировать для завершения резервного копирования.

`pg_switch_wal` производит переключение на следующий файл журнала предзаписи, что позволяет архивировать текущий (в предположении, что архивация выполняется непрерывно). Эта функция возвращает конечную позицию в журнале предзаписи (в только что законченном файле журнала) + 1. Если с момента

последнего переключения файлов не было активности, отражающейся в журнале предзаписи, `pg_switch_wal` не делает ничего и возвращает начальную позицию в файле журнала предзаписи, используемом в данный момент.

`pg_create_restore_point` создаёт именованную запись в журнале предзаписи, которую можно использовать как цель при восстановлении, и возвращает соответствующую позицию в журнале предзаписи. Затем полученное имя можно присвоить параметру `recovery_target_name`, указав тем самым точку, до которой будет выполняться восстановление. Учтите, что если вы создадите несколько точек восстановления с одним именем, восстановление будет остановлено на первой точке с этим именем.

`pg_current_wal_lsn` выводит текущую позицию записи в журнале предзаписи в том же формате, что и вышеописанные функции. `pg_current_wal_insert_lsn` подобным образом выводит текущую позицию добавления, а `pg_current_wal_flush_lsn` – позицию сброса данных журнала. Позицией добавления называется «логический» конец журнала предзаписи в любой момент времени, тогда как позиция записи указывает на конец данных, фактически вынесённых из внутренних буферов сервера, а позиция сброса показывает, до какого места данные гарантированно сохранены в надёжном хранилище. Позиция записи отмечает конец данных, которые может видеть снаружи внешний процесс, и именно она представляет интерес при копировании частично заполненных файлов журнала. Позиция добавления и позиция сброса выводятся в основном для отладки серверной части. Все эти операции выполняются в режиме «только чтение» и не требуют прав суперпользователя.

Из результатов всех описанных выше функций можно получить имя файла журнала предзаписи и смещение в нём, используя функцию `pg_walfile_name_offset`. Например:

```
postgres=# SELECT * FROM
pg_walfile_name_offset(pg_stop_backup());
      file_name      | file_offset
-----+-----
000000010000000000000000D |      4039624
```


(1 row)

Подобным образом, `pg_walfile_name` извлекает только имя файла журнала предзаписи. Когда заданная позиция в журнале предзаписи находится ровно на границе файлов, обе эти функции возвращают имя предыдущего файла. Обычно это поведение предпочтительно при архивировании журнала предзаписи, так как именно предыдущий файл является последним подлежащим архивации.

`pg_wal_lsn_diff` вычисляет разницу в байтах между двумя позициями в журнале предзаписи. Полученный результат можно использовать с `pg_stat_replication` или другими функциями.

1.27.4. Функции управления восстановлением

Функции, приведенные в таблице 1.79, предоставляют сведения о текущем состоянии резервного сервера. Эти функции могут выполняться, как во время восстановления, так и в обычном режиме работы.

Таблица 1.79 – Функции для получения информации о восстановлении

Имя	Тип результата	Описание
<code>pg_is_in_recovery()</code>	<code>bool</code>	Возвращает true в процессе восстановления.
<code>pg_last_wal_receive_lsn()</code>	<code>pg_lsn</code>	Получает позицию последней записи журнала предзаписи, полученной и записанной на диск в процессе потоковой репликации. Пока выполняется потоковая репликация, эта позиция постоянно увеличивается. По окончании восстановления она останавливается на записи WAL, полученной и записанной на диск последней. Если потоковая репликация отключена или ещё не запускалась, функция возвращает NULL.
<code>pg_last_wal_replay_lsn()</code>	<code>pg_lsn</code>	Получает позицию последней записи журнала предзаписи, воспроизведённой при восстановлении. В процессе восстановления эта позиция постоянно увеличивается. По окончании восстановления она останавливается на записи WAL, которая была восстановлена последней. Если сервер был запущен не в режиме восстановления, эта функция возвращает NULL.
<code>pg_last_xact_replay_timestamp()</code>	<code>timestamp with time zone</code>	Получает отметку времени последней транзакции, воспроизведённой при восстановлении. Это время, когда на главном сервере произошла фиксация или откат записи WAL для этой транзакции. Если в процессе восстановления не была воспроизведена ни одна транзакция, эта функция возвращает NULL. В противном случае это значение постоянно увеличивается в процессе восстановления. По окончании восстановления оно останавливается на транзакции, которая была восстановлена последней. Если сервер был запущен не в режиме восстановления, эта функция возвращает NULL.

Функции, перечисленные в таблице 1.80, управляют процессом восстановления. Вызывать их в другое время нельзя.

Таблица 1.80 – Функции управления восстановлением

Имя	Тип результата	Описание
<code>pg_is_wal_replay_paused()</code>	<code>bool</code>	Возвращает true, если восстановление приостановлено.
<code>pg_wal_replay_pause()</code>	<code>void</code>	Немедленно приостанавливает восстановление (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям).
<code>pg_wal_replay_resume()</code>	<code>void</code>	Перезапускает восстановление, если оно было приостановлено (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям).

Когда восстановление приостановлено, запись изменений в базу не производится. Если она находится в «горячем резерве», все последующие запросы будут видеть один согласованный снимок базы данных и до продолжения восстановления конфликты запросов исключаются.

Когда потоковая репликация выключена, пауза при восстановлении может длиться сколь угодно долго без каких-либо проблем. Если же запущена потоковая репликация, новые записи WAL продолжают поступать и заполняют весь диск рано или поздно, в зависимости от длительности паузы, интенсивности записи в WAL и объема свободного пространства.

1.27.5. Функции синхронизации снимков

Синергия-БД позволяет синхронизировать снимки состояния между сеансами баз данных. Снимок состояния определяет, какие данные видны транзакции, работающей с этим снимком. Синхронизация снимков необходима, когда в двух или более сеансах нужно видеть одно и то же содержимое базы данных. Если в двух сеансах транзакции запускаются независимо, всегда есть вероятность, что некая третья транзакция будет зафиксирована между командами START TRANSACTION для первых двух, и в результате в одном сеансе будет виден результат третьей, а в другом – нет.

Для решения этой проблемы Синергия-БД позволяет транзакции экспортировать снимок состояния, с которым она работает. Пока экспортирующая этот снимок транзакция выполняется, другие транзакции могут импортировать его и, таким образом, увидеть абсолютно то же состояние базы данных, что видит первая транзакция. Но учтите, что любые изменения, произведенные этими транзакциями,

будут не видны для других, как это и должно быть с изменениями в незафиксированных транзакциях. Таким образом, транзакции синхронизируют только начальное состояние данных, а последующие производимые в них изменения изолируются как обычно.

Снимки состояния экспортируются с помощью функции `pg_export_snapshot`, показанной в таблице 1.81, и импортируются командой `SET TRANSACTION`.

Таблица 1.81 – Функции синхронизации снимков

Имя	Тип результата	Описание
<code>pg_export_snapshot()</code>	<code>text</code>	Сохраняет снимок текущего состояния и возвращает его идентификатор

Функция `pg_export_snapshot` создает снимок текущего состояния и возвращает его идентификатор в строке типа `text`. Данная строка должна передаваться (за рамками базы данных) клиентам, которые будут импортировать этот снимок. При этом импортировать его нужно раньше, чем завершится транзакция, которая его экспортировала. Если необходимо, транзакция может экспортировать несколько снимков. Заметьте, что это имеет смысл только для транзакций уровня `READ COMMITTED`, так как транзакции `REPEATABLE READ` и более высоких уровней изоляции работают с одним снимком состояния. После того, как транзакция экспортировала снимок, её нельзя подготовить с помощью `PREPARE TRANSACTION`.

Подробнее использование экспортированных снимков рассматривается в описании `SET TRANSACTION`.

1.27.6. Функции репликации

В таблице 1.82 перечислены функции, предназначенные для управления и взаимодействия с механизмом репликации. Использовать эти функции разрешено только суперпользователям.

Функции, описанные в подразделе 1.27.5, подразделе 1.27.4 и подразделе 1.27.3 также имеют отношение к репликации.

Таблица 1.82 – Функции репликации SQL

Функция	Тип результата	Описание
<code>pg_create_physical_replication_slot(slot_name name [, immediately_reserve boolean, temporary boolean])</code>	<code>(slot_name name, lsn pg_lsn)</code>	Создаёт новый физический слот репликации с именем <code>slot_name</code> . Необязательный второй параметр, когда он равен <code>true</code> , указывает, что LSN для этого слота репликации должен быть зарезервирован немедленно; в противном случае LSN резервируется при первом подключении клиента потоковой репликации. Передача изменений из физического слота возможна только по протоколу потоковой репликации. Необязательный третий параметр, <code>temporary</code> , когда он равен <code>true</code> , указывает, что этот слот не должен постоянно храниться на диске, так как он предназначен только для текущего сеанса. Временные слоты также освобождаются при любой ошибке. Эта функция соответствует команде протокола репликации <code>CREATE REPLICATION SLOT ... PHYSICAL</code> .
<code>pg_drop_replication_slot(slot_name name)</code>	<code>void</code>	Удаляет физический или логический слот репликации с именем <code>slot_name</code> . Соответствует команде протокола репликации <code>DROP_REPLICATION_SLOT</code> . Для логических слотов эта функция должна вызываться через подключение к той же базе данных, в которой был создан слот.
<code>pg_create_logical_replication_slot(slot_name name, plugin name [, temporary boolean])</code>	<code>(slot_name name, lsn pg_lsn)</code>	Создаёт новый логический (декодирующий) слот репликации с именем <code>slot_name</code> , используя модуль вывода <code>plugin</code> . Необязательный третий параметр, <code>temporary</code> , когда равен <code>true</code> , указывает, что этот слот не должен постоянно храниться на диске, так как он предназначен только для текущего сеанса. Временные слоты также освобождаются при любой ошибке. Вызов этой функции равнозначен выполнению команды протокола репликации <code>CREATE REPLICATION SLOT ... LOGICAL</code> .
<code>pg_logical_slot_get_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])</code>	<code>(lsn pg_lsn, xid xid, data text)</code>	Возвращает изменения в слоте <code>slot_name</code> с позиции, до которой ранее были получены изменения. Если параметры <code>upto_lsn</code> и <code>upto_nchanges</code> равны <code>NULL</code> , логическое декодирование продолжится до конца журнала транзакций. Если <code>upto_lsn</code> не <code>NULL</code> , декодироваться будут только транзакции, зафиксированные до заданного LSN. Если <code>upto_nchanges</code> не <code>NULL</code> , декодирование остановится, когда число строк, полученных при декодировании, превысит заданное значение. Обратите внимание, однако, что фактическое число возвращённых строк может быть больше, так как это ограничение проверяется только после добавления строк, декодированных для очередной транзакции.
<code>pg_logical_slot_peek_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])</code>	<code>(lsn pg_lsn, xid xid, data text)</code>	Работает так же, как функция <code>pg_logical_slot_get_changes()</code> , но не забирает изменения; то есть, они будут получены снова при следующих вызовах.
<code>pg_logical_slot_get_binary_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])</code>	<code>(lsn pg_lsn, xid xid, data bytea)</code>	Работает так же, как функция <code>pg_logical_slot_get_changes()</code> , но выдаёт изменения в типе <code>bytea</code> .

Продолжение таблицы 1.82

Функция	Тип результата	Описание
<code>pg_logical_slot_peek_binary_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])</code>	<code>(lsn pg_lsn, xid xid, data bytea)</code>	Работает так же, как функция <code>pg_logical_slot_get_changes()</code> , но выдаёт изменения в типе <code>bytea</code> и не забирает их; то есть, они будут получены снова при следующих вызовах.
<code>pg_replication_slot_advance(slot_name name, upto_lsn pg_lsn)</code>	<code>(slot_name name, end_lsn pg_lsn) bool</code>	Продвигает текущую подтверждённую позицию слота репликации с именем <code>slot_name</code> . Слот нельзя переместить назад, а также вперёд за текущую позицию добавления. Возвращает имя слота и позицию, до которой он продвинулся.
<code>pg_replication_origin_create(node_name text)</code>	<code>oid</code>	Создаёт источник репликации с заданным внешним именем и возвращает назначенный ему внутренний идентификатор.
<code>pg_replication_origin_drop(node_name text)</code>	<code>void</code>	Удаляет ранее созданный источник репликации, в том числе связанную информацию о воспроизведении.
<code>pg_replication_origin_oid(node_name text)</code>	<code>oid</code>	Ищет источник репликации по имени и возвращает внутренний идентификатор. Если такой источник не находится, выдаётся ошибка.
<code>pg_replication_origin_session_setup(node_name text)</code>	<code>void</code>	Помечает текущий сеанс, как воспроизводящий журнал из указанного источника, что позволяет отслеживать положение воспроизведения. Чтобы отменить это действие, вызовите <code>pg_replication_origin_session_reset</code> . Может использоваться, только если не был настроен предыдущий источник.
<code>pg_replication_origin_session_reset()</code>	<code>void</code>	Отменяет действие <code>pg_replication_origin_session_setup()</code> .
<code>pg_replication_origin_session_is_setup()</code>	<code>bool</code>	В текущем сеансе настроен источник репликации?
<code>pg_replication_origin_session_progress(flush bool)</code>	<code>pg_lsn</code>	Возвращает позицию воспроизведения для источника репликации, настроенного в текущем сеансе. Параметр <code>flush</code> определяет, будет ли гарантироваться сохранение локальной транзакции на диске.
<code>pg_replication_origin_xact_setup(origin_lsn pg_lsn, origin_timestamp timestamp_tz)</code>	<code>void</code>	Помечает текущую транзакцию как воспроизводящую транзакцию, зафиксированную с указанным LSN и временем. Может вызываться только после того, как был настроен источник репликации в результате вызова <code>pg_replication_origin_session_setup()</code> .
<code>pg_replication_origin_xact_reset()</code>	<code>void</code>	Отменяет действие <code>pg_replication_origin_xact_setup()</code> .
<code>pg_replication_origin_advance(node_name text, lsn pg_lsn)</code>	<code>void</code>	Устанавливает положение репликации для заданного узла в указанную позицию. Это в основном полезно для установки начальной позиции или новой позиции после изменения конфигурации и подобных действий. Но учтите, что неосторожное использование этой функции может привести к несогласованности реплицированных данных.
<code>pg_replication_origin_progress(node_name text, flush bool)</code>	<code>pg_lsn</code>	Возвращает позицию воспроизведения для заданного источника репликации. Параметр <code>flush</code> определяет, будет ли гарантироваться сохранение локальной транзакции на диске.

Окончание таблицы 1.82

Функция	Тип результата	Описание
<code>pg_logical_emit_message(transactional bool, prefix text, content text)</code>	<code>pg_lsn</code>	Выдаёт текстовое сообщение логического декодирования. Её можно использовать для передачи произвольных сообщений через WAL модулям логического декодирования. Параметр <code>transactional</code> устанавливает, должно ли сообщение быть частью текущей транзакции или оно должно записываться немедленно и декодироваться сразу, как только эта запись будет прочитана при логическом декодировании. Параметр <code>prefix</code> задаёт текстовый префикс, по которому модули логического декодирования могут легко распознать интересующие их сообщения. В параметре <code>content</code> передаётся текст сообщения.
<code>pg_logical_emit_message(transactional bool, prefix text, content bytea)</code>	<code>pg_lsn</code>	Выдаёт двоичное сообщение логического декодирования. Её можно использовать для передачи произвольных сообщений через WAL модулям логического декодирования. Параметр <code>transactional</code> устанавливает, должно ли сообщение быть частью текущей транзакции или оно должно записываться немедленно и декодироваться сразу, как только эта запись будет прочитана при логическом декодировании. Параметр <code>prefix</code> задаёт текстовый префикс, по которому модули логического декодирования могут легко распознать интересующие их сообщения. В параметре <code>content</code> передаётся двоичное содержание сообщения.
<code>pg_create_physical_replication_slot(slot_name name [, immediately_reserve boolean, temporary boolean])</code>	<code>(slot_name name, lsn pg_lsn)</code>	Создаёт новый физический слот репликации с именем <code>slot_name</code> . Необязательный второй параметр, когда он равен <code>true</code> , указывает, что LSN для этого слота репликации должен быть зарезервирован немедленно; в противном случае LSN резервируется при первом подключении клиента потоковой репликации. Передача изменений из физического слота возможна только по протоколу потоковой репликации. Необязательный третий параметр, <code>temporary</code> , когда он равен <code>true</code> , указывает, что этот слот не должен постоянно храниться на диске, так как он предназначен только для текущего сеанса. Временные слоты также освобождаются при любой ошибке. Эта функция соответствует команде протокола репликации <code>CREATE REPLICATION SLOT ... PHYSICAL</code> .

1.27.7. Функции управления объектами баз данных

Функции, перечисленные в таблице 1.83, вычисляют объем, который занимают на диске различные объекты баз данных.

Таблица 1.83 – Функции получения размера объектов БД

Имя	Тип результата	Описание
<code>pg_column_size(any)</code>	<code>int</code>	Число байт, необходимых для хранения заданного значения (возможно, в сжатом виде)
<code>pg_database_size(oid)</code>	<code>bigint</code>	Объем, который занимает на диске база данных с заданным OID
<code>pg_database_size(name)</code>	<code>bigint</code>	Объем, который занимает на диске база данных с заданным именем
<code>pg_indexes_size(regclass)</code>	<code>bigint</code>	Общий объем индексов, связанных с указанной таблицей
<code>pg_relation_size(relation regclass, fork text)</code>	<code>bigint</code>	Объем, который занимает на диске указанный слой ('main', 'fsm', 'vm' или 'init') заданной таблицы или индекса

Окончание таблицы 1.83

Имя	Тип результата	Описание
<code>pg_relation_size(relation regclass)</code>	<code>bigint</code>	Краткая форма <code>pg_relation_size(..., 'main')</code>
<code>pg_size_bytes(text)</code>	<code>bigint</code>	Преобразует размер в понятном человеку формате с единицами измерения в число байт
<code>pg_size_pretty(bigint)</code>	<code>text</code>	Преобразует размер в байтах, представленный в 64-битном целом, в понятный человеку формат с единицами измерения
<code>pg_size_pretty(числовой тип)</code>	<code>text</code>	Преобразует размер в байтах, представленный в значении числового типа, в понятный человеку формат с единицами измерения
<code>pg_table_size(regclass)</code>	<code>bigint</code>	Объем, который занимает на диске данная таблица, за исключением индексов (но включая TOAST, карту свободного места и карту видимости)
<code>pg_tablespace_size(oid)</code>	<code>bigint</code>	Объем, который занимает на диске табличное пространство с указанным OID
<code>pg_tablespace_size(name)</code>	<code>bigint</code>	Объем, который занимает на диске табличное пространство с заданным именем
<code>pg_total_relation_size(regclass)</code>	<code>bigint</code>	Общий объем, который занимает на диске заданная таблица, включая все индексы и данные TOAST

`pg_column_size` показывает, какой объем требуется для хранения данного значения.

`pg_total_relation_size` принимает OID или имя таблицы или данных TOAST и возвращает общий объем, который занимает на диске эта таблица, включая все связанные с ней индексы. Результат этой функции равняется `pg_table_size + pg_indexes_size`.

`pg_table_size` принимает OID или имя таблицы и возвращает объем, который занимает на диске эта таблица без индексов. При этом учитывается размер TOAST, карты свободного места и карты видимости.

`pg_indexes_size` принимает OID или имя таблицы и возвращает общий объем, который занимают все индексы таблицы.

`pg_database_size` и `pg_tablespace_size` принимают OID или имя базы данных либо табличного пространства и возвращают общий объем, который они занимают на диске. Для использования `pg_database_size` требуется иметь право CONNECT для указанной базы данных (оно предоставляется по умолчанию) или быть членом роли `pg_read_all_stats`. Для использования `pg_tablespace_size` необходимо иметь право CREATE для указанного табличного пространства или быть членом роли `pg_read_all_stats`, если

только это не табличное пространство по умолчанию для текущей базы данных.

`pg_relation_size` принимает OID или имя таблицы, индекса или TOAST-таблицы и возвращает размер одного слоя этого отношения (в байтах). Заметьте, что в большинстве случаев удобнее использовать более высокоуровневые функции `pg_total_relation_size` и `pg_table_size`, которые суммируют размер всех слоев. С одним аргументом она возвращает размер основного слоя для данных заданного отношения. Название другого интересующего слоя можно передать во втором аргументе:

- 'main' возвращает размер основного слоя данных заданного отношения;
- 'fsm' возвращает размер карты свободного места, связанной с заданным отношением;
- 'vm' возвращает размер карты видимости, связанной с заданным отношением;
- 'init' возвращает размер слоя инициализации для заданного отношения, если он имеется.

`pg_size_pretty` можно использовать для форматирования результатов других функций в виде, более понятном человеку, с единицами измерения bytes, kB, MB, GB и TB.

`pg_size_bytes` можно использовать для получения размера в байтах из строки в формате, понятном человеку. Входная строка может содержать единицы bytes, kB, MB, GB и TB, и разбирается без учёта регистра. Если единицы не указываются, подразумеваются байты.

Примечание. Единицы kB, MB, GB и TB, фигурирующие в функциях `pg_size_pretty` и `pg_size_bytes`, определяются как степени 2, а не 10, так что 1kB равен 1024 байтам, 1MB равен $1024^2 = 1048576$ байтов и т. д.

Вышеописанные функции, работающие с таблицами или индексами, принимают аргумент типа `regclass`, который представляет собой просто OID таблицы или индекса в системном каталоге `pg_class`. Однако вам не нужно вручную вычислять OID, так как процедура ввода значения `regclass` может сделать это за вас. Для этого достаточно записать имя таблицы в апострофах, как обычную текстовую константу. В соответствии с правилами обработки обычных

имен SQL, если имя таблицы не заключено в кавычки, эта строка будет переведена в нижний регистр.

Если переданному значению OID не соответствуют существующий объект, эти функции возвращают NULL.

Функции, перечисленные в таблице 1.84, помогают определить, в каких файлах на диске хранятся объекты базы данных.

Таблица 1.84 – Функции определения расположения объектов

Имя	Тип результата	Описание
<code>pg_relation_filenode(relation regclass)</code>	oid	Номер файлового узла для указанного отношения
<code>pg_relation_filepath(relation regclass)</code>	text	Путь к файлу, в котором хранится указанное отношение
<code>pg_filenode_relation(tablespace oid, filenode oid)</code>	regclass	Находит отношение, связанное с данным табличным пространством и файловым узлом

`pg_relation_filenode` принимает OID или имя таблицы, индекса, последовательности или таблицы TOAST и возвращает номер «файлового узла», связанным с этим объектом. Файловым узлом называется основной компонент имени файла, используемого для хранения данных. Для большинства таблиц этот номер совпадает со значением `pg_class.relfilenode`, но для некоторых системных каталогов `relfilenode` равен 0, и нужно использовать эту функцию, чтобы узнать действительное значение. Если указанное отношение не хранится на диске, как например представление, данная функция возвращает NULL.

`pg_relation_filepath` подобна `pg_relation_filenode`, но возвращает полный путь к файлу (относительно каталога данных PGDATA) отношения.

Функция `pg_filenode_relation` является обратной к `pg_relation_filenode`. Она возвращает OID отношения по заданному OID «табличного пространства» и «файловому узлу». Для таблицы в табличном пространстве по умолчанию в первом параметре можно передать 0.

В таблице 1.85 перечислены функции, предназначенные для управления правилами сортировки.

Таблица 1.85 – Функции управления правилами сортировки

Имя	Тип результата	Описание
<code>pg_collation_actual_version(id)</code>	text	выдаёт действующую версию правила сортировки из операционной системы
<code>pg_import_system_collations(schema regnamespace)</code>	integer	импортирует правила сортировки из операционной системы

Функция `pg_collation_actual_version` возвращает действующую версию объекта правила сортировки, которая в настоящее время установлена в операционной системе. Если она отличается от значения в `pg_collation.collversion`, может потребоваться перестроить объекты, зависящие от данного правила сортировки.

Функция `pg_import_system_collations` добавляет правила сортировки в системный каталог `pg_collation`, анализируя все локали, которые она находит в операционной системе. Эту информацию использует `initdb`. Если позднее в систему устанавливаются дополнительные локали, эту функцию можно запустить снова, чтобы добавились правила сортировки для новых локалей. Локали, для которых обнаруживаются существующие записи в `pg_collation`, будут пропущены. Объекты правил сортировки для локалей, которые перестают существовать в операционной системе, никогда не удаляются этой функцией. В параметре `schema` обычно передаётся `pg_catalog`, но это не обязательно; правила сортировки могут быть установлены и в другую схему. Эта функция возвращает число созданных ей объектов правил сортировки.

1.27.8. Функции обслуживания индексов

В таблице 1.86 перечислены функции, предназначенные для обслуживания индексов. Эти функции нельзя вызывать в процессе восстановления. Использовать эти функции разрешено только суперпользователям и владельцу определённого индекса.

Таблица 1.86 – Функции обслуживания индексов

Имя	Тип результата	Описание
<code>brin_summarize_new_values(index regclass)</code>	integer	обобщает ещё не обобщённые зоны страниц

Окончание таблицы 1.86

Имя	Тип результата	Описание
<code>brin_summarize_range(index regclass, blockNumber bigint)</code>	integer	обобщает зону страниц, охватывающую данный блок (если она ещё не обобщена)
<code>brin_desummarize_range(index regclass, blockNumber bigint)</code>	integer	сброс обобщения зоны страниц, охватывающей данный блок (если она была обобщена)
<code>gin_clean_pending_list(index regclass)</code>	bigint	перемещает элементы из списка записей GIN, ожидающих обработки, в основную структуру индекса

Функция `brin_summarize_new_values` принимает OID или имя индекса BRIN и просматривает индекс в поисках зон страниц в базовой таблице, ещё не обобщённых в индексе; для каждой такой зоны в результате сканирования страниц таблицы создаётся новый обобщающий кортеж в индексе. Возвращает эта функция число вставленных в индекс обобщающих записей о зонах страниц. Функция `brin_summarize_range` делает то же самое, но затрагивает только зону, охватывающую блок с заданным номером.

Функция `gin_clean_pending_list` принимает OID или имя индекса GIN и очищает очередь указанного индекса, массово перемещая элементы из неё в основную структуру данных GIN. Возвращает она число страниц, убранных из очереди. Заметьте, что если для обработки ей передаётся индекс GIN, построенный с отключённым параметром `fastupdate`, очистка не производится и возвращается значение 0, так как у такого индекса нет очереди записей.

1.27.9. Функции для работы с обычными файлами

Функции, перечисленные в таблице 1.87, предоставляют прямой доступ к файлам, находящимся на сервере. Обычным пользователям, не включённым в роль `pg_read_server_files`, они позволяют обращаться только к файлам в каталоге кластера баз данных и в каталоге `log_directory`. Для файлов в каталоге кластера этим функциям передаётся относительный путь, а для файлов журнала – путь, соответствующий значению параметра `log_directory`.

Заметьте, что пользователи, обладающие правом `EXECUTE` для `pg_read_file()` или связанных функций, имеют возможность читать любой файл на сервере, который может прочитать серверный процесс, и это чтение не ограничивается никакими проверками уровня базы данных. В частности это

означает, что пользователь с такими правами может прочитать содержимое таблицы `pg_authid`, в которой хранятся данные аутентификации, равно как и любой другой файл в базе данных. Таким образом, разрешать доступ к этим функциям следует с большой осторожностью.

Таблица 1.87 – Функции для работы с обычными файлами

Имя	Тип результата	Описание
<code>pg_ls_dir(dirname text [, missing_ok boolean, include_dot_dirs boolean])</code>	setof text	Выводит содержимое каталога (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям).
<code>pg_ls_logdir()</code>	setof record	Выводит имя, размер и время последнего изменения файлов в каталоге журнала сообщений. Доступ к этой функции имеют члены роли <code>pg_monitor</code> , также его можно разрешить и другим ролям обычных пользователей.
<code>pg_ls_waldir()</code>	setof record	Выводит имя, размер и время последнего изменения файлов в каталоге WAL. Доступ к этой функции имеют члены роли <code>pg_monitor</code> , также его можно разрешить и другим ролям обычных пользователей.
<code>pg_read_file(filename text [, offset bigint, length bigint [, missing_ok boolean]])</code>	text	Возвращает содержимое текстового файла (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям).
<code>pg_read_binary_file(filename text [, offset bigint, length bigint [, missing_ok boolean]])</code>	bytea	Возвращает содержимое файла (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям).
<code>pg_stat_file(filename text [, missing_ok boolean])</code>	record	Возвращает информацию о файле (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям).

Некоторые из этих функций принимают необязательный параметр `missing_ok`, который определяет их поведение в случае отсутствия файла или каталога. Если он равен `true`, функция возвращает NULL (за исключением `pg_ls_dir`, которая возвращает пустое множество). Если он равен `false`, возникает ошибка. Значение по умолчанию – `false`.

`pg_ls_dir` возвращает имена всех файлов (а также каталогов и других специальных файлов) в заданном каталоге. Параметр `include_dot_dirs` определяет, будут ли в результирующий набор включаться каталоги «.» и «..». По умолчанию они не включаются (`false`), но их можно включить, чтобы с параметром `missing_ok` равным `true`, пустой каталог можно было отличить от несуществующего.

`pg_ls_logdir` возвращает имя, размер и время последнего изменения

(mtime) всех файлов в каталоге журналов сообщений. По умолчанию использовать её разрешено только суперпользователям и членам роли `pg_monitor`. Другим пользователям доступ к ней можно дать, используя `GRANT`.

`pg_ls_waldir` возвращает имя, размер и время последнего изменения (mtime) всех файлов в каталоге журнала предзаписи (WAL). По умолчанию использовать её разрешено только суперпользователям и членам роли `pg_monitor`. Другим пользователям доступ к ней можно дать, используя `GRANT`.

`pg_read_file` возвращает фрагмент текстового файла с заданного смещения (`offset`), размером не больше `length` байт (размер может быть меньше, если файл кончится раньше). Если смещение `offset` отрицательно, оно отсчитывается от конца файла. Если параметры `offset` и `length` опущены, возвращается все содержимое файла. Прочитанные из файла байты обрабатываются как символы в серверной кодировке; если они оказываются недопустимыми для этой кодировки, возникает ошибка.

`pg_read_binary_file` подобна `pg_read_file`, но её результат имеет тип `bytea`; как следствие, никакие проверки кодировки не выполняются. В сочетании с `convert_from` эту функцию можно применять для чтения файлов в произвольной кодировке:

```
SELECT
convert_from(pg_read_binary_file('file_in_utf8.txt'),
'UTF8');
```

`pg_stat_file` возвращает запись, содержащую размер файла, время последнего обращения и последнего изменения, а также время последнего изменения состояния (только в Unix-системах) и признак типа `boolean`, показывающий, что это каталог. Примеры использования:

```
SELECT * FROM pg_stat_file('filename');
SELECT (pg_stat_file('filename')).modification;
```

1.27.10. Функции управления рекомендательными блокировками

Функции, перечисленные в таблице 1.88, предназначены для управления рекомендательными блокировками.

Таблица 1.88 – Функции управления рекомендательными блокировками

Имя	Тип результата	Описание
<code>pg_advisory_lock(key bigint)</code>	<code>void</code>	Получает исключительную блокировку на уровне сеанса
<code>pg_advisory_lock(key1 int, key2 int)</code>	<code>void</code>	Получает исключительную блокировку на уровне сеанса
<code>pg_advisory_lock_shared(key bigint)</code>	<code>void</code>	Получает разделяемую блокировку на уровне сеанса
<code>pg_advisory_lock_shared(key1 int, key2 int)</code>	<code>void</code>	Получает разделяемую блокировку на уровне сеанса
<code>pg_advisory_unlock(key bigint)</code>	<code>boolean</code>	Освобождает исключительную блокировку на уровне сеанса
<code>pg_advisory_unlock(key1 int, key2 int)</code>	<code>boolean</code>	Освобождает исключительную блокировку на уровне сеанса
<code>pg_advisory_unlock_all()</code>	<code>void</code>	Освобождает все блокировки на уровне сеанса, удерживаемые в данном сеансе
<code>pg_advisory_unlock_shared(key bigint)</code>	<code>boolean</code>	Освобождает разделяемую блокировку на уровне сеанса
<code>pg_advisory_unlock_shared(key1 int, key2 int)</code>	<code>boolean</code>	Освобождает разделяемую блокировку на уровне сеанса
<code>pg_advisory_xact_lock(key bigint)</code>	<code>void</code>	Получает исключительную блокировку на уровне транзакции
<code>pg_advisory_xact_lock(key1 int, key2 int)</code>	<code>void</code>	Получает исключительную блокировку на уровне транзакции
<code>pg_advisory_xact_lock_shared(key bigint)</code>	<code>void</code>	Получает разделяемую блокировку на уровне транзакции
<code>pg_advisory_xact_lock_shared(key1 int, key2 int)</code>	<code>void</code>	Получает разделяемую блокировку на уровне транзакции
<code>pg_try_advisory_lock(key bigint)</code>	<code>boolean</code>	Получает исключительную блокировку на уровне сеанса, если это возможно
<code>pg_try_advisory_lock(key1 int, key2 int)</code>	<code>boolean</code>	Получает исключительную блокировку на уровне сеанса, если это возможно
<code>pg_try_advisory_lock_shared(key bigint)</code>	<code>boolean</code>	Получает разделяемую блокировку на уровне сеанса, если это возможно
<code>pg_try_advisory_lock_shared(key1 int, key2 int)</code>	<code>boolean</code>	Получает разделяемую блокировку на уровне сеанса, если это возможно
<code>pg_try_advisory_xact_lock(key bigint)</code>	<code>boolean</code>	Получает исключительную блокировку на уровне транзакции, если это возможно
<code>pg_try_advisory_xact_lock(key1 int, key2 int)</code>	<code>boolean</code>	Получает исключительную блокировку на уровне транзакции, если это возможно
<code>pg_try_advisory_xact_lock_shared(key bigint)</code>	<code>boolean</code>	Получает разделяемую блокировку на уровне транзакции, если это возможно
<code>pg_try_advisory_xact_lock_shared(key1 int, key2 int)</code>	<code>boolean</code>	Получает разделяемую блокировку на уровне транзакции, если это возможно

`pg_advisory_lock` блокирует определенный приложением ресурс, задаваемый одним 64-битным или двумя 32-битными ключами (заметьте, что их значения не пересекаются). Если идентификатор этого ресурса удерживает другой сеанс, эта функция не завершится, пока ресурс не станет доступным. Данная функция устанавливает блокировку в исключительном режиме. Если поступает сразу несколько запросов на блокировку, они накапливаются, так что если один ресурс был заблокирован три раза, его необходимо три раза разблокировать, чтобы он был доступен в других сеансах.

`pg_advisory_lock_shared` работает подобно `pg_advisory_lock`, но позволяет разделять блокировку с другими сеансами, запрашивающими её как разделяемую. Выполнение может быть приостановлено, только если другой сеанс запросил её в исключительном режиме.

`pg_try_advisory_lock` работает подобно `pg_advisory_lock`, но не ждет освобождения ресурса. Эта функция либо немедленно получает блокировку и возвращает `true`, либо сразу возвращает `false`, если получить её не удастся.

`pg_try_advisory_lock_shared` работает как `pg_try_advisory_lock`, но пытается получить разделяемую, а не исключительную блокировку.

`pg_advisory_unlock` освобождает ранее полученную исключительную блокировку на уровне сеанса. Если блокировка освобождена успешно, эта функция возвращает `true`, а если она не была занята – `false`, при этом сервер выдает предупреждение SQL.

`pg_advisory_unlock_shared` работает подобно `pg_advisory_unlock`, но освобождает разделяемую блокировку на уровне сеанса.

`pg_advisory_unlock_all` освобождает все блокировки на уровне сеанса, закрепленные за текущим сеансом. Эта функция неявно вызывается в конце любого сеанса, даже при нештатном отключении клиента.

`pg_advisory_xact_lock` работает подобно `pg_advisory_lock`, но её блокировка автоматически освобождается в конце текущей транзакции и не может быть освобождена явным образом.

`pg_advisory_xact_lock_shared` подобна функции `pg_advisory_lock_shared`, но её блокировка автоматически освобождается в конце текущей транзакции и не может быть освобождена явным образом.

`pg_try_advisory_xact_lock` работает подобно `pg_try_advisory_lock`, но её блокировка (если она была получена) автоматически освобождается в конце текущей транзакции и не может быть освобождена явным образом.

`pg_try_advisory_xact_lock_shared` работает подобно `pg_try_advisory_lock_shared`, но её блокировка (если она была получена) автоматически освобождается в конце текущей транзакции и не может быть освобождена явным образом.

1.28. Триггерные функции

В настоящее время в Синергия-БД есть единственная встроенная триггерная функция, `suppress_redundant_updates_trigger`, которая предотвращает изменения, фактически не влияющие на данные в строке, тогда как обычно изменения выполняются вне зависимости от того, были ли изменены данные. Обычное поведение не предполагает сравнения данных, поэтому изменения выполняются быстрее, и в ряде случаев именно это поведение желательно.

В идеале, следует избегать операций изменения, которые фактически не меняют данные в записях. Подобные ненужные изменения могут обходиться дорого, особенно когда требуется обновлять множество индексов, к тому же впоследствии базу данных придется очищать от «мертвых» строк. Однако выявить такие изменения в клиентском коде бывает сложно, если вообще возможно, а при составлении соответствующих проверочных выражений легко допустить ошибку. В качестве альтернативного решения можно использовать функцию `suppress_redundant_updates_trigger`, которая опускает изменения, не меняющие данные. Однако использовать её следует с осторожностью. Данный триггер выполняется не большое, но значительное время для каждой записи, так что если большинство записей все-таки фактически изменяются, этот триггер скорее замедлит операцию изменения.

Функцию `suppress_redundant_updates_trigger` можно привязать к таблице так:

```
CREATE TRIGGER z_min_update
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE PROCEDURE
suppress_redundant_updates_trigger();
```

В большинстве случаев этот триггер должен вызываться для каждой строки

последним. А так как триггеры вызываются по порядку сортировки их имен, имя для него нужно выбирать таким, чтобы оно было последним среди имен всех триггеров, которые могут быть в таблице.

Подробнее о создании триггеров можно узнать в описании CREATE TRIGGER.

1.29. Функции событийных триггеров

Синергия-БД предоставляет следующие вспомогательные функции для получения информации в событийных триггерах.

1.29.1. Получение изменений в конце команды

Функция `pg_event_trigger_ddl_commands` возвращает список команд DDL, выполняемых в результате действия пользователя. Вызывать её можно только в функции, реализующей событийный триггер `ddl_command_end`. При попытке вызвать её в любом другом контексте возникнет ошибка. Функция `pg_event_trigger_ddl_commands` возвращает одну строку для каждой базовой команды; для некоторых команд, записываемых в виде одного предложения SQL, может возвращаться несколько строк. Эта функция возвращает следующие столбцы:

Имя	Тип	Описание
<code>classid</code>	<code>oid</code>	OID каталога, к которому относится объект
<code>objid</code>	<code>oid</code>	OID самого объекта
<code>objsubid</code>	<code>integer</code>	Идентификатор подобъекта (например, номер для столбца)
<code>command tag</code>	<code>text</code>	Тег команды
<code>object type</code>	<code>text</code>	Тип объекта
<code>schema_name</code>	<code>text</code>	Имя схемы, к которой относится объект; если объект не относится ни к какой схеме – NULL. В кавычки имя не заключается.
<code>object_identity</code>	<code>text</code>	Текстовое представление идентификатора объекта, включающее схему. При необходимости компоненты этого идентификатора заключаются в кавычки.
<code>in extension</code>	<code>bool</code>	True, если команда является частью скрипта расширения
<code>command</code>	<code>pg_ddl_command</code>	Полное представление команды, во внутреннем формате. Его нельзя вывести непосредственно, но можно передать другим функциям, чтобы получить различные сведения о команде.

1.29.2. Обработка объектов, удалённых командой DDL

Функция `pg_event_trigger_dropped_objects` возвращает список всех объектов, удаленных командой, вызвавшей событие `sql_drop`. При вызове в

другом контексте `pg_event_trigger_dropped_objects` выдает ошибку.
`pg_event_trigger_dropped_objects` возвращает следующие колонки:

Имя	Тип	Описание
<code>classid</code>	<code>oid</code>	OID каталога, к которому относился объект
<code>objid</code>	<code>oid</code>	OID самого объекта
<code>objsubid</code>	<code>integer</code>	Идентификатор подобъекта (например, номер для столбца)
<code>original</code>	<code>bool</code>	True, если это один из корневых удаляемых объектов
<code>normal</code>	<code>bool</code>	True, если к этому объекту в графе зависимостей привело отношение обычной зависимости
<code>is_temporary</code>	<code>bool</code>	True, если объект был временным
<code>object_type</code>	<code>text</code>	Тип объекта
<code>schema_name</code>	<code>text</code>	Имя схемы, к которой относился объект; если объект не относился ни к какой схеме – NULL. В кавычки имя не заключается.
<code>object_name</code>	<code>text</code>	Имя объекта, если сочетание схемы и имени позволяет уникально идентифицировать объект; в противном случае – NULL. Имя не заключается в кавычки и не дополняется именем схемы.
<code>object_identity</code>	<code>text</code>	Текстовое представление идентификатора объекта, включающее схему. При необходимости компоненты этого идентификатора заключаются в кавычки.
<code>address_names</code>	<code>text[]</code>	Массив, который в сочетании с <code>object_type</code> и массивом <code>address_args</code> можно передать функции <code>pg_get_object_address()</code> , чтобы воссоздать адрес объекта на удалённом сервере, содержащем одноимённый объект того же рода.
<code>address_args</code>	<code>text[]</code>	Дополнение к массиву <code>address_names</code>

Функцию `pg_event_trigger_dropped_objects` можно использовать в событийном триггере так:

```
CREATE FUNCTION test_event_trigger_for_drops()
    RETURNS event_trigger LANGUAGE plpgsql AS $$
DECLARE
    obj record;
BEGIN
    FOR obj IN SELECT * FROM
pg_event_trigger_dropped_objects()
    LOOP
        RAISE NOTICE '% dropped object: % %.% %',
            tg_tag,
            obj.object_type,
            obj.schema_name,
            obj.object_name,
            obj.object_identity;
    END LOOP;
END
```

```
$$;
CREATE EVENT TRIGGER test_event_trigger_for_drops
    ON sql_drop
    EXECUTE FUNCTION test_event_trigger_for_drops();
```

1.29.3. Обработка события перезаписи таблицы

В таблице 1.89 показаны функции, выдающие информацию о таблице, для которой произошло событие перезаписи таблицы (table_rewrite). При попытке вызвать их в другом контексте возникнет ошибка.

Таблица 1.89 – Информация о перезаписи таблицы

Имя	Тип результата	Описание
pg_event_trigger_table_rewrite_oid()	Oid	OID таблицы, которая будет перезаписана.
pg_event_trigger_table_rewrite_reason()	int	Код причины, показывающий, чем вызвана перезапись. Точное значение кодов зависит от выпуска (версии).

Функцию pg_event_trigger_table_rewrite_oid можно использовать в событийном триггере так:

```
CREATE FUNCTION test_event_trigger_table_rewrite_oid()
    RETURNS event_trigger
    LANGUAGE plpgsql AS
$$
BEGIN
    RAISE NOTICE 'rewriting table % for reason %',
pg_event_trigger_table_rewrite_oid()::regclass,
        pg_event_trigger_table_rewrite_reason();
END;
$$;

CREATE EVENT TRIGGER test_table_rewrite_oid
    ON table_rewrite
    EXECUTE FUNCTION
test_event_trigger_table_rewrite_oid();
```

2. КОМАНДЫ SQL

Эта глава одержит справочную информацию о командах SQL, поддерживаемых СУБД «Синергия-БД». Под «SQL» понимается язык в общем; информация о соответствии стандарту SQL и совместимости с ним приведена на страницах, соответствующих каждой команде.

2.1. ABORT

ABORT – прервать текущую транзакцию.

2.1.1. Синтаксис

ABORT [WORK | TRANSACTION]

2.1.2. Описание

ABORT откатывает текущую транзакцию и приводит к отмене всех изменений, внесенных транзакцией. Эта команда ведет себя так же, как и стандартная SQL-команда ROLLBACK, и существует только по историческим причинам.

2.1.3. Параметры

WORK
TRANSACTION

Необязательные ключевые слова, не оказывают никакого влияния.

2.1.4. Замечания

Чтобы завершить и зафиксировать транзакцию, используйте COMMIT.

При выполнении команды ABORT вне блока транзакции выдается предупреждение и больше ничего не происходит.

2.1.5. Примеры

Чтобы прервать все операции:

ABORT;

2.1.6. Совместимость

Эта команда является расширением Синергия-БД и существует по

историческим причинам. Ей равнозначна стандартная SQL-команда ROLLBACK.

2.2. ALTER AGGREGATE

ALTER AGGREGATE – изменить определение агрегатной функции.

2.2.1. Синтаксис

```
ALTER AGGREGATE имя ( сигнатура_агр_функции ) RENAME TO  
новое_имя  
ALTER AGGREGATE имя ( сигнатура_агр_функции ) OWNER TO  
новый_владелец  
ALTER AGGREGATE имя ( сигнатура_агр_функции ) SET SCHEMA  
новая_схема
```

Здесь сигнатура_агр_функции:

```
* |  
[ режим_аргумента ] [ имя_аргумента ] тип_аргумента  
[ , ... ] |  
[ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента  
[ , ... ] ] ORDER BY [ режим_аргумента ]  
[ имя_аргумента ] тип_аргумента [ , ... ]
```

2.2.2. Описание

ALTER AGGREGATE изменяет определение агрегатной функции.

Выполнить ALTER AGGREGATE может только владелец соответствующей агрегатной функции. Чтобы сменить схему агрегатной функции, необходимо также иметь право CREATE в новой схеме. Чтобы сменить владельца, требуется также быть непосредственным или опосредованным членом новой роли, а эта роль должна иметь право CREATE в схеме агрегатной функции. С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать агрегатную функцию. Однако суперпользователь может сменить владельца агрегатной функции в любом случае.

2.2.3. Параметры

ИМЯ

Имя (возможно, дополненное схемой) существующей агрегатной функции.

режим_аргумента

Режим аргумента: IN или VARIADIC. По умолчанию подразумевается IN.

имя_аргумента

Имя аргумента. Заметьте, что на самом деле ALTER AGGREGATE не обращает внимание на имена аргументов, так как для однозначной идентификации агрегатной функции достаточно только типов аргументов.

тип_аргумента

Тип входных данных, с которыми работает агрегатная функция. Чтобы сослаться на агрегатную функцию без аргументов, укажите вместо списка аргументов *, а чтобы сослаться на сортирующую агрегатную функцию, добавьте ORDER BY между указаниями непосредственных и агрегируемых аргументов.

новое_имя

Новое имя агрегатной функции.

новый_владелец

Новый владелец агрегатной функции.

новая_схема

Новая схема агрегатной функции.

2.2.4. Замечания

Если вы хотите сослаться на сортирующую агрегатную функцию, рекомендуется добавить ORDER BY между непосредственными и агрегируемыми аргументами так же, как и в CREATE AGGREGATE. Однако, команда сработает и без ORDER BY, если непосредственные и агрегирующие аргументы перечислены подряд в одном списке. В такой сокращенной форме, если и в списке непосредственных, и в списке агрегирующих аргументов содержится VARIADIC «any», достаточно написать VARIADIC «any» только один раз.

2.2.5. Примеры

Переименование агрегатной функции myavg для типа integer в my_average:

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

Смена владельца агрегатной функции `myavg` для типа `integer` на `joe`:

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

Перемещение сортирующей агрегатной функции `mypercentile` с непосредственным аргументом типа `float8` и агрегируемым аргументом типа `integer` в схему `myschema`:

```
ALTER AGGREGATE mypercentile(float8 ORDER BY integer)  
SET SCHEMA myschema;
```

Это тоже будет работать:

```
ALTER AGGREGATE mypercentile(float8, integer) SET SCHEMA  
myschema;
```

2.2.6. Совместимость

Оператор `ALTER AGGREGATE` отсутствует в стандарте SQL.

2.3. ALTER COLLATION

`ALTER COLLATION` – изменить определение правила сортировки.

2.3.1. Синтаксис

```
ALTER COLLATION имя REFRESH VERSION  
ALTER COLLATION имя RENAME TO новое_имя  
ALTER COLLATION имя OWNER TO { новый_владелец |  
CURRENT_USER | SESSION_USER }  
ALTER COLLATION имя SET SCHEMA новая_схема
```

2.3.2. Описание

`ALTER COLLATION` изменяет определение правила сортировки.

Выполнить `ALTER COLLATION` может только владелец соответствующего правила сортировки. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме правила сортировки. С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать правило сортировки. Однако суперпользователь может сменить владельца правила сортировки в любом случае.

2.3.3. Параметры

ИМЯ

Имя (возможно, дополненное схемой) существующего правила сортировки.

новое_имя

Новое имя правила сортировки.

новый_владелец

Новый владелец правила сортировки.

новая_схема

Новая схема правила сортировки.

REFRESH VERSION

Обновить версию правила сортировки.

Когда применяются правила сортировки, предоставляемые библиотекой ICU, внутренняя версия сортировщика ICU записывается в системный каталог при создании объекта для данного правила. Когда такое правило используется, текущая версия сверяется с записанной и в случае несовпадения выдаётся предупреждение, например такое:

```
WARNING: collation "xx-x-icu" has version mismatch
```

```
DETAIL: The collation in the database was created using  
version 1.2.3.4, but the operating system provides  
version 2.3.4.5.
```

```
HINT: Rebuild all objects affected by this collation  
and run ALTER COLLATION pg_catalog."xx-x-icu" REFRESH  
VERSION, or build PostgreSQL with the right library  
version.
```

Изменения в определениях правил сортировки могут приводить к разрушению индексов и другим проблемам, так как СУБД рассчитывает на то, что хранимые объекты отсортированы в определённом порядке. Вообще этого следует избегать, но это может иметь место в совершенно легальных обстоятельствах, например при обновлении с помощью `pg_upgrade` исполняемых файлов сервера, скомпонованных с более новой версией ICU. Когда возникает такая ситуация, все объекты, зависящие от данного правила сортировки, должны быть перестроены, например, командой `REINDEX`. После этой операции можно обновить версию правила сортировки, выполнив команду `ALTER COLLATION ... REFRESH`

VERSION. При этом системный каталог будет обновлён, в него будет записана текущая версия сортировщика, и предупреждение уйдёт. Заметьте, что эта команда собственно не проверяет, были ли все зависимые объекты перестроены корректно.

Следующий запрос позволяет выбрать все правила сортировки в текущей базе данных, которые требуют обновления, и зависящие от них объекты:

```
SELECT pg_describe_object(refclassid, refobjid,
refobjsubid) AS "Collation",
       pg_describe_object(classid, objid, objsubid) AS
"Object"
FROM pg_depend d JOIN pg_collation c
      ON refclassid = 'pg_collation'::regclass AND
refobjid = c.oid
WHERE c.collversion <>
pg_collation_actual_version(c.oid)
ORDER BY 1, 2;
```

2.3.4. Примеры

Переименование правила сортировки de_DE в german:

```
ALTER COLLATION «de_DE» RENAME TO german;
```

Изменение владельца правила сортировки en_US на joe:

```
ALTER COLLATION «en_US» OWNER TO joe;
```

2.3.5. Совместимость

Оператор ALTER COLLATION отсутствует в стандарте SQL.

2.4. ALTER CONVERSION

ALTER CONVERSION – изменить определение перекодировки.

2.4.1. Синтаксис

```
ALTER CONVERSION имя RENAME TO новое_имя
ALTER CONVERSION имя OWNER TO { новый_владелец |
CURRENT_USER | SESSION_USER }
ALTER CONVERSION имя SET SCHEMA новая_схема
```

2.4.2. Описание

ALTER CONVERSION изменяет определение перекодировки.

Выполнить ALTER CONVERSION может только владелец соответствующей перекодировки. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право CREATE в схеме перекодировки. С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать перекодировку. Однако суперпользователь может сменить владельца перекодировки в любом случае.

2.4.3. Параметры

ИМЯ

Имя (возможно, дополненное схемой) существующей перекодировки.

новое_имя

Новое имя перекодировки.

новый_владелец

Новый владелец перекодировки.

новая_схема

Новая схема перекодировки.

2.4.4. Примеры

Переименование перекодировки iso_8859_1_to_utf8 в
latin1_to_unicode:

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO  
latin1_to_unicode;
```

Смена владельца перекодировки iso_8859_1_to_utf8 на joe:

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

2.4.5. Совместимость

Оператор ALTER CONVERSION отсутствует в стандарте SQL.

2.5. ALTER DATABASE

ALTER DATABASE – изменить атрибуты базы данных.

2.5.1. Синтаксис

```
ALTER DATABASE имя [ [ WITH ] параметр [ ... ] ]
```

Здесь параметр:

```
ALLOW_CONNECTIONS разр_подключения  
CONNECTION LIMIT предел_подключений  
IS_TEMPLATE это_шаблон
```

```
ALTER DATABASE имя RENAME TO новое_имя
```

```
ALTER DATABASE имя OWNER TO { новый_владелец |  
CURRENT_USER | SESSION_USER }
```

```
ALTER DATABASE имя SET TABLESPACE  
новое_табл_пространство
```

```
ALTER DATABASE имя SET параметр_конфигурации { TO | = }  
{ значение | DEFAULT }
```

```
ALTER DATABASE имя SET параметр_конфигурации FROM  
CURRENT
```

```
ALTER DATABASE имя RESET параметр_конфигурации
```

```
ALTER DATABASE имя RESET ALL
```

2.5.2. Описание

ALTER DATABASE изменяет атрибуты базы данных.

Первая форма оператора меняет параметры на уровне базы данных. (Подробнее описано ниже.) Изменять эти параметры может только владелец базы данных или суперпользователь.

Вторая форма меняет имя базы данных. Переименовать базу данных может только владелец БД или суперпользователь (не суперпользователю требуется также право CREATEDB). Переименовать текущую базу данных нельзя. Если вам нужно сделать это, сначала подключитесь к другой базе.

Третья форма меняет владельца базы данных. Чтобы сменить владельца базы, необходимо быть её владельцем и также непосредственным или опосредованным членом новой роли-владельца, и кроме того, иметь право CREATEDB. Заметьте, что суперпользователи наделяются всеми этими правами автоматически.

Четвертая форма меняет основное табличное пространство базы данных

(табличное пространство по умолчанию). Произвести это изменение может только её владелец или суперпользователь; кроме того, необходимо иметь право для создания нового табличного пространства. Эта команда физически переносит все таблицы или индексы из прежнего основного табличного пространства БД в новое. Заметьте, что таблицы и индексы, находящиеся не в основном табличном пространстве, при этом не затрагиваются.

Остальные формы меняют значение по умолчанию конфигурационных переменных времени выполнения для базы данных Синергия-БД. Когда устанавливается следующий сеанс работы с указанной базой данных, заданное этой командой значение становится значением по умолчанию. Значения переменных, заданные для базы данных, переопределяют значения, определенные в `postgresql.conf` или полученные через командную строку `postgres`. Менять сеансовые значения переменных для базы данных может только её владелец или суперпользователь. Некоторые параметры изменить таким образом нельзя, а некоторые может изменить только суперпользователь.

2.5.3. Параметры

имя

Имя базы данных, атрибуты которой изменяются.

разр_подключения

Если `false`, никто не сможет подключаться к этой базе данных.

предел_подключений

Число разрешённых одновременно подключений к этой базе данных (-1 снимает ограничение).

это_шаблон

Если `true`, базу данных сможет клонировать любой пользователь с правами `CREATEDB`; в противном случае клонировать эту базу смогут только суперпользователи и её владелец.

новое_имя

Новое имя базы данных.

новый_владелец

Новый владелец базы данных.

новое_табл_пространство

Новое основное табличное пространство базы данных.

Эту форму команды нельзя выполнять внутри блока транзакции.

параметр_конфигурации
значение

Устанавливает сеансовое значение по умолчанию для указанного параметра конфигурации. Если указывается значение DEFAULT или равнозначный вариант, RESET, определение параметра на уровне базы данных удаляется, так что в новых сеансах будет действовать значение по умолчанию, определённое на уровне системы. Для очистки всех значений параметров на уровне базы данных выполните RESET ALL. SET FROM CURRENT устанавливает значение параметра на уровне базы данных из текущего значения в активном сеансе.

2.5.4. Замечания

Также возможно связать параметры сеанса не с базой данных, а с определенной ролью. В случае конфликта параметры на уровне роли переопределяют параметры на уровне базы данных.

2.6. ALTER DEFAULT PRIVILEGES

ALTER DEFAULT PRIVILEGES – определить права доступа по умолчанию.

2.6.1. Синтаксис

```
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } целевая_роль [, ...] ]
    [ IN SCHEMA имя_схемы [, ...] ]
    предложение_GRANT_или_REVOKE
```

Где возможные варианты предложение_GRANT_или_REVOKE:

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE |
REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
ON TABLES
```

```
TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH  
GRANT OPTION ]
```

```
GRANT { { USAGE | SELECT | UPDATE }  
      [, ...] | ALL [ PRIVILEGES ] }  
ON SEQUENCES  
TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH  
GRANT OPTION ]
```

```
GRANT { EXECUTE | ALL [ PRIVILEGES ] }  
ON { FUNCTIONS | ROUTINES }  
TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH  
GRANT OPTION ]
```

```
GRANT { USAGE | ALL [ PRIVILEGES ] }  
ON TYPES  
TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH  
GRANT OPTION ]
```

```
GRANT { USAGE | CREATE | ALL [ PRIVILEGES ] }  
ON SCHEMAS  
TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH  
GRANT OPTION ]
```

```
REVOKE [ GRANT OPTION FOR ]  
      { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE |  
REFERENCES | TRIGGER }  
      [, ...] | ALL [ PRIVILEGES ] }  
ON TABLES  
FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]  
      [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]  
      { { USAGE | SELECT | UPDATE }  
      [, ...] | ALL [ PRIVILEGES ] }
```

```
ON SEQUENCES
FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
{ EXECUTE | ALL [ PRIVILEGES ] }
ON { FUNCTIONS | ROUTINES }
FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
{ USAGE | ALL [ PRIVILEGES ] }
ON TYPES
FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
{ USAGE | CREATE | ALL [ PRIVILEGES ] }
ON SCHEMAS
FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]
```

2.6.2. Описание

ALTER DEFAULT PRIVILEGES позволяет задавать права, применяемые к объектам, которые будут создаваться в будущем. Эта команда не затрагивает права, назначенные уже существующим объектам. В настоящее время можно задавать права только для таблиц (включая представления и сторонние таблицы), последовательностей, функций и типов (включая домены). Применительно к данной команде функциями считаются также агрегатные функции и процедуры. Слова FUNCTIONS и ROUTINES для неё считаются равнозначными. ROUTINES предпочтительнее в перспективе как стандартный термин, охватывающий и функции, и процедуры. Задать права по умолчанию для функций и процедур по отдельности нельзя.

Вы можете изменить права по умолчанию только для объектов, которые будут созданы вами или ролями, членами которых вы являетесь. Права можно задать глобально (т.е. для всех объектов, создаваемых в текущей базе данных) или для определенных схем. Заданные на уровне схемы права по умолчанию добавляются к тем, что определены глобально для конкретного типа объекта.

Как объясняется в GRANT, права по умолчанию для объектов любого типа обычно дают все назначаемые разрешения владельцу объекта, а также могут давать некоторые разрешения роли PUBLIC. Однако это поведение можно поменять, изменив права по умолчанию командой ALTER DEFAULT PRIVILEGES.

2.6.3. Параметры

целевая_роль

Имя существующей роли, членом которой является текущая. Если FOR ROLE опущено, подразумевается текущая роль.

имя_схемы

Имя существующей схемы. Если указано, права по умолчанию меняются для объектов, которые будут созданы в этой схеме. Если IN SCHEMA опущено, меняются глобальные права по умолчанию.

имя_роли

Имя существующей роли, для которой даются или отзываются права. Этот и все другие параметры в предложении_grant_или_revoke действуют как описано в GRANT или REVOKE, за исключением того, что они распространяются не на один конкретный объект, а на целый класс объектов.

2.6.4. Замечания

Чтобы узнать текущие назначенные права по умолчанию, воспользуйтесь командой \ddp в psql. Интерпретация значений прав приведена в описании команды \dp в GRANT.

Если вы желаете удалить роль, права по умолчанию для которой были изменены, необходимо явно отменить изменения прав по умолчанию или воспользоваться командой DROP OWNED BY для избавления от назначенных для

этой роли прав по умолчанию.

2.6.5. Примеры

Наделение всех правом SELECT для всех таблиц (и представлений), которые будут созданы в дальнейшем в схеме myschema, и наделение роли webuser правом INSERT для этих же таблиц:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT  
ON TABLES TO PUBLIC;
```

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT INSERT  
ON TABLES TO webuser;
```

Отмена предыдущих изменений с тем, чтобы для таблиц, создаваемых в будущем, были определены только обычные права, без дополнительных:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE  
SELECT ON TABLES FROM PUBLIC;
```

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE  
INSERT ON TABLES FROM webuser;
```

Лишение роли public права на выполнение (EXECUTE), которое обычно дается для функций (для всех функций, которые будут созданы ролью admin):

```
ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE  
ON FUNCTIONS FROM PUBLIC;
```

2.6.6. Совместимость

Оператор ALTER DEFAULT PRIVILEGES отсутствует в стандарте SQL.

2.7. ALTER DOMAIN

ALTER DOMAIN – изменить определение домена.

2.7.1. Синтаксис

```
ALTER DOMAIN имя  
    { SET DEFAULT выражение | DROP DEFAULT }  
ALTER DOMAIN имя  
    { SET | DROP } NOT NULL  
ALTER DOMAIN имя  
    ADD ограничение_домена [ NOT VALID ]  
ALTER DOMAIN имя
```

```
DROP CONSTRAINT [ IF EXISTS ] имя_ограничения  
[ RESTRICT | CASCADE ]
```

```
ALTER DOMAIN имя
```

```
    RENAME CONSTRAINT имя_ограничения TO  
    имя_нового_ограничения
```

```
ALTER DOMAIN имя
```

```
    VALIDATE CONSTRAINT имя_ограничения  
ALTER DOMAIN имя
```

```
    OWNER TO { новый_владелец | CURRENT_USER |  
SESSION_USER }
```

```
ALTER DOMAIN имя
```

```
    RENAME TO новое_имя  
ALTER DOMAIN имя
```

```
    SET SCHEMA новая_схема
```

2.7.2. Описание

ALTER DOMAIN изменяет определение существующего домена. Эта команда имеет несколько разновидностей:

- SET/DROP DEFAULT

Эти формы задают/убирают значение по умолчанию для домена. Заметьте, что эти значения по умолчанию применяются только при последующих командах INSERT; они не меняются в строках с данным доменом, уже добавленных в таблицу;

- SET/DROP NOT NULL

Эти формы определяют, будет ли домен принимать значения NULL или нет. SET NOT NULL можно выполнить, только если колонки с этим доменом ещё не содержат значений NULL;

- ADD ограничение_домена [NOT VALID]

Эта форма добавляет новое ограничение для домена с тем же синтаксисом, что описан в CREATE DOMAIN. Когда добавляется новое ограничение домена, все колонки с этим доменом будут проверены на соответствие этому ограничению. Эти проверки можно подавить, добавив указание NOT VALID, а затем активировать позднее с помощью команды ALTER DOMAIN ... VALIDATE

CONSTRAINT. Вновь вставленные или изменённые строки всегда проверяются по всем ограничениям, даже тем, что отмечены как NOT VALID. Указание NOT VALID допускается только для ограничений CHECK;

- DROP CONSTRAINT [IF EXISTS]

Эта форма убирает ограничения домена. Если указано IF EXISTS и заданное ограничение не существует, это не считается ошибкой. В этом случае выдается только замечание;

- RENAME CONSTRAINT

Эта форма меняет название ограничения домена;

- VALIDATE CONSTRAINT

Эта форма включает проверку ограничения, ранее добавленного как NOT VALID, то есть, проверяет все данные в колонках с этим доменом на соответствие этому ограничению;

- OWNER

Эта форма меняет владельца домена на заданного пользователя;

- RENAME

Эта форма меняет название домена;

- SET SCHEMA

Эта форма меняет схему домена. Все ограничения, связанные с данным доменом, так же переносятся в новую схему.

Выполнить ALTER DOMAIN может только владелец соответствующего домена. Чтобы сменить схему домена, необходимо также иметь право CREATE в новой схеме. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право CREATE в схеме домена. С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать домен. Однако суперпользователь может сменить владельца домена в любом случае.

2.7.3. Параметры

имя

Имя (возможно, дополненное схемой) существующего домена, подлежащего изменению.

ограничение_домена

Новое ограничение домена.

имя_ограничения

Имя существующего ограничения, подлежащего удалению или переименованию.

NOT VALID

Не проверять существующие данные в колонках на соответствие ограничению.

CASCADE

Автоматически удалять объекты, зависящие от данного ограничения.

RESTRICT

Отказать в удалении ограничения, если существуют зависящие от него объекты. Это поведение по умолчанию.

новое_имя

Новое имя домена.

имя_нового_ограничения

Новое имя ограничения.

новый_владелец

Имя пользователя, назначаемого новым владельцем домена.

новая_схема

Новая схема домена.

2.7.4. Замечания

В настоящее время команды ALTER DOMAIN ADD CONSTRAINT, ALTER DOMAIN VALIDATE CONSTRAINT и ALTER DOMAIN SET NOT NULL выдают ошибку, если указанный домен или любой производный от него используется в столбце с типом-контейнером (это может быть составной, диапазонный тип или массив) в какой-либо таблице базы данных. В дальнейшем они будут доработаны, с тем чтобы новое ограничение проверялось и при такой вложенности.

2.7.5. Примеры

Добавление ограничения NOT NULL к домену:

```
ALTER DOMAIN zipcode SET NOT NULL;
```

Удаление ограничения NOT NULL из домена:

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

Добавление ограничения-проверки к домену:

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK  
(char_length(VALUE) = 5);
```

Удаление ограничения-проверки из домена:

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

Переименование ограничения-проверки в домене:

```
ALTER DOMAIN zipcode RENAME CONSTRAINT zipchk TO  
zip_check;
```

Перемещение домена в другую схему:

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

2.7.6. Совместимость

ALTER DOMAIN соответствует стандарту SQL, за исключением подвидов OWNER, RENAME, SET SCHEMA и VALIDATE CONSTRAINT, которые являются расширениями Синергия-БД. Предложение NOT VALID вариации ADD CONSTRAINT также является расширением Синергия-БД.

2.8. ALTER EVENT TRIGGER

ALTER EVENT TRIGGER – изменить определение событийного триггера.

2.8.1. Синтаксис

```
ALTER EVENT TRIGGER имя DISABLE
```

```
ALTER EVENT TRIGGER имя ENABLE [ REPLICA | ALWAYS ]
```

```
ALTER EVENT TRIGGER имя OWNER TO { новый_владелец |  
CURRENT_USER | SESSION_USER }
```

```
ALTER EVENT TRIGGER имя RENAME TO новое_имя
```

2.8.2. Описание

ALTER EVENT TRIGGER изменяет свойства существующего событийного

триггера.

Для изменения событийного триггера нужно быть суперпользователем.

2.8.3. Параметры

ИМЯ

Имя существующего триггера, подлежащего изменению.

новый_владелец

Имя пользователя, назначаемого новым владельцем событийного триггера.

новое_имя

Новое имя событийного триггера.

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

Эти формы настраивают срабатывание событийных триггеров. Отключенный триггер сохраняется в системе, но не выполняется, когда происходит его событие срабатывания.

2.8.4. Совместимость

Оператор ALTER EVENT TRIGGER отсутствует в стандарте SQL.

2.9. ALTER EXTENSION

ALTER EXTENSION – изменить определение расширения.

2.9.1. Синтаксис

ALTER EXTENSION имя UPDATE [TO новая_версия]

ALTER EXTENSION имя SET SCHEMA новая_схема

ALTER EXTENSION имя ADD элемент_объект

ALTER EXTENSION имя DROP элемент_объект

Здесь элемент_объект:

ACCESS METHOD имя_объекта |

AGGREGATE имя_агрегатной_функции

(сигнатура_агр_функции) |

CAST (исходный_тип AS целевой_тип) |

COLLATION имя_объекта |

CONVERSION имя_объекта |

DOMAIN имя_объекта |
EVENT TRIGGER имя_объекта |
FOREIGN DATA WRAPPER имя_объекта |
FOREIGN TABLE имя_объекта |
FUNCTION имя_функции [([[режим_аргумента]
[имя_аргумента] тип_аргумента [, ...]])] |
MATERIALIZED VIEW имя_объекта |
OPERATOR имя_оператора (тип_слева, тип_справа) |
OPERATOR CLASS имя_объекта USING метод_индекса |
OPERATOR FAMILY имя_объекта USING метод_индекса |
[PROCEDURAL] LANGUAGE имя_объекта |
PROCEDURE имя_процедуры [([[режим_аргумента]
[имя_аргумента] тип_аргумента [, ...]])] |
ROUTINE имя_подпрограммы [([[режим_аргумента]
[имя_аргумента] тип_аргумента [, ...]])] |
SCHEMA имя_объекта |
SEQUENCE имя_объекта |
SERVER имя_объекта |
TABLE имя_объекта |
TEXT SEARCH CONFIGURATION имя_объекта |
TEXT SEARCH DICTIONARY имя_объекта |
TEXT SEARCH PARSER имя_объекта |
TEXT SEARCH TEMPLATE имя_объекта |
TRANSFORM FOR имя_типа LANGUAGE имя_языка |
TYPE имя_объекта |
VIEW имя_объекта

и сигнатура_агр_функции:

* |
[режим_аргумента] [имя_аргумента] тип_аргумента
[, ...] |
[[режим_аргумента] [имя_аргумента] тип_аргумента
[, ...]] ORDER BY [режим_аргумента]
[имя_аргумента] тип_аргумента [, ...]

2.9.2. Описание

`ALTER EXTENSION` изменяет определение установленного расширения. Эта команда имеет несколько подвидов:

- `UPDATE`

Эта форма обновляет версию расширения. Расширение должно предоставлять подходящий скрипт обновления (или набор скриптов), который может сменить текущую установленную версию на требуемую;

- `SET SCHEMA`

Эта форма переносит объекты расширения в другую схему. Чтобы эта команда выполнялась успешно, расширение должно быть перемещаемым;

- `ADD элемент_объект`

Эта форма добавляет существующий объект в расширение. В основном это применяется в скриптах обновления расширений. Добавленный объект затем будет считаться частью расширения, и удалить его можно будет, только удалив расширение;

- `DROP элемент_объект`

Эта форма удаляет из расширения включенный в него объект. В основном это применяется в скриптах обновления расширений. Сам объект при этом не уничтожается, а только отделяется от расширения.

Чтобы выполнить команду `ALTER EXTENSION`, необходимо быть владельцем данного расширения. Для форм `ADD/DROP` требуется также быть владельцем добавляемого/удаляемого объекта.

2.9.3. Параметры

имя

Имя установленного расширения.

новая_версия

Запрашиваемая новая версия расширения. Ее можно записать в виде идентификатора или строкового значения. Если она не указана, `ALTER EXTENSION UPDATE` пытается выполнить обновление до версии, указанной в качестве версии по умолчанию в управляющем файле расширения.

новая_схема

Новая схема расширения.

имя_объекта

имя_агрегатной_функции

имя_функции

имя_оператора

имя_процедуры

имя_подпрограммы

Имя объекта, добавляемого или удаляемого из расширения. Имена таблиц, агрегатных функций, доменов, сторонних таблиц, функций, операторов, классов операторов, семейств операторов, последовательностей, объектов текстового поиска, типов и представлений можно дополнить именем схемы.

исходный_тип

Имя исходного типа данных для приведения.

целевой_тип

Имя целевого типа данных для приведения.

режим_аргумента

Режим аргумента обычной или агрегатной функции: IN, OUT, INOUT или VARIADIC. По умолчанию подразумевается IN. Заметьте, что ALTER EXTENSION на самом деле не учитывает аргументы OUT, так как для идентификации функции нужны только типы входных аргументов. Поэтому достаточно перечислить только аргументы IN, INOUT и VARIADIC.

имя_аргумента

Имя аргумента обычной или агрегатной функции. Заметьте, что на самом деле ALTER EXTENSION не обращает внимание на имена аргументов, так как для однозначной идентификации функции достаточно только типов аргументов.

тип_аргумента

Тип данных аргумента обычной или агрегатной функции.

тип_слева

тип_справа

Тип данных аргументов оператора (возможно, дополненный именем схемы). В случае отсутствия аргумента префиксного или постфиксного оператора укажите вместо типа NONE.

PROCEDURAL

Это слово не несет смысловой нагрузки.

имя_типа

Имя типа данных, для которого предназначена трансформация.

имя_языка

Имя языка, для которого предназначена трансформация.

2.9.4. Примеры

Обновление расширения hstore до версии 2.0:

```
ALTER EXTENSION hstore UPDATE TO '2.0';
```

Смена схемы расширения hstore на utils:

```
ALTER EXTENSION hstore SET SCHEMA utils;
```

Добавление существующей функции в расширение hstore:

```
ALTER EXTENSION hstore ADD FUNCTION  
populate_record(anyelement, hstore);
```

2.9.5. Совместимость

Оператор ALTER EXTENSION является расширением Синергия-БД.

2.10.ALTER FOREIGN DATA WRAPPER

ALTER FOREIGN DATA WRAPPER – изменить определение обертки сторонних данных.

2.10.1. Синтаксис

```
ALTER FOREIGN DATA WRAPPER имя
```

```
    [ HANDLER функция_обработчик | NO HANDLER ]
```

```
    [ VALIDATOR функция_проверки | NO VALIDATOR ]
```

```
    [ OPTIONS ( [ ADD | SET | DROP ] параметр  
    ['значение'] [, ... ] ) ]
```

```
ALTER FOREIGN DATA WRAPPER имя OWNER TO { новый_владелец  
| CURRENT_USER | SESSION_USER }
```

```
ALTER FOREIGN DATA WRAPPER имя RENAME TO новое_имя
```

2.10.2. Описание

ALTER FOREIGN DATA WRAPPER изменяет определение обертки сторонних

данных. Первая форма команды меняет вспомогательные функции или общие параметры обертки (требуется минимум одно предложение), а вторая – владельца обертки.

Настраивать обертки сторонних данных могут только суперпользователи и только суперпользователи могут быть их владельцами.

2.10.3. Параметры

ИМЯ

Имя существующей обертки сторонних данных.

HANDLER функция_обработчик

Задаёт новое имя функции-обработчика для обертки сторонних данных.

NO HANDLER

Эти ключевые слова указывают, что обертка сторонних данных теперь не имеет функции-обработчика. Заметьте, что обращаться к сторонним таблицам, если их обертка сторонних данных не имеет обработчика, нельзя.

VALIDATOR функция_проверки

Задаёт новое имя функции проверки для обертки сторонних данных.

Заметьте, что возможна ситуация, что предыдущие параметры обертки данных, зависящих серверов, сопоставлений пользователей или сторонних таблиц окажутся неприемлемыми для новой функции проверки. Синергия-БД не проверяет их, поэтому пользователь сам должен убедиться в правильности этих параметров, прежде чем использовать измененную обертку данных. Однако параметры, изменяемые в данной команде ALTER FOREIGN DATA WRAPPER, будут проверены новой функцией проверки.

NO VALIDATOR

Эти ключевые слова указывают, что обертка сторонних данных теперь не имеет функции проверки.

OPTIONS ([ADD | SET | DROP] параметр ['значение']
[, ...])

Эта форма настраивает параметры обертки сторонних данных. ADD, SET и DROP определяют, какое действие будет выполнено (добавление, установка и удаление, соответственно). Если действие не задано явно, подразумевается ADD.

Имена параметров должны быть уникальными, они вместе со значениями проверяются функцией проверки, если она установлена.

новый_владелец

Имя пользователя, назначаемого новым владельцем обертки сторонних данных.

новое_имя

Новое имя обертки сторонних данных.

2.10.4. Примеры

Изменение параметров обертки сторонних данных dbi: добавление параметра foo, удаление bar:

```
ALTER FOREIGN DATA WRAPPER dbi OPTIONS (ADD foo '1',  
DROP 'bar');
```

Установление для обертки сторонних данных dbi новой функции проверки bob.myvalidator:

```
ALTER FOREIGN DATA WRAPPER dbi VALIDATOR  
bob.myvalidator;
```

2.10.5. Совместимость

ALTER FOREIGN DATA WRAPPER соответствует стандарту ISO/IEC 9075-9 (SQL/MED), за исключением предложений HANDLER, VALIDATOR, OWNER TO и RENAME, являющихся расширениями.

2.11.ALTER FOREIGN TABLE

ALTER FOREIGN TABLE – изменить определение сторонней таблицы.

2.11.1. Синтаксис

```
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]  
    действие [, ... ]  
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]  
    RENAME [ COLUMN ] имя_столбца TO новое_имя_столбца  
ALTER FOREIGN TABLE [ IF EXISTS ] имя  
    RENAME TO новое_имя
```

```
ALTER FOREIGN TABLE [ IF EXISTS ] имя
    SET SCHEMA новая_схема
```

Где действие может быть следующим:

```
ADD [ COLUMN ] имя_столбца тип_данных [ COLLATE
правило_сортировки ] [ ограничение_столбца [ ... ] ]
DROP [ COLUMN ] [ IF EXISTS ] имя_столбца [ RESTRICT
| CASCADE ]
ALTER [ COLUMN ] имя_столбца [ SET DATA ] TYPE
тип_данных [ COLLATE правило_сортировки ]
ALTER [ COLUMN ] имя_столбца SET DEFAULT выражение
ALTER [ COLUMN ] имя_столбца DROP DEFAULT
ALTER [ COLUMN ] имя_столбца { SET | DROP } NOT NULL
ALTER [ COLUMN ] имя_столбца SET STATISTICS integer
ALTER [ COLUMN ] имя_столбца SET ( атрибут =
значение [, ... ] )
ALTER [ COLUMN ] имя_столбца RESET ( атрибут
[, ... ] )
ALTER [ COLUMN ] имя_столбца SET STORAGE { PLAIN |
EXTERNAL | EXTENDED | MAIN }
ALTER [ COLUMN ] имя_столбца OPTIONS ( [ ADD | SET |
DROP ] параметр ['значение'] [, ... ] )
ADD ограничение_таблицы [ NOT VALID ]
VALIDATE CONSTRAINT имя_ограничения
DROP CONSTRAINT [ IF EXISTS ] имя_ограничения
[ RESTRICT | CASCADE ]
DISABLE TRIGGER [ имя_триггера | ALL | USER ]
ENABLE TRIGGER [ имя_триггера | ALL | USER ]
ENABLE REPLICA TRIGGER имя_триггера
ENABLE ALWAYS TRIGGER имя_триггера
SET WITH OIDS
SET WITHOUT OIDS
INHERIT таблица_родитель
NO INHERIT таблица_родитель
OWNER TO { новый_владелец | CURRENT_USER |
SESSION_USER }
```

```
OPTIONS ( [ ADD | SET | DROP ] параметр ['значение']  
[, ... ])
```

2.11.2. Описание

ALTER FOREIGN TABLE меняет определение существующей сторонней таблицы. Эта команда имеет несколько разновидностей:

- ADD COLUMN

Эта форма добавляет в стороннюю таблицу новую колонку, следуя тому же синтаксису, что и CREATE FOREIGN TABLE. В отличие от добавления колонки в обычную таблицу, при данной операции в базовом хранилище ничего не меняется; эта команда просто объявляет о доступности новой колонки через данную стороннюю таблицу;

- DROP COLUMN [IF EXISTS]

Эта форма удаляет колонку из сторонней таблицы. Если что-либо зависит от этой колонки, например, представление, для успешного результата потребуется добавить CASCADE. Если указано IF EXISTS и эта колонка не существует, ошибка не происходит, вместо этого выдается замечание;

- SET DATA TYPE

Эта форма меняет тип колонки в сторонней таблице. И это не влияет на нижележащее хранилище: данная операция просто меняет тип, который по мнению Синергия-БД будет иметь этот столбец;

- SET/DROP DEFAULT

Эти формы задают или удаляют значение по умолчанию для колонок. Значения по умолчанию применяются только при последующих командах INSERT или UPDATE; их изменения не отражаются в строках, уже существующих в таблице;

- SET/DROP NOT NULL

Устанавливает, будет ли колонка принимать значения NULL или нет;

- SET STATISTICS

Эта форма задает цель сбора статистики по колонкам для последующих операций ANALYZE. За подробностями обратитесь к описанию подобной формы ALTER TABLE;

- SET (атрибут = значение [, ...])

RESET (атрибут [, ...])

Эта форма задает или сбрасывает значения атрибутов. За подробностями обратитесь к описанию подобной формы ALTER TABLE;

- SET STORAGE

Эта форма задаёт режим хранения для столбца. За подробностями обратитесь к описанию подобной формы ALTER TABLE. Заметьте, что режим хранения не имеет значения, если обёртка сторонних данных для этой таблицы будет игнорировать его;

- ADD *ограничение_таблицы* [NOT VALID]

Эта форма добавляет новое ограничение в стороннюю таблицу с применением того же синтаксиса, что и CREATE FOREIGN TABLE. В настоящее время поддерживаются только ограничения CHECK.

В отличие от ограничения, добавляемого для обычной таблицы, ограничение сторонней таблицы фактически никак не проверяется; эта команда сводится просто к заявлению о том, что все строки в сторонней таблице предположительно удовлетворяют новому условию. Подробнее это рассматривается в описании CREATE FOREIGN TABLE. Если ограничение помечено как NOT VALID (непроверенное), сервер не будет полагать, что оно выполняется; такая запись делается только на случай использования в будущем;

- VALIDATE CONSTRAINT

Эта форма отмечает ограничение, которая ранее было помечено NOT VALID, как проверенное. Собственно для проверки этого ограничения ничего не делается, но последующие запросы будут полагать, что оно действует;

- DROP CONSTRAINT [IF EXISTS]

Эта форма удаляет указанное ограничение сторонней таблицы. Если указано IF EXISTS и заданное ограничение не существует, это не считается ошибкой. В этом случае выдаётся только замечание;

- DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

Эти формы управляют триггерами, принадлежащими сторонней таблице. За подробностями обратитесь к описанию подобной формы ALTER TABLE;

- SET WITH OIDS

Эта форма добавляет в таблицу системный столбец `oid`. Если в таблице уже есть такой столбец, она не делает ничего. Обёртка сторонних данных должна поддерживать `OID`, иначе из этого столбца будут читаться просто нулевые значения.

Заметьте, что это не равнозначно команде `ADD COLUMN oid oid` (эта команда добавит не системный, а обычный столбец с подходящим именем `oid`);

- `SET WITHOUT OIDS`

Эта форма удаляет из таблицы системный столбец `oid`. Это в точности равнозначно `DROP COLUMN oid RESTRICT`, за исключением того, что в случае отсутствия столбца `oid` ошибки не будет;

- `INHERIT таблица_родитель`

Эта форма делает целевую стороннюю таблицу потомком указанной родительской таблицы. За подробностями обратитесь к описанию подобной формы `ALTER TABLE`;

- `NO INHERIT таблица_родитель`

Эта форма удаляет целевую стороннюю таблицу из списка потомков указанной родительской таблицы;

- `OWNER`

Эта форма меняет владельца сторонней таблицы на заданного пользователя.

- `OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ] )`

Эта форма настраивает параметры сторонней таблицы или одной из её колонок. `ADD`, `SET` и `DROP` определяют, какое действие будет выполнено (добавление, установка и удаление, соответственно). Если действие не задано явно, подразумевается `ADD`. Имена параметров не должны повторяться (хотя параметр таблицы и параметр колонки вполне могут иметь одно имя). Имена и значения параметров также проверяются библиотекой обертки сторонних данных;

- `RENAME`

Формы `RENAME` меняют имя сторонней таблицы или имя колонки в сторонней таблице;

- SET SCHEMA

Эта форма переносит стороннюю таблицу в другую схему.

Все действия, кроме RENAME и SET SCHEMA, можно объединить в один список изменений и выполнить одновременно. Например, можно добавить несколько колонок и/или изменить тип колонок одной командой.

Если команда записана в виде ALTER FOREIGN TABLE IF EXISTS ... и сторонняя таблица не существует, это не считается ошибкой. В этом случае выдаётся только замечание.

Выполнить ALTER FOREIGN TABLE может только владелец соответствующей таблицы. Чтобы сменить схему сторонней таблицы, необходимо также иметь право CREATE в новой схеме. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право CREATE в схеме таблицы. С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать таблицу. Однако суперпользователь может сменить владельца таблицы в любом случае. Чтобы добавить колонку или изменить тип колонки, ещё требуется иметь право USAGE для её типа данных.

2.11.3. Параметры

ИМЯ

Имя (возможно, дополненное схемой) существующей сторонней таблицы, подлежащей изменению. Если перед именем таблицы указано ONLY, изменяется только заданная таблица. Без ONLY изменяется и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно также добавить необязательное указание *, чтобы явно обозначить, что изменению подлежат все дочерние таблицы.

ИМЯ_СТОЛБЦА

Имя нового или существующего столбца.

НОВОЕ_ИМЯ_СТОЛБЦА

Новое имя существующего столбца.

НОВОЕ_ИМЯ

Новое имя таблицы.

тип_данных

Тип данных нового столбца или новый тип данных существующего столбца.

ограничение_таблицы

Новое ограничение уровня таблицы для сторонней таблицы.

имя_ограничения

Имя существующего ограничения, подлежащего удалению.

CASCADE

Автоматически удалять объекты, зависящие от удаляемого столбца или ограничения (например, представления, содержащие этот столбец), и, в свою очередь, все зависящие от них объекты.

RESTRICT

Отказать в удалении столбца или ограничения, если существуют зависящие от них объекты. Это поведение по умолчанию.

имя_триггера

Имя включаемого или отключаемого триггера.

ALL

Отключает или включает все триггеры, принадлежащие сторонней таблице. Если какие-либо из триггеров являются внутрисистемными, для этого требуются права суперпользователя. Сама система не добавляет такие триггеры в сторонние таблицы, но дополнительный код может сделать это.

USER

Отключает или включает все триггеры, принадлежащие сторонней таблице, кроме сгенерированных внутрисистемных.

таблица_родитель

Родительская таблица, с которой будет установлена или разорвана связь данной сторонней таблицы.

новый_владелец

Имя пользователя, назначаемого новым владельцем таблицы.

новая_схема

Имя схемы, в которую будет перемещена таблица.

2.11.4. Замечания

Ключевое слово COLUMN не несет смысловой нагрузки и может быть опущено.

При добавлении или удалении колонок (ADD COLUMN/DROP COLUMN), добавлении ограничений NOT NULL или изменении типа колонки (SET DATA TYPE) согласованность этих определений с внешним сервером не гарантируется. Ответственность за соответствие определений таблицы удаленной стороне лежит на пользователе.

2.11.5. Примеры

Установление ограничения NOT NULL для колонки:

```
ALTER FOREIGN TABLE distributors ALTER COLUMN street SET NOT NULL;
```

Изменение параметров сторонней таблицы:

```
ALTER FOREIGN TABLE myschema.distributors OPTIONS (ADD opt1 'value', SET opt2, 'value2', DROP opt3 'value3');
```

2.11.6. Совместимость

Формы ADD, DROP и SET DATA TYPE соответствуют стандарту SQL. Другие формы являются собственными расширениями Синергия-БД. Кроме того, возможность указать в одной команде ALTER FOREIGN TABLE несколько операций так же является расширением.

ALTER FOREIGN TABLE DROP COLUMN позволяет удалить единственную колонку сторонней таблицы и оставить таблицу без колонок. Это является расширением стандарта SQL, который не допускает существование сторонних таблиц с нулем колонок.

2.12.ALTER FUNCTION

ALTER FUNCTION – изменить определение функции.

2.12.1. Синтаксис

```
ALTER FUNCTION имя [ ( [ [ режим_аргумента ]  
[ имя_аргумента ] тип_аргумента [, ...] ] ) ]  
действие [ ... ] [ RESTRICT ]
```

```

ALTER FUNCTION имя [ ( [ [ режим_аргумента ]
[ имя_аргумента ] тип_аргумента [, ...] ] ) ]
    RENAME TO новое_имя
ALTER FUNCTION имя [ ( [ [ режим_аргумента ]
[ имя_аргумента ] тип_аргумента [, ...] ] ) ]
    OWNER TO { новый_владелец | CURRENT_USER |
SESSION_USER }
ALTER FUNCTION имя [ ( [ [ режим_аргумента ]
[ имя_аргумента ] тип_аргумента [, ...] ] ) ]
    SET SCHEMA новая_схема
ALTER FUNCTION имя [ ( [ [ режим_аргумента ]
[ имя_аргумента ] тип_аргумента [, ...] ] ) ]
    DEPENDS ON EXTENSION имя_расширения

```

Где возможные варианты action:

```

    CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT |
    STRICT
    IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ]
    SECURITY DEFINER
    PARALLEL { UNSAFE | RESTRICTED | SAFE }
    COST стоимость_выполнения
    ROWS строк_в_результате
    SET параметр_конфигурации { TO | = } { значение |
    DEFAULT }
    SET параметр_конфигурации FROM CURRENT
    RESET параметр_конфигурации
    RESET ALL

```

2.12.2. Описание

ALTER FUNCTION изменяет определение функции.

Выполнить ALTER FUNCTION может только владелец соответствующей функции. Чтобы сменить схему функции, необходимо также иметь право CREATE в новой схеме. Чтобы сменить владельца, требуется также быть непосредственным или опосредованным членом новой роли, а эта роль должна иметь право CREATE в схеме функции. С такими ограничениями при смене владельца не происходит ничего

такого, что нельзя было бы сделать, имея право удалить и вновь создать функцию. Однако суперпользователь может сменить владельца функции в любом случае.

2.12.3. Параметры

имя

Имя (возможно, дополненное схемой) существующей функции.

режим_аргумента

Режим аргумента: IN, OUT, INOUT или VARIADIC. По умолчанию подразумевается IN. Заметьте, что ALTER FUNCTION на самом деле не учитывает аргументы OUT, так как для идентификации функции нужны только типы входных аргументов. Поэтому достаточно перечислить только аргументы IN, INOUT и VARIADIC.

имя_аргумента

Имя аргумента. Заметьте, что на самом деле ALTER FUNCTION не обращает внимание на имена аргументов, так как для однозначной идентификации функции достаточно только типов аргументов.

тип_аргумента

Тип данных аргументов функции (возможно, дополненный именем схемы), если таковые имеются.

новое_имя

Новое имя функции.

новый_владелец

Новый владелец функции. Заметьте, что если функция помечена как SECURITY DEFINER, в дальнейшем она будет выполняться от имени нового владельца.

новая_схема

Новая схема функции.

имя_расширения

Имя расширения, от которого будет зависеть функция.

CALLED ON NULL INPUT

RETURNS NULL ON NULL INPUT

STRICT

CALLED ON NULL INPUT меняет функцию так, чтобы она вызывалась, когда некоторые или все её аргументы равны NULL. RETURNS NULL ON NULL INPUT или STRICT меняет функцию так, чтобы она не вызывалась, когда некоторые или все её аргументы равны NULL, а вместо вызова автоматически выдавался результат NULL. За подробностями обратитесь к CREATE FUNCTION.

IMMUTABLE
STABLE
VOLATILE

Устанавливает заданный вариант изменчивости функции. Подробнее это описано в CREATE FUNCTION.

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

Устанавливает, является ли функция определяющей контекст безопасности или нет. Ключевое слово EXTERNAL игнорируется для соответствия стандарту SQL. Подробнее это свойство описано в CREATE FUNCTION.

PARALLEL

Устанавливает, будет ли функция считаться безопасной для распараллеливания. Подробнее это описано в CREATE FUNCTION.

LEAKPROOF

Устанавливает, является ли функция герметичной или нет. Подробнее это свойство описано в CREATE FUNCTION.

COST стоимость_выполнения

Изменяет ориентировочную стоимость выполнения функции. Подробнее это описывается в CREATE FUNCTION.

ROWS строк_в_результате

Изменяет ориентировочное число строк в результате функции, возвращающей множество.

параметр_конфигурации
значение

Добавляет или изменяет установку параметра конфигурации, выполняемую при вызове функции. Если задано значение DEFAULT или, что равнозначно, выполняется действие RESET, локальная настройка для функции удаляется и

функция выполняется со значением, установленным в окружении. Для удаления всех настроек для данной функции укажите `RESET ALL`. `SET FROM CURRENT` сохраняет в качестве значения параметра, которое будет применяться при вызове функции, текущее значение в активном сеансе.

За подробными сведениями об именах и значениях параметров обратитесь к `SET` и `RESTRICT`.

Игнорируется для соответствия стандарту SQL.

2.12.4. Примеры

Переименование функции `sqrt` для типа `integer` в `square_root`:

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

Смена владельца функции `sqrt` для типа `integer` на `joe`:

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

Смена схемы функции `sqrt` для типа `integer` на `maths`:

```
ALTER FUNCTION sqrt(integer) SET SCHEMA maths;
```

Обозначение функции `sqrt` для типа `integer` как зависимой от расширения `mathlib`:

```
ALTER FUNCTION sqrt(integer) DEPENDS ON EXTENSION  
mathlib;
```

Изменение пути поиска, чтобы он устанавливался автоматически при вызове функции:

```
ALTER FUNCTION check_password(text) SET search_path =  
admin, pg_temp;
```

Отмена автоматической настройки `search_path` для функции:

```
ALTER FUNCTION check_password(text) RESET search_path;
```

Теперь функция будет выполняться с тем путем, который определен в момент вызова.

2.12.5. Совместимость

Этот оператор частично совместим с оператором `ALTER FUNCTION` в стандарте SQL. Стандарт позволяет менять больше свойств функции, но не позволяет переименовывать функции, переключать контекст безопасности,

связывать с функциями значения параметров конфигурации, а также менять владельца, схему и тип изменчивости функции. Также в стандарте слово RESTRICT считается обязательным, тогда как в Синергия-БД оно не требуется.

2.13.ALTER GROUP

ALTER GROUP – изменить имя роли или членство.

2.13.1. Синтаксис

```
ALTER GROUP указание_роли ADD USER имя_пользователя  
[ , ... ]
```

```
ALTER GROUP указание_роли DROP USER имя_пользователя  
[ , ... ]
```

Здесь указание_роли:

```
    имя_роли  
| CURRENT_USER  
| SESSION_USER
```

2.13.2. Описание

ALTER GROUP изменяет атрибуты группы пользователей. Эта команда считается устаревшей, хотя и поддерживается для обратной совместимости, так как группы (и пользователи) были заменены более общей концепцией ролей.

Первые две формы добавляют пользователей в группу или удаляют их из группы. В данном случае в качестве «пользователя» или «группы» может фигурировать любая роль. По сути они равнозначны командам разрешающим/запрещающим членство в роли «группа»; поэтому вместо них рекомендуется использовать GRANT и REVOKE.

Третья форма меняет имя группы. Она в точности равнозначна команде ALTER PROCEDURE.

2.14.ALTER INDEX

ALTER INDEX – изменить определение индекса.

2.14.1. Синтаксис

```
ALTER INDEX [ IF EXISTS ] имя RENAME TO новое_имя
ALTER INDEX [ IF EXISTS ] имя SET TABLESPACE
табл_пространство
ALTER INDEX имя ATTACH PARTITION имя_индекса
ALTER INDEX имя DEPENDS ON EXTENSION имя_расширения
ALTER INDEX [ IF EXISTS ] имя SET ( параметр_хранения =
значение [, ... ] )
ALTER INDEX [ IF EXISTS ] имя RESET ( параметр_хранения
[, ... ] )
ALTER INDEX [ IF EXISTS ] имя ALTER [ COLUMN ]
номер_столбца
        SET STATISTICS целое
ALTER INDEX ALL IN TABLESPACE имя [ OWNED BY имя_роли
[, ... ] ]
        SET TABLESPACE новое_табл_пространство [ NOWAIT ]
```

2.14.2. Описание

ALTER INDEX меняет определение существующего индекса. Эта команда имеет несколько разновидностей:

- RENAME

Форма RENAME меняет имя индекса. Если этот индекс связан с ограничением таблицы (UNIQUE, PRIMARY KEY или EXCLUDE), это ограничение тоже переименовывается. На сохранённые данные это не влияет;

- SET TABLESPACE

Эта форма меняет табличное пространство индекса на заданное и переносит в него файл(ы) данных, связанные с индексом. Для изменения табличного пространства индекса нужно быть владельцем индекса и иметь право CREATE в новом табличном пространстве. Форма ALL IN TABLESPACE позволяет перенести из заданного пространства все индексы в текущей базе данных, блокируя их для перемещения и затем перемещая каждый индекс. Эта форма также поддерживает указание OWNED BY, с которым будут перемещены только индексы, принадлежащие заданным ролям. Если указан параметр NOWAIT, команда завершится ошибкой, если не сможет немедленно получить

все требуемые блокировки. Заметьте, что эта команда не переместит системные каталоги; вместо нее следует использовать ALTER DATABASE или явные вызовы ALTER INDEX;

- ATTACH PARTITION

Эта форма присоединяет указанный индекс к изменяемому. Указанный индекс должен относиться к секции таблицы, содержащей изменяемый индекс, и иметь такое же определение. Присоединённый индекс не может быть удалён независимо, но будет удалён автоматически при удалении родительского индекса;

- DEPENDS ON EXTENSION

Эта форма помечает индекс как зависимый от расширения, так что при удалении расширения будет автоматически удалён и индекс;

- SET (параметр_хранения = значение [, ...])

Эта форма настраивает один или несколько специфичных для индекса параметров хранения. Список доступных параметров приведен в CREATE INDEX. Заметьте, что эта команда не меняет содержимое индекса немедленно; для получения желаемого эффекта в зависимости от параметров может потребоваться перестроить индекс командой REINDEX;

- RESET (параметр_хранения [, ...])

Эта форма сбрасывает один или несколько специфичных для индекса параметров хранения к значениям по умолчанию. Как и с SET, для полного обновления индекса может потребоваться выполнить REINDEX;

- ALTER [COLUMN] номер_столбца SET STATISTICS целое

Эта форма задаёт ориентир сбора статистики по столбцу для последующих операций ANALYZE, хотя её можно использовать только для индексируемых столбцов, заданных в виде выражений. Так как у выражений нет уникальных имён, мы обращаемся к ним по порядковым номерам столбцов в индексе. Диапазон допустимых значений ориентира: 0..10000; при -1 применяется системное значение по умолчанию (default_statistics_target).

2.14.3. Параметры

IF EXISTS

Не считать ошибкой, если индекс не существует. В этом случае будет выдано замечание.

номер_столбца

Число, указывающее на номер столбца в индексе по порядку (слева направо).

имя

Имя (возможно, дополненное схемой) существующего индекса, подлежащего изменению.

новое_имя

Новое имя индекса.

табл_пространство

Табличное пространство, в которое будет перемещен индекс.

имя_расширения

Имя расширения, от которого будет зависеть индекс.

параметр_хранения

Имя специфичного для индекса параметра хранения.

значение

Новое значение специфичного для индекса параметра хранения. Это может быть число или строка, в зависимости от параметра.

2.14.4. Замечания

Эти операции также возможно выполнить с помощью `ALTER TABLE`. На самом деле `ALTER INDEX` – это просто синоним нескольких форм `ALTER TABLE`, работающих с индексами.

Ранее существовала форма `ALTER INDEX OWNER`, но сейчас она игнорируется (с предупреждением). Владелец индекса может быть только владелец соответствующей таблицы. При смене владельца таблицы владелец индекса меняется автоматически.

Какие-либо изменения индексов системного каталога не допускаются.

2.14.5. Примеры

Переименование существующего индекса:

```
ALTER INDEX distributors RENAME TO suppliers;
```

Перемещение индекса в другое табличное пространство:

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

Изменение фактора заполнения индекса (предполагается, что это поддерживает метод индекса):

```
ALTER INDEX distributors SET (fillfactor = 75);  
REINDEX INDEX distributors;
```

Устанавливает ориентир сбора статистики для индекса по выражению:

```
CREATE INDEX coord_idx ON measured (x, y, (z + t));  
ALTER INDEX coord_idx ALTER COLUMN 3 SET STATISTICS  
1000;
```

2.14.6. Совместимость

ALTER INDEX является расширением Синергия-БД.

2.15.ALTER LANGUAGE

ALTER LANGUAGE – изменить определение процедурного языка.

2.15.1. Синтаксис

```
ALTER [ PROCEDURAL ] LANGUAGE имя RENAME TO новое_имя  
ALTER [ PROCEDURAL ] LANGUAGE имя OWNER TO  
{ новый_владелец | CURRENT_USER | SESSION_USER }
```

2.15.2. Описание

ALTER LANGUAGE изменяет определение процедурного языка. Единственное, что может это команда – переименовать язык или назначить нового владельца. Выполнить ALTER LANGUAGE может только суперпользователь или владелец языка.

2.15.3. Параметры

имя

Имя языка.

новое_имя

Новое имя языка.

новый_владелец

Новый владелец языка.

2.15.4. Совместимость

Оператор ALTER LANGUAGE отсутствует в стандарте SQL.

2.16.ALTER LARGE OBJECT

ALTER LARGE OBJECT – изменить определение большого объекта.

2.16.1. Синтаксис

```
ALTER LARGE OBJECT oid_большого_объекта OWNER TO  
новый_владелец
```

2.16.2. Описание

ALTER LARGE OBJECT изменяет определение большого объекта. Выполнить ALTER LARGE OBJECT может только владелец большого объекта. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца. Однако суперпользователь может изменять свойства больших объектов в любом случае. В настоящее время единственное возможное изменение заключается в назначении нового владельца, так что всегда действуют оба ограничения.

2.16.3. Параметры

oid_большого_объекта
OID изменяемого большого объекта.
новый_владелец
Новый владелец большого объекта.

2.16.4. Совместимость

Оператор ALTER LARGE OBJECT отсутствует в стандарте SQL.

2.17.ALTER MATERIALIZED VIEW

ALTER MATERIALIZED VIEW – изменить определение материализованного представления.

2.17.1. Синтаксис

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] имя  
    действие [, ... ]
```

```
ALTER MATERIALIZED VIEW имя
```

```
    DEPENDS ON EXTENSION имя_расширения
```

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] имя
```

```
    RENAME [ COLUMN ] имя_столбца TO новое_имя_столбца
```

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] имя
```

```
    RENAME TO новое_имя
```

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] имя
```

```
    SET SCHEMA новая_схема
```

```
ALTER MATERIALIZED VIEW ALL IN TABLESPACE имя [ OWNED BY  
имя_роли [, ... ] ]
```

```
    SET TABLESPACE новое_табл_пространство [ NOWAIT ]
```

Где действие может быть следующим:

```
    ALTER [ COLUMN ] имя_столбца SET STATISTICS integer
```

```
    ALTER [ COLUMN ] имя_столбца SET ( атрибут =  
значение [, ... ] )
```

```
    ALTER [ COLUMN ] имя_столбца RESET ( атрибут  
[, ... ] )
```

```
    ALTER [ COLUMN ] имя_столбца SET STORAGE { PLAIN |  
EXTERNAL | EXTENDED | MAIN }
```

```
    CLUSTER ON имя_индекса
```

```
    SET WITHOUT CLUSTER
```

```
    SET ( параметр_хранения = значение [, ... ] )
```

```
    RESET ( параметр_хранения [, ... ] )
```

```
    OWNER TO { новый_владелец | CURRENT_USER |  
SESSION_USER }
```

2.17.2. Описание

ALTER MATERIALIZED VIEW изменяет различные расширенные свойства существующего материализованного представления.

Выполнить ALTER MATERIALIZED VIEW может только владелец материализованного представления. Чтобы сменить схему материализованного представления, необходимо также иметь право CREATE в новой схеме. Чтобы

сменить владельца, требуется также быть непосредственным или опосредованным членом новой роли, а эта роль должна иметь право CREATE в схеме материализованного представления. С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать материализованное представление. Однако суперпользователь может сменить владельца материализованного представления в любом случае.

Форма `DEPENDS ON EXTENSION` помечает материализованное представление как зависимое от расширения, так что представление будет автоматически удаляться при удалении расширения.

Подвиды и действия оператора `ALTER MATERIALIZED VIEW` являются подмножеством тех, что относятся к команде `ALTER TABLE`, и имеют то же значение применительно к материализованным представлениям.

2.17.3. Параметры

`имя`

Имя (возможно, дополненное схемой) существующего материализованного представления.

`имя_колонки`

Имя новой или существующей колонки.

`новое_имя_колонки`

Новое имя существующей колонки.

`имя_расширения`

Имя расширения, от которого будет зависеть материализованное представление.

`новый_владелец`

Имя пользователя, назначаемого новым владельцем материализованного представления.

`новое_имя`

Новое имя материализованного представления.

`новая_схема`

Новая схема материализованного представления.

2.17.4. Примеры

Переименование материализованного представления foo в bar:

```
ALTER MATERIALIZED VIEW foo RENAME TO bar;
```

2.17.5. Совместимость

ALTER MATERIALIZED VIEW является расширением Синергия-БД.

2.18.ALTER OPERATOR

ALTER OPERATOR – изменить определение оператора

2.18.1. Синтаксис

```
ALTER OPERATOR имя ( { тип_слева | NONE } , { тип_справа  
| NONE } )
```

```
    OWNER TO { новый_владелец | CURRENT_USER |  
SESSION_USER }
```

```
ALTER OPERATOR имя ( { тип_слева | NONE } , { тип_справа  
| NONE } )
```

```
    SET SCHEMA новая_схема
```

```
ALTER OPERATOR имя ( { тип_слева | NONE } , { тип_справа  
| NONE } )
```

```
    SET ( { RESTRICT = { процедура_ограничения | NONE }  
          | JOIN = { процедура_соединения | NONE }  
          } [, ... ] )
```

2.18.2. Описание

ALTER OPERATOR изменяет определение оператора.

Выполнить ALTER OPERATOR может только владелец соответствующего оператора. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право CREATE в схеме оператора. С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать оператор. Однако суперпользователь может сменить владельца оператора в

любом случае.

2.18.3. Параметры

имя

Имя (возможно, дополненное схемой) существующего оператора.

тип_слева

Тип данных левого операнда оператора; если у оператора нет левого операнда, укажите NONE.

тип_справа

Тип данных правого операнда оператора; если у оператора нет правого операнда, укажите NONE.

новый_владелец

Новый владелец оператора.

новая_схема

Новая схема оператора.

процедура_ограничения

Функция оценки избирательности ограничения для данного оператора; значение NONE удаляет существующую функцию оценки.

процедура_соединения

Функция оценки избирательности соединения для этого оператора; значение NONE удаляет существующую функцию оценки.

2.18.4. Примеры

Смена владельца нестандартного оператора a @@ b для типа text:

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

Смена функций оценки избирательности ограничения и соединения для нестандартного оператора a && b для типа int[]:

```
ALTER OPERATOR && (_int4, _int4) SET (RESTRICT =  
_int_contsel, JOIN = _int_contjoinset);
```

2.18.5. Совместимость

Команда ALTER OPERATOR отсутствует в стандарте SQL.

2.19.ALTER OPERATOR CLASS

ALTER OPERATOR CLASS – изменить определение класса операторов.

2.19.1. Синтаксис

```
ALTER OPERATOR CLASS имя USING метод_индекса  
    RENAME TO новое_имя
```

```
ALTER OPERATOR CLASS имя USING метод_индекса  
    OWNER TO { новый_владелец | CURRENT_USER |  
SESSION_USER }
```

```
ALTER OPERATOR CLASS имя USING метод_индекса  
    SET SCHEMA новая_схема
```

2.19.2. Описание

ALTER OPERATOR CLASS изменяет определение класса операторов.

Выполнить ALTER OPERATOR CLASS может только владелец соответствующего класса операторов. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право CREATE в схеме класса операторов. С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать класс операторов. Однако суперпользователь может сменить владельца классов операторов в любом случае.

2.19.3. Параметры

имя

Имя (возможно, дополненное схемой) существующего класса операторов.

метод_индекса

Имя индексного метода, для которого предназначен этот класс операторов.

новое_имя

Новое имя класса операторов.

новый_владелец

Новый владелец класса операторов.

новая_схема

Новая схема класса операторов.

2.19.4. Совместимость

Команда ALTER OPERATOR CLASS отсутствует в стандарте SQL.

2.20.ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY – изменить определение семейства операторов.

2.20.1. Синтаксис

```
ALTER OPERATOR FAMILY имя USING метод_индекса ADD
    { OPERATOR номер_стратегии имя_оператора
    ( тип_операнда, тип_операнда )
        [ FOR SEARCH | FOR ORDER BY
семейство_сортировки ]
        | FUNCTION номер_опорной_функции [ ( тип_операнда [ ,
тип_операнда ] ) ]
            имя_функции [ ( тип_аргумента [, ...] ) ]
    } [, ... ]
```

```
ALTER OPERATOR FAMILY имя USING метод_индекса DROP
    { OPERATOR номер_стратегии ( тип_операнда [ ,
тип_операнда ] )
        | FUNCTION номер_опорной_функции ( тип_операнда [ ,
тип_операнда ] )
    } [, ... ]
```

```
ALTER OPERATOR FAMILY имя USING метод_индекса
    RENAME TO новое_имя
```

```
ALTER OPERATOR FAMILY имя USING метод_индекса
    OWNER TO { новый_владелец | CURRENT_USER |
SESSION_USER }
```

```
ALTER OPERATOR FAMILY имя USING метод_индекса
```

SET SCHEMA новая_схема

2.20.2. Описание

ALTER OPERATOR FAMILY меняет определение семейства операторов. Она позволяет добавлять в семейство операторы и опорные функции, удалять их из семейства или менять имя и владельца семейства операторов.

Когда операторы и опорные функции добавляются в семейство с помощью ALTER OPERATOR FAMILY, они не становятся частью какого-либо определенного класса операторов в семействе, а просто считаются «слабосвязанными» с семейством. Это показывает, что эти операторы и функции семантически совместимы с семейством, но не требуются для корректной работы какого-либо индекса. Операторы и функции, которые действительно требуются для этого, должны быть включены не в семейство, а в класс операторов. Синергия-БД позволяет удалять слабосвязанные члены из семейства в любое время, но члены класса операторов не могут быть удалены, пока не будет удален весь класс и все зависимые от него индексы. Обычно в классы операторов включаются операторы и функции, работающие с одним типом данным (так как они нужны для поддержки индексов данных такого типа), а функции и операторы, работающие с разными типами, становятся слабосвязанными членами семейства.

Выполнить ALTER OPERATOR FAMILY может только суперпользователь. Это ограничение введено потому, что ошибочное определение семейства операторов может вызвать нарушения или даже сбой в работе сервера.

ALTER OPERATOR FAMILY в настоящее время не проверяет, включает ли определение семейства операторов все операторы и функции, требуемые для индексного метода, и образуют ли они целостный набор. Ответственность за правильность определения семейства лежит на пользователе.

2.20.3. Параметры

имя

Имя (возможно, дополненное схемой) существующего семейства операторов.

метод_индекса

Имя индексного метода, для которого предназначено это семейство

операторов.

номер_стратегии

Номер стратегии индексного метода для оператора, связанного с данным семейством операторов.

имя_оператора

Имя (возможно, дополненное схемой) оператора, связанного с данным семейством операторов.

тип_операнда

В предложении `OPERATOR` указывается тип(ы) данных оператора или `NONE`, если это левый или правый унарный оператор. В отличие от похожего синтаксиса в `CREATE OPERATOR CLASS`, здесь типы операндов должны указываться всегда.

В предложении `ADD FUNCTION` это тип данных, который должна поддерживать эта функция, если он отличается от входного типа данных функции. Для функций сравнения B-деревьев и хеш-функций указывать тип_операнда необязательно, так как их входные типы данных всегда будут верными. Однако для функций поддержки сортировки B-деревьев и всех функций в классах операторов `GiST`, `SP-GiST` и `GIN` необходимо указать тип(ы) операндов, с которыми будут использоваться эти функции.

В предложении `DROP FUNCTION` тип операнда, который должна поддерживать эта функция.

семейство_сортировки

Имя (возможно, дополненное схемой) существующего семейства операторов `btree`, описывающего порядок сортировки, связанный с оператором сортировки.

Если не указано ни `FOR SEARCH` (для поиска), ни `FOR ORDER BY` (для сортировки), подразумевается `FOR SEARCH`.

номер_опорной_функции

Номер опорной процедуры индексного метода для функции, связанной с данным семейством операторов.

имя_функции

Имя (возможно, дополненное схемой) функции, которая является опорной процедурой индексного метода для данного семейства операторов. Если список

аргументов отсутствует, имя функции должно быть уникальным в её схеме.

тип_аргумента

Тип данных параметра функции.

новое_имя

Новое имя семейства операторов.

новый_владелец

Новый владелец семейства операторов.

новая_схема

Новая схема семейства операторов.

Предложения OPERATOR и FUNCTION могут указываться в любом порядке.

2.20.4. Замечания

Заметьте, что в синтаксисе DROP указывается только «слот» в семействе операторов, по номеру стратегии или опорной функции, и входные типы данных. Имя оператора или функции, занимающих этот слот, не упоминается. Также учтите, что в DROP FUNCTION указываются типы входных данных, которые должна поддерживать функция, но для индексов GiST, SP-GiST и GIN они могут не иметь ничего общего с типами фактических аргументов функции.

Так как механизмы индексов не проверяют права доступа к функциям прежде чем вызывать их, включение функций или операторов в семейство операторов по сути дает всем право на выполнение их. Обычно это не проблема для таких функций, какие бывают полезны в семействе операторов.

Операторы не должны реализовываться в функциях на языке SQL. SQL-функция вероятнее всего будет встроена в вызывающий запрос, что помешает оптимизатору понять, что этот запрос соответствует индексу.

2.20.5. Примеры

Следующий пример добавляет опорные функции и операторы смешанных типов в семейство операторов, уже содержащее классы операторов B-дерева для типов данных int4 и int2.

```
ALTER OPERATOR FAMILY integer_ops USING btree ADD
```

```
-- int4 и int2
OPERATOR 1 < (int4, int2) ,
OPERATOR 2 <= (int4, int2) ,
OPERATOR 3 = (int4, int2) ,
OPERATOR 4 >= (int4, int2) ,
OPERATOR 5 > (int4, int2) ,
FUNCTION 1 btint42cmp(int4, int2) ,
```

```
-- int2 и int4
OPERATOR 1 < (int2, int4) ,
OPERATOR 2 <= (int2, int4) ,
OPERATOR 3 = (int2, int4) ,
OPERATOR 4 >= (int2, int4) ,
OPERATOR 5 > (int2, int4) ,
FUNCTION 1 btint24cmp(int2, int4) ;
```

Удаление этих же элементов:

```
ALTER OPERATOR FAMILY integer_ops USING btree DROP
```

```
-- int4 vs int2
OPERATOR 1 (int4, int2) ,
OPERATOR 2 (int4, int2) ,
OPERATOR 3 (int4, int2) ,
OPERATOR 4 (int4, int2) ,
OPERATOR 5 (int4, int2) ,
FUNCTION 1 (int4, int2) ,
```

```
-- int2 vs int4
OPERATOR 1 (int2, int4) ,
OPERATOR 2 (int2, int4) ,
OPERATOR 3 (int2, int4) ,
OPERATOR 4 (int2, int4) ,
OPERATOR 5 (int2, int4) ,
FUNCTION 1 (int2, int4) ;
```

2.20.6. Совместимость

Команда ALTER OPERATOR FAMILY отсутствует в стандарте SQL.

2.21.ALTER POLICY

ALTER POLICY – изменить определение политики защиты на уровне строк.

2.21.1. Синтаксис

```
ALTER POLICY имя ON имя_таблицы RENAME TO новое_имя
```

```
ALTER POLICY имя ON имя_таблицы  
    [ TO { имя_роли | PUBLIC | CURRENT_USER |  
SESSION_USER } [, ...] ]  
    [ USING ( выражение_использования ) ]  
    [ WITH CHECK ( выражение_проверки ) ]
```

2.21.2. Описание

ALTER POLICY изменяет определение существующей политики на уровне строк. Заметьте, что ALTER POLICY позволяет изменить только набор ролей, для которых применяется политика, и выражения USING и WITH CHECK. Чтобы изменить другие свойства политики, например команду, к которой она применяется, а также характеристику разрешительная/ограничительная, политику надо удалить и создать заново.

Использовать ALTER POLICY может только владелец таблицы (или представления), к которой применяется эта политика. Во второй форме ALTER POLICY список ролей, выражение_использования и выражение_проверки заменяются независимо, если они указаны. Когда одно из этих предложений опущено, соответствующая часть политики остаётся неизменной.

2.21.3. Параметры

имя

Имя существующей политики, подлежащей изменению.

имя_таблицы

Имя (возможно, дополненное схемой) таблицы, к которой применяется эта

политика.

новое_имя

Новое имя политики.

имя_роли

Роль, на которую действует политика. В одной команде можно указать несколько ролей. Чтобы применить политику ко всем ролям, укажите PUBLIC (этот вариант также подразумевается по умолчанию).

выражение_использования

Выражение USING для политики.

выражение_проверки

Выражение WITH CHECK для политики.

2.21.4. Совместимость

ALTER POLICY является расширением Синергия-БД.

2.22.ALTER PROCEDURE

ALTER PROCEDURE – изменить определение процедуры.

2.22.1. Синтаксис

```
ALTER PROCEDURE имя [ ( [ [ режим_аргумента ]  
[ имя_аргумента ] тип_аргумента [, ...] ] ) ]  
действие [ ... ] [ RESTRICT ]
```

```
ALTER PROCEDURE имя [ ( [ [ режим_аргумента ]  
[ имя_аргумента ] тип_аргумента [, ...] ] ) ]  
RENAME TO новое_имя
```

```
ALTER PROCEDURE имя [ ( [ [ режим_аргумента ]  
[ имя_аргумента ] тип_аргумента [, ...] ] ) ]  
OWNER TO { новый_владелец | CURRENT_USER |  
SESSION_USER }
```

```
ALTER PROCEDURE имя [ ( [ [ режим_аргумента ]  
[ имя_аргумента ] тип_аргумента [, ...] ] ) ]  
SET SCHEMA новая_схема
```

```
ALTER PROCEDURE имя [ ( [ [ режим_аргумента ]  
[ имя_аргумента ] тип_аргумента [, ...] ] ) ]  
DEPENDS ON EXTENSION имя_расширения
```

Где действие может быть следующим:

```
[ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ]  
SECURITY DEFINER
```

```
SET параметр_конфигурации { TO | = } { значение |  
DEFAULT }
```

```
SET параметр_конфигурации FROM CURRENT
```

```
RESET параметр_конфигурации
```

```
RESET ALL
```

2.22.2. Описание

ALTER PROCEDURE изменяет определение процедуры.

Выполнить ALTER PROCEDURE может только владелец процедуры. Чтобы сменить схему процедуры, необходимо также иметь право CREATE в новой схеме. Чтобы сменить владельца, требуется также быть непосредственным или опосредованным членом новой роли, а эта роль должна иметь право CREATE в схеме представления. С таким ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать процедуру. Однако суперпользователь может сменить владельца процедуры в любом случае.

2.22.3. Параметры

имя

Имя существующей процедуры (возможно, дополненное схемой). Если список аргументов не указан, имя процедуры должно быть уникальным в её схеме.

режим_аргумента

Режим аргумента: IN или VARIADIC. По умолчанию подразумевается IN.

имя_аргумента

Имя аргумента. Заметьте, что на самом деле ALTER PROCEDURE не обращает внимание на имена аргументов, так как для однозначной идентификации процедуры достаточно только типов аргументов.

тип_аргумента

Тип данных аргументов процедуры (возможно, дополненный именем схемы), если таковые имеются.

новое_имя

Новое имя процедуры.

новый_владелец

Новый владелец процедуры. Заметьте, что если процедура помечена как SECURITY DEFINER, в дальнейшем она будет выполняться от имени нового владельца.

новая_схема

Новая схема процедуры.

имя_расширения

Имя расширения, от которого будет зависеть процедура.

[EXTERNAL] SECURITY INVOKER

[EXTERNAL] SECURITY DEFINER

Устанавливает, является ли процедура определяющей контекст безопасности.

Ключевое слово EXTERNAL игнорируется для соответствия стандарту SQL.

параметр_конфигурации

значение

Добавляет или изменяет установку параметра конфигурации, выполняемую при вызове процедуры. Если задано значение DEFAULT или, что равнозначно, выполняется действие RESET, локальное переопределение для процедуры удаляется и процедура выполняется со значением, установленным в окружении. Для удаления всех установок параметров для данной процедуры укажите RESET ALL. SET FROM CURRENT устанавливает для последующих вызовов процедуры значение параметра, действующее в момент выполнения ALTER PROCEDURE.

RESTRICT

Игнорируется для соответствия стандарту SQL.

2.22.4. Примеры

Переименование процедуры insert_data с двумя аргументами типа integer в insert_record:

```
ALTER PROCEDURE insert_data(integer, integer) RENAME TO  
insert_record;
```

Смена владельца процедуры insert_data с двумя аргументами типа

integer на joe:

```
ALTER PROCEDURE insert_data(integer, integer) OWNER TO  
joe;
```

Смена схемы процедуры insert_data с двумя аргументами типа integer на accounting:

```
ALTER PROCEDURE insert_data(integer, integer) SET SCHEMA  
accounting;
```

Обозначение процедуры insert_data(integer, integer) как зависимой от расширения myext:

```
ALTER PROCEDURE insert_data(integer, integer) DEPENDS ON  
EXTENSION myext;
```

Изменение пути поиска, который устанавливается автоматически для процедуры:

```
ALTER PROCEDURE check_password(text) SET search_path =  
admin, pg_temp;
```

Отмена автоматического определения search_path для процедуры:

```
ALTER PROCEDURE check_password(text) RESET search_path;
```

Теперь процедура будет выполняться с тем путём, который задан в момент вызова.

2.22.5. Совместимость

Этот оператор частично совместим с оператором ALTER PROCEDURE в стандарте SQL. Стандарт позволяет изменить больше свойств процедуры, но не даёт возможности переименовать процедуру, сделать процедуру определяющей контекст безопасности, связать с процедурой значения параметров конфигурации или изменить владельца, схему или характеристику изменчивости процедуры. Также стандарт требует наличия ключевого слова RESTRICT, но в Синергия-БД оно необязательное.

2.23.ALTER PUBLICATION

ALTER PUBLICATION – изменить определение публикации.

2.23.1. Синтаксис

```
ALTER PUBLICATION имя ADD TABLE [ ONLY ] имя_таблицы  
[ * ] [, ...]  
ALTER PUBLICATION имя SET TABLE [ ONLY ] имя_таблицы  
[ * ] [, ...]  
ALTER PUBLICATION имя DROP TABLE [ ONLY ] имя_таблицы  
[ * ] [, ...]  
ALTER PUBLICATION имя SET ( параметр_публикации [= значение] [, ... ] )  
ALTER PUBLICATION имя OWNER TO { новый_владелец |  
CURRENT_USER | SESSION_USER }  
ALTER PUBLICATION имя RENAME TO новое_имя
```

2.23.2. Описание

Команда ALTER PUBLICATION может изменять атрибуты публикации.

Первые три формы управляют вхождением таблиц в публикации. Предложение SET TABLE заменяет список таблиц в публикации заданным. Предложения ADD TABLE и DROP TABLE добавляют и удаляют таблицы в публикации, соответственно. Заметьте, что при добавлении таблиц в публикацию, на которую уже оформлена подписка, необходимо выполнить ALTER SUBSCRIPTION ... REFRESH PUBLICATION на стороне подписчика, чтобы это изменение вступило в силу.

Четвёртая форма этой команды, показанная в сводке синтаксиса, может изменять все свойства публикации, заданные в CREATE PUBLICATION. Свойства, которые не упоминаются в этой команде, сохраняют предыдущие значения.

Остальные формы команды меняют владельца и имя публикации.

Для выполнения ALTER PUBLICATION необходимо владеть данной публикацией. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право CREATE в базе данных. Кроме того, новым владельцем публикации FOR ALL TABLES должен быть суперпользователь. Однако суперпользователь может менять владение публикацией, обходя эти ограничения.

2.23.3. Параметры

ИМЯ

Имя существующей публикации, определение которой изменяется.

ИМЯ_таблицы

Имя существующей таблицы. Если перед именем таблицы указано ONLY, затрагивается только заданная таблица. Без ONLY затрагивается и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно добавить необязательное указание *, чтобы явно обозначить, что должны затрагиваться и все дочерние таблицы.

```
SET ( параметр_публикации [= значение] [, ... ] )
```

Это предложение изменяет параметры публикации, изначально установленные командой CREATE PUBLICATION. За дополнительными сведениями обратитесь к её описанию.

новый_владелец

Имя пользователя, назначаемого новым владельцем публикации.

новое_имя

Новое имя публикации.

2.23.4. Примеры

Изменение публикации, чтобы публиковались только удаления и изменения:

```
ALTER PUBLICATION noinsert SET (publish = 'update,  
delete');
```

Добавление таблиц в публикацию:

```
ALTER PUBLICATION mypublication ADD TABLE users,  
departments;
```

2.23.5. Совместимость

ALTER PUBLICATION является расширением Синергия-БД.

2.24.ALTER ROLE

ALTER ROLE – изменить роль в базе данных.

2.24.1. Синтаксис

ALTER ROLE имя [[WITH] параметр [...]]

Здесь параметр:

SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| BYPASSRLS | NOBYPASSRLS
| CONNECTION LIMIT предел_подключений
| [ENCRYPTED] PASSWORD 'пароль' | PASSWORD NULL
| VALID UNTIL 'дата_время'

ALTER ROLE имя RENAME TO новое_имя

ALTER ROLE { указание_роли | ALL } [IN DATABASE
имя_бд] SET параметр_конфигурации { TO | = } { значение
| DEFAULT }

ALTER ROLE { указание_роли | ALL } [IN DATABASE
имя_бд] SET параметр_конфигурации FROM CURRENT

ALTER ROLE { указание_роли | ALL } [IN DATABASE
имя_бд] RESET параметр_конфигурации

ALTER ROLE { указание_роли | ALL } [IN DATABASE
имя_бд] RESET ALL

Здесь указание_роли:

имя_роли
| CURRENT_USER
| SESSION_USER

2.24.2. Описание

ALTER ROLE изменяет атрибуты роли Синергия-БД.

Первая форма команды в этой справке может изменить многие атрибуты роли, которые можно указать в CREATE ROLE. Покрываются все возможные атрибуты, отсутствуют только возможности добавления/удаления членов роли; для этого

нужно использовать GRANT и REVOKE. Атрибуты, не упомянутые в команде, сохраняют свои предыдущие значения. Суперпользователи базы данных могут изменить любые параметры любой роли, а пользователи с правом CREATEROLE могут также менять любые параметры, но только не ролей суперпользователей и репликации. Обычные пользователи (роли) могут менять только свой пароль.

Вторая форма меняет имя роли. Суперпользователи базы данных могут переименовать любую роль, а пользователи с правом CREATEROLE могут переименовывать роли не суперпользователей. Также нельзя переименовать роль текущего пользователя в активном сеансе. Если вам нужно сделать это, подключитесь другим пользователем. Так как в паролях с MD5-шифрованием имя роли используется в качестве криптосоли, при переименовании роли её пароль очищается, если он был зашифрован MD5.

Оставшиеся формы меняют значение по умолчанию конфигурационной переменной, которое будет распространяться на сеансы роли во всех базах данных, либо, если добавлено предложение IN DATABASE, только на сеансы роли в заданной базе. Если вместо имени роли указано ALL, это значение переменной распространяется на все роли. Использование ALL с IN DATABASE по сути равносильно использованию команды ALTER DATABASE ... SET

Когда эта роль впоследствии установит новое подключение, указанное значение станет значением по умолчанию в сеансе, переопределяя значение, заданное в postgresql.conf или полученное из командной строки postgres. Это происходит только в момент входа; при выполнении SET ROLE или SET SESSION AUTHORIZATION новые значения не применяются. Набор параметров для всех баз данных переопределяется параметрами уровня БД, установленными для роли. Параметры для конкретной базы данных или конкретной роли переопределяют параметры для всех ролей.

Суперпользователи могут менять значения переменных по умолчанию для любых ролей, а пользователи с правом CREATEROLE могут менять их только для ролей не суперпользователей. Обычные пользователи могут определять переменные только для себя. Некоторые переменные конфигурации нельзя задать таким

способом, а некоторые может настроить только суперпользователь. Параметры всех ролей во всех базах данных могут настраивать только суперпользователи.

2.24.3. Параметры

ИМЯ

Имя роли, атрибуты которой изменяются.

CURRENT_USER

Выбирает для изменения текущего пользователя, а не явно задаваемую роль.

SESSION_USER

Выбирает для изменения текущего пользователя сеанса, а не явно задаваемую роль.

SUPERUSER

NOSUPERUSER

CREATEDB

NOCREATEDB

CREATEROLE

NOCREATEROLE

INHERIT

NOINHERIT

LOGIN

NOLOGIN

REPLICATION

NOREPLICATION

BYPASSRLS

NOBYPASSRLS

CONNECTION LIMIT *предел_подключений*

[ENCRYPTED] PASSWORD '*пароль*'

PASSWORD NULL

VALID UNTIL '*дата_время*'

Эти предложения меняют атрибуты, изначально установленные командой CREATE ROLE.

новое_имя

Новое имя роли.

имя_бд

Имя базы данных, в которой устанавливается конфигурационная переменная.

параметр_конфигурации

значение

Указанный параметр конфигурации принимает заданное значение по

умолчанию в сеансах роли. Если значение задано как DEFAULT или, что то же самое, применяется операция RESET, настройка этого параметра, связанная с ролью, удаляется и роль будет получать в новых сеансах системное значение параметра. Для очистки значений всех параметров, связанных с ролью, применяется RESET ALL. SET FROM CURRENT сохраняет текущее значение параметра в активном сеансе в качестве настроенного значения для данной роли. Если указано IN DATABASE, параметр конфигурации настраивается или удаляется только для данной роли и указанной базы данных.

Настройки переменных для роли применяются только в начале сеанса; команды SET ROLE и SET SESSION AUTHORIZATION не обрабатывают настройки, связанные с ролью.

2.24.4. Замечания

Для добавления новых ролей используйте команду CREATE ROLE, а для удаления роли – DROP PROCEDURE.

ALTER ROLE не может управлять членством роли, для этого применяется GRANT и REVOKE.

Указывая в этой команде незашифрованный пароль, следует проявлять осторожность. Пароль будет передаваться на сервер открытым текстом и может также записаться в историю команд клиента или в протокол работы сервера. В psql есть команда \password, с помощью которой можно сменить пароль роли, не рискуя рассекретить пароль.

Также возможно связать сеансовые значения по умолчанию с определенной базой данных, а не с ролью. В случае конфликта параметры для базы данных и роли переопределяют параметры только для роли, которые, в свою очередь, переопределяют параметры для базы данных.

2.24.5. Примеры

Изменение пароля роли:

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

Удаление пароля роли:

```
ALTER ROLE davide WITH PASSWORD NULL;
```

Изменение срока действия пароля (в частности, определяется, что пароль должен перестать действовать в полдень 4 мая 2015 г. в часовом поясе UTC+1):

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```

Установка бесконечного срока действия пароля:

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

Наделение роли правами на создание других ролей и новых баз данных:

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

Определение нестандартного значения параметра `maintenance_work_mem` для роли:

```
ALTER ROLE worker_bee SET maintenance_work_mem = 100000;
```

Определение нестандартного значения параметра `client_min_messages` для роли и заданной базы:

```
ALTER ROLE fred IN DATABASE devel SET  
client_min_messages = DEBUG;
```

2.24.6. Совместимость

Оператор `ALTER ROLE` является расширением Синергия-БД.

2.25.ALTER ROUTINE

`ALTER ROUTINE` – изменить определение подпрограммы.

2.25.1. Синтаксис

```
ALTER ROUTINE имя [ ( [ [ режим_аргумента ]  
[ имя_аргумента ] тип_аргумента [, ...] ] ) ]  
действие [ ... ] [ RESTRICT ]
```

```
ALTER ROUTINE имя [ ( [ [ режим_аргумента ]  
[ имя_аргумента ] тип_аргумента [, ...] ] ) ]  
RENAME TO новое_имя
```

```
ALTER ROUTINE имя [ ( [ [ режим_аргумента ]  
[ имя_аргумента ] тип_аргумента [, ...] ] ) ]  
OWNER TO { новый_владелец | CURRENT_USER |  
SESSION_USER }
```

```
ALTER ROUTINE имя [ ( [ [ режим_аргумента ]
```

```
[ имя_аргумента ] тип_аргумента [, ...] ] ) ]
    SET SCHEMA новая_схема
ALTER ROUTINE имя [ ( [ [ режим_аргумента ]
[ имя_аргумента ] тип_аргумента [, ...] ] ) ]
    DEPENDS ON EXTENSION имя_расширения
```

Где действие может быть следующим:

```
IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
[ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ]
SECURITY DEFINER
PARALLEL { UNSAFE | RESTRICTED | SAFE }
COST стоимость_выполнения
ROWS строк_в_результате
SET параметр_конфигурации { TO | = } { значение |
DEFAULT }
SET параметр_конфигурации FROM CURRENT
RESET параметр_конфигурации
RESET ALL
```

2.25.2. Описание

ALTER ROUTINE изменяет определение подпрограммы, то есть агрегата, функции или процедуры.

2.25.3. Примеры

Переименование подпрограммы foo для типа integer в foobar:

```
ALTER ROUTINE foo(integer) RENAME TO foobar;
```

Эта команда будет работать независимо от того, является ли foo процедурой, агрегатной или обычной функцией.

2.25.4. Совместимость

Этот оператор частично совместим с оператором ALTER ROUTINE в стандарте SQL. Возможность сослаться по имени подпрограммы на агрегатную функцию является расширением Синергия-БД.

2.26.ALTER RULE

ALTER RULE – изменить определение правила.

2.26.1. Синтаксис

ALTER RULE имя ON имя_таблицы RENAME TO новое_имя

2.26.2. Описание

ALTER RULE изменяет свойства существующего правила. В настоящее время эта команда может только изменить имя правила.

Использовать ALTER RULE может только владелец таблицы (или представления), к которой применяется это правило.

2.26.3. Параметры

имя

Имя существующего правила, подлежащего изменению.

имя_таблицы

Имя (возможно, дополненное схемой) существующей таблицы (или представления), к которой применяется это правило.

новое_имя

Новое имя правила.

2.26.4. Примеры

Переименование существующего правила:

```
ALTER RULE notify_all ON emp RENAME TO notify_me;
```

2.26.5. Совместимость

Оператор ALTER RULE является языковым расширением Синергия-БД, как и вся система перезаписи запросов.

2.27.ALTER SCHEMA

ALTER SCHEMA – изменить определение схемы.

2.27.1. Синтаксис

```
ALTER SCHEMA имя RENAME TO новое_имя  
ALTER SCHEMA имя OWNER TO { новый_владелец |  
CURRENT_USER | SESSION_USER }
```

2.27.2. Описание

ALTER SCHEMA изменяет определение схемы.

Выполнить ALTER SCHEMA может только владелец соответствующей схемы. Чтобы переименовать схему, необходимо также иметь право CREATE в базе данных схемы. Чтобы сменить владельца необходимо быть непосредственным или опосредованным членом новой роли-владельца и иметь право CREATE в базе данных. Суперпользователи наделяются этими правами автоматически.

2.27.3. Параметры

имя

Имя существующей схемы.

новое_имя

Новое имя схемы. Новое имя не может начинаться с pg_, так как такие имена зарезервированы для системных схем.

новый_владелец

Новый владелец схемы.

2.27.4. Совместимость

Оператор ALTER SCHEMA отсутствует в стандарте SQL.

2.28.ALTER SEQUENCE

ALTER SEQUENCE – изменить определение генератора последовательности.

2.28.1. Синтаксис

```
ALTER SEQUENCE [ IF EXISTS ] имя  
[ AS тип_данных ]  
[ INCREMENT [ BY ] шаг ]  
[ MINVALUE мин_значение | NO MINVALUE ] [ MAXVALUE
```

```
макс_значение | NO MAXVALUE ]  
[ START [ WITH ] начало ]  
[ RESTART [ [ WITH ] перезапуск ] ]  
[ CACHE кеш ] [ [ NO ] CYCLE ]  
[ OWNED BY { имя_таблицы.имя_столбца | NONE } ]  
ALTER SEQUENCE [ IF EXISTS ] имя OWNER TO  
{ новый_владелец | CURRENT_USER | SESSION_USER }  
ALTER SEQUENCE [ IF EXISTS ] имя RENAME TO новое_имя  
ALTER SEQUENCE [ IF EXISTS ] имя SET SCHEMA новая_схема
```

2.28.2. Описание

ALTER SEQUENCE меняет параметры существующего генератора последовательности. Параметры, не определяемые явно в команде ALTER SEQUENCE, сохраняют свои предыдущие значения.

Выполнить ALTER SEQUENCE может только владелец соответствующей последовательности. Чтобы сменить схему последовательности, необходимо также иметь право CREATE в новой схеме. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право CREATE в схеме последовательности. С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать последовательность. Однако суперпользователь может сменить владельца последовательности в любом случае.

2.28.3. Параметры

имя

Имя (возможно, дополненное схемой) последовательности, подлежащей изменению.

IF EXISTS

Не считать ошибкой, если последовательность не существует. В этом случае будет выдано замечание.

тип_данных

Необязательное предложение AS *тип_данных* меняет тип данных для последовательности. Допустимые типы: smallint, integer и bigint.

При изменении типа данных автоматически меняются минимальное и максимальное значения последовательности, в том и только в том случае, если это были минимальные и максимальные значения старого типа данных (другими словами, если последовательность была создана со свойствами NO MINVALUE или NO MAXVALUE, неявно или явно). В противном случае минимальные и максимальные значения сохраняются, если только в этой же команде не указаны новые значения. Если минимальное/максимальное значение не помещается в новом типе данных, выдаётся ошибка.

шаг

Предложение INCREMENT BY шаг является необязательным. При положительном значении шага генерируется возрастающая последовательность, при отрицательном – убывающая; если шаг не указан, сохраняется предыдущее значение.

мин_значение
NO MINVALUE

Необязательное предложение MINVALUE мин_значение определяет минимальное значение, которое будет генерировать данная последовательность. Если указано NO MINVALUE, для возрастающей последовательности этим значением будет 1, а для убывающей – минимальное число для её типа данных. В отсутствие этих указаний будет сохранено текущее минимальное значение.

макс_значение
NO MAXVALUE

Необязательное предложение MAXVALUE макс_значение определяет максимальное значение, которое будет генерировать данная последовательность. Если указано NO MAXVALUE, для возрастающей последовательности этим значением будет максимальное число для её типа данных, а для убывающей – -1. В отсутствие этих указаний будет сохранено текущее максимальное значение.

начало

Необязательное предложение START WITH начало меняет записанное начальное значение последовательности. При этом текущее значение последовательности не меняется, а только устанавливается значение, которое будет

применено будущими командами ALTER SEQUENCE RESTART.

перезапуск

Необязательное предложение RESTART [WITH перезапуск] меняет текущее значение последовательности. Оно равнозначно вызову функции setval с параметром is_called = false: указанное значение перезапуска будет возвращено при следующем вызове функции nextval. Отсутствие в RESTART значения перезапуск равносильно передаче стартового значения, записанного командой CREATE SEQUENCE или последнего установленного командой ALTER SEQUENCE START WITH. В отличие от вызова setval, операция RESTART с последовательностью является транзакционной и не даёт параллельным транзакциям получать числа из той же последовательности. Если это поведение не устраивает, следует воспользоваться функцией setval.

кеш

Предложение CACHE кеш разрешает предварительно выделять и сохранять в памяти числа последовательности для ускорения доступа к ним. Минимальное значение равно 1 (т.е. за один раз генерируется только одно значение, кеширования нет). Если это предложение отсутствует, сохраняется старое значение размера кеша.

CYCLE

Необязательное ключевое слово CYCLE позволяет зациклить последовательность при достижении макс_значения или мин_значения для возрастающей и убывающей последовательности, соответственно. Когда этот предел достигается, следующим числом этих последовательностей будет соответственно мин_значение или макс_значение.

NO CYCLE

Если добавляется необязательное указание NO CYCLE, при каждом вызове nextval после достижения предельного значения будет возникать ошибка. Если же указания CYCLE и NO CYCLE отсутствуют, сохраняется предыдущая настройка зацикливания.

OWNED BY имя_таблицы.имя_колонки
OWNED BY NONE

Указание OWNED BY связывает последовательность с определенной колонкой

таблицы, с тем чтобы при удалении этой колонки (или всей таблицы) автоматически удалилась и последовательность. Это указание заменяет любую ранее установленную связь данной последовательности. Целевая таблица должна иметь того же владельца и находиться в той же схеме, что и последовательность. Указание OWNED BY NONE убирает все существующие связи, обозначая последовательность «независимой».

новый_владелец

Имя пользователя, назначаемого новым владельцем последовательности.

новое_имя

Новое имя последовательности.

новая_схема

Новая схема последовательности.

2.28.4. Замечания

ALTER SEQUENCE не оказывает немедленного влияния на результаты nextval в серверных процессах, кроме текущего, которые могли предварительно сгенерировать (кешировать) значения последовательности. Эти процессы заметят изменения только после того, как будут израсходованы все кешированные значения. Текущий серверный процесс реагирует на изменения сразу.

ALTER SEQUENCE не влияет на значение currval последовательности.

ALTER SEQUENCE блокирует параллельные вызовы nextval, currval, lastval и setval.

По историческим причинам ALTER TABLE тоже может работать с последовательностями, но все разновидности ALTER TABLE, допустимые для управления последовательностями, равнозначны вышеперечисленным формам.

2.28.5. Примеры

Перезапуск последовательности serial с числа 105:

```
ALTER SEQUENCE serial RESTART WITH 105;
```

2.28.6. Совместимость

Оператор ALTER SEQUENCE соответствует стандарту SQL, за исключением

предложений START WITH, OWNED BY, OWNER TO, RENAME TO и SET SCHEMA, являющихся расширениями Синергия-БД.

2.29.ALTER SERVER

ALTER SERVER – изменить определение стороннего сервера.

2.29.1. Синтаксис

```
ALTER SERVER имя [ VERSION 'новая_версия' ]  
      [ OPTIONS ( [ ADD | SET | DROP ] параметр  
      ['значение'] [, ... ] ) ]  
ALTER SERVER имя OWNER TO { новый_владелец |  
CURRENT_USER | SESSION_USER }  
ALTER SERVER имя RENAME TO новое_имя
```

2.29.2. Описание

ALTER SERVER изменяет определение стороннего сервера. Первая форма меняет строку версии сервера или общие параметры сервера (требуется минимум одно предложение). Вторая форма меняет владельца сервера.

Изменить настройки сервера может только его владелец. Чтобы изменить владельца, необходимо быть его владельцем, а также непосредственным или опосредованным членом новой роли-владельца, и кроме того, иметь право USAGE для обертки сторонних данных сервера. Суперпользователи удовлетворяют всем этим условиям автоматически.

2.29.3. Параметры

имя

Имя существующего сервера.

новая_версия

Новая версия сервера.

```
OPTIONS ( [ ADD | SET | DROP ] параметр ['значение']  
[, ... ] )
```

Эти формы изменяют параметры сервера. Указания ADD, SET и DROP определяют выполняемое действие (добавление, установка и удаление, соответственно). Если действие не задано явно, подразумевается ADD. Имена

параметров должны быть уникальными, они вместе со значениями также проверяются библиотекой обертки сторонних данных.

новый_владелец

Имя пользователя, назначаемого новым владельцем стороннего сервера.

новое_имя

Новое имя стороннего сервера.

2.29.4. Примеры

Изменение настройки сервера `foo`, добавление параметров подключения:

```
ALTER SERVER foo OPTIONS (host 'foo', dbname 'foodb');
```

Изменение настроек сервера `foo`: смена версии, изменение параметра `host`:

```
ALTER SERVER foo VERSION '8.4' OPTIONS (SET host 'baz');
```

2.29.5. Совместимость

`ALTER SERVER` соответствует стандарту ISO/IEC 9075-9 (SQL/MED). Формы `OWNER TO` и `RENAME` являются расширениями Синергия-БД.

2.30.ALTER SYSTEM

`ALTER SYSTEM` – изменить параметр конфигурации сервера.

2.30.1. Синтаксис

```
ALTER SYSTEM SET параметр_конфигурации { TO | = }  
{ значение | 'значение' | DEFAULT }
```

```
ALTER SYSTEM RESET параметр_конфигурации
```

```
ALTER SYSTEM RESET ALL
```

2.30.2. Описание

Оператор `ALTER SYSTEM` применяется для изменения параметров конфигурации сервера, распространяющихся на весь кластер баз данных. Пользоваться им может быть удобнее, чем вручную редактировать файл `postgresql.conf`. `ALTER SYSTEM` записывает заданное значение параметра в файл `postgresql.auto.conf`, который считывается сервером в дополнение к

`postgresql.conf`. При указании в качестве значения параметра `DEFAULT` или применении формы `RESET` соответствующий элемент конфигурации удаляется из `postgresql.auto.conf`. Удалить все настроенные таким способом параметры позволяет предложение `RESET ALL`.

Значения, установленные командой `ALTER SYSTEM`, вступают в силу только после следующей перезагрузки сервера (`SIGHUP` или `pg_ctl reload`), либо после следующего перезапуска, если эти параметры устанавливаются только при запуске сервера.

Выполнить `ALTER SYSTEM` могут только суперпользователи.

2.30.3. Параметры

параметр_конфигурации

Имя устанавливаемого параметра конфигурации.

значение

Новое значение параметра. Значениями могут быть строковые константы, идентификаторы, числа или списки таких элементов через запятую, в зависимости от конкретного параметра. Если в качестве значения указать `DEFAULT`, параметр и его значение удаляется из `postgresql.auto.conf`.

2.30.4. Замечания

С помощью этой команды нельзя задать `data_directory`, равно как и другие параметры, недопустимые в `postgresql.conf` (например, предустановленные параметры).

2.30.5. Примеры

Установка уровня ведения журнала транзакций (`wal_level`):

```
ALTER SYSTEM SET wal_level = hot_standby;
```

Отмена изменения, восстановление значения, заданного в `postgresql.conf`:

```
ALTER SYSTEM RESET wal_level;
```

2.30.6. Совместимость

Оператор ALTER SYSTEM является расширением Синергия-БД.

2.31.ALTER TABLE

ALTER TABLE – изменить определение таблицы

2.31.1. Синтаксис

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]  
    action [, ... ]  
ALTER TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]  
    RENAME [ COLUMN ] имя_колонки TO новое_имя_колонки  
ALTER TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]  
    RENAME CONSTRAINT имя_ограничения TO  
имя_нового_ограничения  
ALTER TABLE [ IF EXISTS ] имя  
    RENAME TO новое_имя  
ALTER TABLE [ IF EXISTS ] имя  
    SET SCHEMA новая_схема  
ALTER TABLE ALL IN TABLESPACE имя [ OWNED BY имя_роли  
[, ... ] ]  
    SET TABLESPACE новое_табл_пространство [ NOWAIT ]
```

Где возможные варианты action:

```
ADD [ COLUMN ] имя_колонки тип_данных [ COLLATE  
правило_сортировки ] [ ограничение_колонки [ ... ] ]  
DROP [ COLUMN ] [ IF EXISTS ] имя_колонки [ RESTRICT  
| CASCADE ]  
ALTER [ COLUMN ] имя_колонки [ SET DATA ] TYPE  
тип_данных [ COLLATE правило_сортировки ] [ USING  
выражение ]  
ALTER [ COLUMN ] имя_колонки SET DEFAULT выражение  
ALTER [ COLUMN ] имя_колонки DROP DEFAULT  
ALTER [ COLUMN ] имя_колонки { SET | DROP } NOT NULL  
ALTER [ COLUMN ] имя_колонки SET STATISTICS integer  
ALTER [ COLUMN ] имя_колонки SET ( атрибут =  
значение [, ... ] )
```

ALTER [COLUMN] имя_колонки RESET (атрибут
[, ...])

ALTER [COLUMN] имя_колонки SET STORAGE { PLAIN |
EXTERNAL | EXTENDED | MAIN }

ADD ограничение_таблицы [NOT VALID]

ADD ограничение_таблицы_по_индексу

ALTER CONSTRAINT имя_ограничения [DEFERRABLE | NOT
DEFERRABLE] [INITIALLY DEFERRED | INITIALLY
IMMEDIATE]

VALIDATE CONSTRAINT имя_ограничения

DROP CONSTRAINT [IF EXISTS] имя_ограничения
[RESTRICT | CASCADE]

DISABLE TRIGGER [имя_триггера | ALL | USER]

ENABLE TRIGGER [имя_триггера | ALL | USER]

ENABLE REPLICA TRIGGER имя_триггера

ENABLE ALWAYS TRIGGER имя_триггера

DISABLE RULE имя_правила_перезаписи

ENABLE RULE имя_правила_перезаписи

ENABLE REPLICA RULE имя_правила_перезаписи

ENABLE ALWAYS RULE имя_правила_перезаписи

DISABLE ROW LEVEL SECURITY

ENABLE ROW LEVEL SECURITY

FORCE ROW LEVEL SECURITY

NO FORCE ROW LEVEL SECURITY

CLUSTER ON имя_индекса

SET WITHOUT CLUSTER

SET WITH OIDS

SET WITHOUT OIDS

SET (параметр_хранения = значение [, ...])

RESET (параметр_хранения [, ...])

INHERIT таблица_родитель

NO INHERIT таблица_родитель

OF type_name

NOT OF

OWNER TO новый_владелец

SET TABLESPACE новое_табл_пространство

```
    REPLICATION IDENTITY {DEFAULT | USING INDEX имя_индекса  
    | FULL | NOTHING}
```

и ограничение_таблицы_по_индексу:

```
    [ CONSTRAINT имя_ограничения ]  
    { UNIQUE | PRIMARY KEY } USING INDEX имя_индекса  
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED  
    | INITIALLY IMMEDIATE ]
```

2.31.2. Описание

ALTER TABLE меняет определение существующей таблицы. Несколько её разновидностей описаны ниже. Заметьте, что для разных разновидностей могут требоваться разные уровни блокировок. Если явно не отмечено другое, требуется блокировка ACCESS EXCLUSIVE. При перечислении нескольких подкоманд будет запрашиваться самая сильная блокировка из требуемых ими.

```
ADD COLUMN
```

Форма добавляет в таблицу новую колонку, с тем же синтаксисом, что и CREATE TABLE.

```
ADD ограничение_таблицы_по_индексу
```

Форма добавляет в таблицу новое ограничение PRIMARY KEY или UNIQUE на базе существующего уникального индекса. В это ограничение будут включены все колонки данного индекса.

Индекс не может быть частичным и включать колонки-выражения. Кроме того, это должен быть индекс В-дерева с порядком сортировки по умолчанию. С такими ограничениями добавляемые индексы не будут ничем отличаться от индексов, создаваемых обычными командами ADD PRIMARY KEY и ADD UNIQUE.

В случае с указанием PRIMARY KEY, если колонки индекса ещё не помечены NOT NULL, данная команда попытается выполнить ALTER COLUMN SET NOT NULL для каждой колонки. При этом потребуется произвести полное сканирование таблицы, чтобы убедиться, что колонка(и) не содержат NULL. Во всех остальных случаях это быстрая операция.

Если задано имя ограничения, индекс будет переименован и получит заданное имя. В противном случае, именем ограничения станет имя индекса.

После выполнения этой команды индекс становится «принадлежащим» ограничению, так же, как если бы он был создан обычной командой `ADD PRIMARY KEY` или `ADD UNIQUE`. Это значит, в частности, что при удалении ограничения индекс будет удален вместе с ним.

Примечание. Добавление ограничения на базе существующего индекса бывает полезно в ситуациях, когда новое ограничение требуется добавить, не блокируя изменения в таблице на долгое время. Для этого можно создать индекс командой `CREATE INDEX CONCURRENTLY`, а затем задействовать его как полноценное ограничение, используя эту запись.

`ALTER CONSTRAINT`

Форма меняет атрибуты созданного ранее ограничения. В настоящее время изменять можно только ограничения внешнего ключа.

`VALIDATE CONSTRAINT`

Форма проверяет ограничение внешнего ключа или ограничение-проверку, созданное ранее с указанием `NOT VALID`, сканируя всю таблицу с целью убедиться, что ограничению удовлетворяют все строки. Если ограничение уже помечено как верное, ничего не происходит.

Проверка больших таблиц может быть довольно длительной. Смысл отделения проверки от собственно создания ограничения состоит в том, что это позволяет отложить проверку на время меньшей активности или провести дополнительную работу с существующими ошибками и при этом не допустить новых. Заметьте также, что во время осуществления проверки сама по себе она не препятствует выполнению обычных команд записи в таблицу.

При проверке запрашивается только блокировка `SHARE UPDATE EXCLUSIVE` целевой таблицы. Для ограничений внешнего ключа требуется также блокировка `ROW SHARE` в таблице, на которую ссылается ограничение.

`DROP CONSTRAINT [ IF EXISTS ]`

Форма удаляет указанное ограничение таблицы. Если указано `IF EXISTS` и заданное ограничение не существует, это не считается ошибкой. В этом случае выдается только замечание.

`DISABLE/ENABLE [ REPLICA | ALWAYS ] TRIGGER`

Формы настраивают срабатывание триггера(ов), принадлежащего таблице.

Отключенный триггер сохраняется в системе, но не выполняется, когда происходит вызывающего его событие. Для отложенных триггеров состояние включения проверяется при возникновении события, а не когда на самом деле вызывается функция триггера. Эта команда может отключить или включить один триггер по имени, либо все триггеры таблицы, либо только пользовательские триггеры (исключая сгенерированные внутрисистемные триггеры ограничений, например, триггеры, реализующие ограничения внешнего ключа или отложенные ограничения уникальности или исключений). Для отключения или включения сгенерированных внутрисистемных триггеров ограничений требуются права суперпользователя; отключать их следует с осторожностью, так как очевидно, что гарантировать целостность ограничений, если триггеры не работают, невозможно. На механизм срабатывания триггеров также влияет конфигурационная переменная `session_replication_role`. Просто включаемые триггеры будут срабатывать, когда роль репликации – «origin» (по умолчанию) или «local». Триггеры, включенные указанием `ENABLE REPLICA`, будут срабатывать, только если текущий режим сеанса – «replica», а после указания `ENABLE ALWAYS` триггеры срабатывают вне зависимости от текущего режима репликации.

`DISABLE/ENABLE [ REPLICA | ALWAYS ] RULE`

Формы настраивают срабатывание правил перезаписи, относящихся к таблице. Отключенное правило сохраняется в системе, но не применяется во время переписывания запроса. По сути эти операции подобны операциям включения/отключения триггеров. Однако настройка правил `ON SELECT` игнорируется, они применяются всегда, чтобы представления продолжали работать, даже в сеансах, исполняющих не основную роль репликации.

`DISABLE/ENABLE ROW LEVEL SECURITY`

Формы управляют применением политик защиты строк, принадлежащих таблице. Если применение разрешено, но политики для таблицы не определены, применяется политика запрещения по умолчанию. Заметьте, что политики могут существовать, даже если их применение отключено – в этом случае политики игнорируются и не применяются.

`NO FORCE/FORCE ROW LEVEL SECURITY`

Формы управляют применением политик защиты строк, принадлежащих таблице, когда пользователь является её владельцем. Если указано `FORCE ROW LEVEL SECURITY`, политики будут применяться. В противном случае (по умолчанию) политики не применяются.

`CLUSTER ON`

Форма выбирает индекс по умолчанию для последующих операций `CLUSTER`. Собственно кластеризация таблицы при этом не выполняется.

Для изменения параметров кластеризации запрашивается блокировка `SHARE UPDATE EXCLUSIVE`.

`SET WITHOUT CLUSTER`

Форма удаляет последнее заданное указание индекса для `CLUSTER`. Его действие отразится на будущих операциях кластеризации, для которых не будет задан индекс.

Для изменения параметров кластеризации запрашивается блокировка `SHARE UPDATE EXCLUSIVE`.

`SET WITH OIDS`

Форма добавляет в таблицу системную колонку `oid`. Если в таблице уже есть такая колонка, она не делает ничего.

Заметьте, что это не равнозначно команде `ADD COLUMN oid oid` (эта команда добавит не системную, а обычную колонку с подходящим именем `oid`).

`SET WITHOUT OIDS`

Форма удаляет из таблицы системную колонку `oid`. Это в точности равнозначно `DROP COLUMN oid RESTRICT`, за исключением того, что в случае отсутствия колонки `oid` ошибки не будет.

`SET ( параметр_хранения = значение [, ... ] )`

Форма меняет один или несколько параметров хранения таблицы. Заметьте, что эта команда не меняет содержимое таблицы немедленно; в зависимости от параметра может потребоваться перезаписать таблицы, чтобы получить желаемый эффект. Это можно сделать с помощью команд `VACUUM`, `CLUSTER` или одной из форм `ALTER TABLE`, принудительно перезаписывающих таблицу.

Примечание. Хотя CREATE TABLE позволяет указать OIDS в синтаксисе WITH (параметр_хранения), ALTER TABLE не воспринимает OIDS как параметр хранения. Поэтому для изменения характеристики OID следует применять формы SET WITH OIDS и SET WITHOUT OIDS.

RESET (параметр_хранения [, ...])

Форма сбрасывает один или несколько параметров хранения к значениям по умолчанию. Как и с SET, для полного обновления таблицы может потребоваться перезаписать таблицу.

INHERIT таблица_родитель

Форма назначает целевую таблицу потомком заданной родительской таблицы. Впоследствии запросы к родительской таблице будут включать записи и целевой таблицы. Чтобы таблица могла стать потомком, она должна содержать те же колонки, что и родительская (хотя она может включать и дополнительные колонки). Колонки должны иметь одинаковые типы данных и, если в родительской таблице какие-то из них имеют ограничение NOT NULL, они должны иметь ограничение NOT NULL и в таблице-потомке.

Также в таблице-потомке должны присутствовать все ограничения CHECK родительской таблицы, за исключением ненаследуемых (то есть созданных командой ALTER TABLE ... ADD CONSTRAINT ... NO INHERIT), которые игнорируются; при этом все соответствующие ограничения в таблице-потомке не должны быть ненаследуемыми. В настоящее время ограничения UNIQUE, PRIMARY KEY и FOREIGN KEY не учитываются, но в будущем это может измениться.

NO INHERIT таблица_родитель

Форма удаляет целевую таблицу из списка потомков указанной родительской таблицы. Результаты запросов к родительской таблице после этого не будут включать записи, взятые из целевой таблицы.

OF type_name

Форма связывает таблицу с составным типом, как если бы она была сформирована командой CREATE TABLE OF. При этом список имен и типов колонок должен точно соответствовать тому, что образует составной тип; отличие возможно в системном столбце oid. Кроме того, таблица не должна быть потомком

какой-либо другой таблицы. Эти ограничения – залог того, что команда `CREATE TABLE OF` позволит создать таблицу с таким же определением.

`NOT OF`

Форма разрывает связь типизированной таблицы с её типом.

`OWNER`

Форма меняет владельца таблицы, последовательности, представления, материализованного представления или сторонней таблицы на заданного пользователя.

`SET TABLESPACE`

Форма меняет табличное пространство таблицы на заданное и перемещает файлы данных, связанные с таблицей, в новое пространство. Индексы таблицы, если они имеются, не перемещаются; однако их можно переместить отдельно дополнительными командами `SET TABLESPACE`. Форма `ALL IN TABLESPACE` позволяет перенести в другое табличное пространство все таблицы текущей базы данных, при этом она сначала блокирует все таблицы, а затем переносит каждую из них. Эта форма также поддерживает указание `OWNED BY`, с которым перемещаются только таблицы указанного владельца. Если указан параметр `NOWAIT`, команда завершится ошибкой, если не сможет получить все требуемые блокировки немедленно. Заметьте, что системные каталоги эта форма не перемещает; если требуется переместить их, следует использовать `ALTER DATABASE` или явные вызовы `ALTER TABLE`. Отношения `information_schema` не считаются частью системных каталогов и подлежат перемещению.

`REPLICA IDENTITY`

Форма меняет информацию, записываемую в журнал упреждающей записи для идентификации изменяемых или удаляемых строк. Данная настройка действует только при использовании логической репликации. В режиме `DEFAULT` (по умолчанию для не системных таблиц) записывается старые значения колонок первичного ключа, если он есть. В режиме `USING INDEX` записываются старые значения колонок, составляющих заданный индекс, который должен быть уникальным, не частичным, не отложенным и включать только колонки, помеченные

NOT NULL. В режиме FULL записываются старые значения всех колонок в строке, а в режиме NOTHING (по умолчанию для системных таблиц) никакая информация о старой строке не записывается. Во всех случаях старые значения записываются в журнал, только если как минимум в одной колонке из тех, что должны быть записаны, произошли изменения в новой строке.

RENAME

Формы RENAME меняют имя таблицы (или индекса, последовательности, представления, материализованного представления или сторонней таблицы), имя отдельной колонки таблицы или имя ограничения таблицы. На хранимые данные это не влияет.

SET SCHEMA

Форма перемещает таблицу в другую схему. Вместе с таблицей перемещаются связанные с ней индексы и ограничения, а также последовательности, принадлежащие колонкам таблицы.

Все действия, кроме RENAME, SET TABLESPACE и SET SCHEMA, можно объединить в список множественных изменений и применить параллельно. Например, можно добавить несколько колонок и/или изменить тип колонок в одной команде. Это особенно полезно для больших таблиц, так как вся таблица обрабатывается за один проход.

Выполнить ALTER TABLE может только владелец соответствующей таблицы. Чтоб сменить схему или табличное пространство таблицы, необходимо также иметь право CREATE в новой схеме или табличном пространстве. Чтобы сделать таблицу потомком другой таблицы, нужно быть владельцем и родительской таблицы. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право CREATE в схеме таблицы. С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать таблицу. Однако суперпользователь может сменить владельца таблицы в любом случае. Чтобы добавить колонку, сменить тип колонки или применить предложение OF, необходимо также иметь право USAGE для соответствующего типа данных.

2.31.3. Параметры

ИМЯ

Имя (возможно, дополненное схемой) существующей таблицы, подлежащей изменению. Если перед именем таблицы указано ONLY, изменяется только заданная таблица. Без ONLY изменяется и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно также добавить необязательное указание *, чтобы явно обозначить, что изменению подлежат все дочерние таблицы.

ИМЯ_КОЛОНКИ

Имя новой или существующей колонки.

НОВОЕ_ИМЯ_КОЛОНКИ

Новое имя существующей колонки.

НОВОЕ_ИМЯ

Новое имя таблицы.

ТИП_ДАННЫХ

Тип данных новой колонки или новый тип данных существующей колонки.

ОГРАНИЧЕНИЕ_ТАБЛИЦЫ

Новое ограничение таблицы.

ИМЯ_ОГРАНИЧЕНИЯ

Имя нового или существующего ограничения.

CASCADE

Автоматически удалять объекты, зависящие от удаляемой колонки или ограничения (например, представления, содержащие эту колонку).

RESTRICT

Отказаться в удалении колонки или ограничения, если существуют зависящие от них объекты. Это поведение по умолчанию.

ИМЯ_ТРИГГЕРА

Имя включаемого или отключаемого триггера.

ALL

Отключить или включить все триггеры, принадлежащие таблице. Для этого требуются права суперпользователя, если в числе этих триггеров оказываются сгенерированные внутрисистемные триггеры исключений, например те, что

реализуют ограничения внешнего ключа или отложенные ограничения уникальности и исключений.

USER

Отключить или включить все триггеры, принадлежащие таблице, за исключением сгенерированных внутрисистемных триггеров исключений, например, тех, что реализуют ограничения внешнего ключа или отложенные ограничения уникальности и исключений.

имя_индекса

Имя существующего индекса.

параметр_хранения

Имя параметра хранения таблицы

значение

Новое значение параметра хранения таблицы. Это может быть число или строка, в зависимости от параметра.

таблица_родитель

Родительская таблица, с которой будет установлена или разорвана связь данной таблицы.

новый_владелец

Имя пользователя, назначаемого новым владельцем таблицы.

новое_табл_пространство

Имя табличного пространства, в которое будет перемещена таблица.

новая_схема

Имя схемы, в которую будет перемещена таблица.

2.31.4. Замечания

Ключевое слово COLUMN не несет смысловой нагрузки и может быть опущено.

Когда колонка добавляется с помощью ADD COLUMN, во всех существующих в таблице строках эта колонка инициализируется значением по умолчанию (или NULL, если предложение DEFAULT для колонки отсутствует). Если предложение DEFAULT отсутствует, это сводится только к изменению метаданных, непосредственного изменения данных таблицы не происходит; добавленные значения NULL выводятся

при чтении.

Добавление колонки с предложением `DEFAULT` или изменение типа существующей колонки влечет за собой перезапись всей таблицы и её индексов. Но возможно исключение при смене типа существующей колонки: если предложение `USING` не меняет содержимое колонки и старый тип двоично приводится к новому или является неограниченным доменом поверх нового типа, перезапись таблицы не требуется; хотя все индексы с затронутыми колонками все же требуется перестроить. При добавлении или удалении системной колонки `oid` также необходима перезапись всей таблицы. Для перестроения больших таблиц и/или их индексов может понадобиться довольно много времени и временно потребуются вдвое больше места на диске.

Добавление ограничения `CHECK` или `NOT NULL` требует просмотра таблицы для того, чтобы проверить, что существующие строки удовлетворяют ограничению, но не требует перезаписи таблицы.

Возможность объединения множества изменений в одну команду `ALTER TABLE` полезна в основном тем, что позволяет совместить сканирования и перезаписи таблицы, требуемые этим операциям, и выполнить их за один проход.

Форма `DROP COLUMN` не удаляет колонку физически, а просто делает её невидимой для операций `SQL`. При последующих операциях добавления или изменения в эту колонку будет записываться значение `NULL`. Таким образом, удаление колонки выполняется быстро, но при этом размер таблицы на диске не уменьшается, так как пространство, занимаемое удаленной колонкой, не высвобождается. Это пространство будет освобождено со временем, по мере изменения существующих строк. При удалении системной колонки `oid` это поведение не наблюдается, так как немедленно выполняется перезапись таблицы.

Чтобы немедленно освободить место, занимаемое удаленным столбцом, можно выполнить одну из форм команды `ALTER TABLE`, которая выполняет перезапись всей таблицы. Это приведет к повторному созданию каждой строки со значением `NULL` в удаленном столбце.

Перезаписывающие формы `ALTER TABLE` не являются безопасными с точки

зрения многоверсионности. После перезаписи таблицы, она будет выглядеть пустой для одновременно выполняемых транзакций, которые используют снимок, сделанный до того, как произошла перезапись.

В указании `USING` предложения `SET DATA TYPE` на самом деле можно записать выражение со старыми значениями строки; то есть, оно может ссылаться как на преобразуемые колонки, так и на другие. Это позволяет записывать в `SET DATA TYPE` очень общие преобразования данных. Ввиду такой гибкости, выражение `USING` не применяется к значению по умолчанию данной колонки (если таковое есть); результат может быть не константным выражением, что требуется для значения по умолчанию. Это означает, что в случае отсутствия явного приведения или присваивания старого типа новому, `SET DATA TYPE` может не справиться с преобразованием значения по умолчанию, несмотря на то, что применяется предложение `USING`. В этих случаях нужно удалить значение по умолчанию с помощью `DROP DEFAULT`, выполнить `ALTER TYPE`, а затем с помощью `SET DEFAULT` задать новое подходящее значение по умолчанию. Подобные соображения применимы и в отношении индексов и ограничений с этой колонкой.

Если у таблицы есть дочерние таблицы, добавлять, переименовывать или менять тип колонки, либо переименовывать наследуемое ограничение в родительской таблице, не делая того же самого во всех дочерних таблицах, нельзя. То есть, команда `ALTER TABLE ONLY` выполнена не будет. Это гарантирует, что дочерние таблицы всегда будут содержать те же колонки, что и родительская.

Рекурсивная операция `DROP COLUMN` удалит колонку из дочерней таблицы, только если эта колонка не наследуется от каких-то других родителей и не была определена в дочерней таблице независимо. Нерекурсивная операция `DROP COLUMN` (т.е., `ALTER TABLE ONLY ... DROP COLUMN`) никогда не удаляет унаследованные колонки, но вместо этого помечает их как независимо определенные, а не наследуемые.

Действия `TRIGGER`, `CLUSTER`, `OWNER` и `TABLESPACE` никогда не распространяются рекурсивно на дочерние таблицы; то есть, они всегда выполняются так, как будто указано `ONLY`. Операция добавления ограничения

выполняется рекурсивно только для ограничений CHECK, не помеченных как NO INHERIT.

Какие-либо изменения таблиц системного каталога не допускаются.

2.31.5. Примеры

Добавление в таблицу колонки типа varchar:

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

Удаление колонки из таблицы:

```
ALTER TABLE distributors DROP COLUMN address RESTRICT;
```

Изменение типов двух существующих колонок в одной операции:

```
ALTER TABLE distributors  
    ALTER COLUMN address TYPE varchar(80),  
    ALTER COLUMN name TYPE varchar(100);
```

Смена типа целочисленной колонки, содержащей время в стиле UNIX, на тип timestamp with time zone с применением предложения USING:

```
ALTER TABLE foo  
    ALTER COLUMN foo_timestamp SET DATA TYPE timestamp  
    with time zone  
    USING  
        timestamp with time zone 'epoch' + foo_timestamp  
        * interval '1 second';
```

То же самое, но в случае, когда у колонки есть значение по умолчанию, не приводимое автоматически к новому типу данных:

```
ALTER TABLE foo  
    ALTER COLUMN foo_timestamp DROP DEFAULT,  
    ALTER COLUMN foo_timestamp TYPE timestamp with time  
    zone  
  
    USING  
        timestamp with time zone 'epoch' + foo_timestamp  
        * interval '1 second',  
    ALTER COLUMN foo_timestamp SET DEFAULT now();
```

Переименование существующей колонки:

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

Переименование существующей таблицы:

```
ALTER TABLE distributors RENAME TO suppliers;
```

Переименование существующего ограничения:

```
ALTER TABLE distributors RENAME CONSTRAINT zipchk TO  
zip_check;
```

Добавление в колонку ограничения NOT NULL:

```
ALTER TABLE distributors ALTER COLUMN street SET NOT  
NULL;
```

Удаление ограничения NOT NULL из колонки:

```
ALTER TABLE distributors ALTER COLUMN street DROP NOT  
NULL;
```

Добавление ограничения-проверки в таблицу и все её потомки:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK  
(char_length(zipcode) = 5);
```

Добавление ограничения-проверки только в таблицу, но не в её потомки:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK  
(char_length(zipcode) = 5) NO INHERIT;
```

Данное ограничение-проверка не будет наследоваться и будущими потомками
тоже.

Удаление ограничения-проверки из таблицы и из всех её потомков:

```
ALTER TABLE distributors DROP CONSTRAINT zipchk;
```

Удаление ограничения-проверки только из самой таблицы:

```
ALTER TABLE ONLY distributors DROP CONSTRAINT zipchk;
```

Ограничение-проверка остается во всех дочерних таблицах.

Добавление в таблицу ограничения внешнего ключа:

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN  
KEY (address) REFERENCES addresses (address);
```

Добавление в таблицу ограничения внешнего ключа с наименьшим влиянием
на работу других:

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN  
KEY (address) REFERENCES addresses (address) NOT VALID;
```

```
ALTER TABLE distributors VALIDATE CONSTRAINT distfk;
```

Добавление в таблицу ограничения уникальности (по нескольким колонкам):

```
ALTER TABLE distributors ADD CONSTRAINT
```

```
dist_id_zipcode_key UNIQUE (dist_id, zipcode);
```

Добавление в таблицу первичного ключа с автоматическим именем (учтите, что в таблице может быть только один первичный ключ):

```
ALTER TABLE distributors ADD PRIMARY KEY (dist_id);
```

Перемещение таблицы в другое табличное пространство:

```
ALTER TABLE distributors SET TABLESPACE fasttablespace;
```

Перемещение таблицы в другую схему:

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

Пересоздание ограничения первичного ключа без блокировки изменений в процессе перестроения индекса:

```
CREATE UNIQUE INDEX CONCURRENTLY dist_id_temp_idx ON  
distributors (dist_id);
```

```
ALTER TABLE distributors DROP CONSTRAINT  
distributors_pkey,
```

```
    ADD CONSTRAINT distributors_pkey PRIMARY KEY USING  
INDEX dist_id_temp_idx;
```

2.31.6. Совместимость

Формы ADD (без USING INDEX), DROP, SET DEFAULT и SET DATA TYPE (без USING) соответствуют стандарту SQL. Другие формы являются расширениями стандарта SQL, реализованными в Синергия-БД. Кроме того, расширением является возможность указать в одной команде ALTER TABLE несколько операций изменения.

ALTER TABLE DROP COLUMN позволяет удалить единственную колонку таблицы и оставить таблицу без колонок. Это является расширением стандарта SQL, который не допускает существование таблиц с нулем колонок.

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

SQL	-	Structured Query Language (язык структурированных запросов)
WAL	-	Write-Ahead Log (журнал упреждающей записи)
БД	-	база данных
КСЗ	-	комплекс средств защиты
ОЗУ	-	оперативное запоминающее устройство
ОС	-	операционная система
ПРД	-	правила разграничения доступа
СУБД	-	система управления базами данных

Лист регистрации изменений

[illegible]

Федеральное государственное унитарное предприятие
Российский федеральный ядерный центр
Всероссийский научно-исследовательский институт экспериментальной физики

УТВЕРЖДЕН
07623615.00082-02 34 01-ЛУ

КОМПЛЕКС ПРОГРАММ В ЗАЩИЩЕННОМ ИСПОЛНЕНИИ
«СИСТЕМА ПОЛНОГО ЖИЗНЕННОГО ЦИКЛА ИЗДЕЛИЙ
«ЦИФРОВОЕ ПРЕДПРИЯТИЕ»

**Программный модуль
«Система управления базами данных «Синергия-БД»**

Руководство оператора

Часть 3

07623615.00082-02 34 01-3

Листов 273

Име. № подл.	Подп. и дата	Взам. инв. №	Име. № дубл.	Подп. и дата

СОДЕРЖАНИЕ

1. ALTER TABLESPACE	5
2. ALTER TEXT SEARCH CONFIGURATION	6
3. ALTER TEXT SEARCH DICTIONARY	9
4. ALTER TEXT SEARCH PARSER	10
5. ALTER TEXT SEARCH TEMPLATE	11
6. ALTER TRIGGER	12
7. ALTER TYPE	13
8. ALTER USER	17
9. ALTER USER MAPPING	18
10. ALTER VIEW	19
11. ANALYZE	22
12. BEGIN	25
13. CHECKPOINT	27
14. CLOSE	28
15. CLUSTER	29
16. COMMENT	32
17. COMMIT	37
18. COMMIT PREPARED	38
19. COPY	39
20. CREATE AGGREGATE	53
21. CREATE CAST	63
22. CREATE COLLATION	69
23. CREATE CONVERSION	71
24. CREATE DATABASE	73
25. CREATE DOMAIN	77
26. CREATE EVENT TRIGGER	80
27. CREATE EXTENSION	82
28. CREATE FOREIGN DATA WRAPPER	84
29. CREATE FOREIGN TABLE	87

30. CREATE FUNCTION.....	90
31. CREATE GROUP	101
32. CREATE INDEX	102
33. CREATE LANGUAGE.....	112
34. CREATE MATERIALIZED VIEW	116
35. CREATE OPERATOR	117
36. CREATE OPERATOR CLASS.....	121
37. CREATE OPERATOR FAMILY.....	125
38. CREATE POLICY	126
39. CREATE ROLE	132
40. CREATE RULE	137
41. CREATE SCHEMA	142
42. CREATE SEQUENCE	145
43. CREATE SERVER.....	150
44. CREATE STATISTICS	152
45. CREATE SUBSCRIPTION	154
46. CREATE TABLE	158
47. CREATE TABLE AS.....	193
48. CREATE TABLESPACE	197
49. CREATE TEXT SEARCH CONFIGURATION	199
50. CREATE TEXT SEARCH DICTIONARY	200
51. CREATE TEXT SEARCH PARSER	202
52. CREATE TEXT SEARCH TEMPLATE.....	203
53. CREATE TRANSFORM	205
54. CREATE TRIGGER.....	208
55. CREATE TYPE	218
56. CREATE USER.....	234
57. CREATE USER MAPPING.....	235
58. CREATE VIEW.....	236
59. DEALLOCATE	244

60. DECLARE.....	245
61. DELETE	250
62. DISCARD.....	253
63. DO	255
64. DROP ACCESS METHOD	256
65. DROP AGGREGATE.....	257
66. DROP CAST	259
67. DROP COLLATION	260
68. DROP CONVERSION.....	261
69. DROP DATABASE	262
70. DROP DOMAIN	263
71. DROP EVENT TRIGGER	264
72. DROP EXTENSION	264
73. DROP FOREIGN DATA WRAPPER	266
74. DROP FOREIGN TABLE.....	267
75. DROP FUNCTION	268
76. DROP GROUP	270
77. DROP INDEX	270
Перечень сокращений	272

Документ содержит справочную информацию о командах SQL, поддерживаемых СУБД «Синергия-БД» (далее Синергия-БД). Под «SQL» понимается язык в общем; информация о соответствии стандарту SQL и совместимости с ним приведена на страницах, соответствующих каждой команде.

1. ALTER TABLESPACE

ALTER TABLESPACE – изменить определение табличного пространства.

1.1. Синтаксис

ALTER TABLESPACE имя RENAME TO новое_имя

ALTER TABLESPACE имя OWNER TO новый_владелец

ALTER TABLESPACE имя SET (параметр_табличного
пространства = значение [, ...])

ALTER TABLESPACE имя RESET (параметр_табличного
пространства [, ...])

1.2. Описание

ALTER TABLESPACE может применяться для изменения определения табличного пространства.

Чтобы изменить определение табличного пространства, нужно быть его владельцем. Чтобы изменить владельца, нужно быть непосредственным или опосредованным членом новой роли-владельца. Заметьте, что суперпользователи наделяются этими правами автоматически.

1.3. Параметры

имя

Имя существующего табличного пространства.

новое_имя

Новое имя табличного пространства. Новое имя не может начинаться с pg_, так как такие имена зарезервированы для системных табличных пространств.

новый_владелец

Новый владелец табличного пространства.

параметр_табличного пространства

Устанавливаемый или сбрасываемый параметр табличного пространства. В настоящее время поддерживаются только параметры `seq_page_cost` и `random_page_cost`. При установке этих значений для заданного табличного пространства будет переопределена обычная оценка стоимости чтения страниц из таблиц в этом пространстве, настраиваемая одноименным параметром конфигурации. Это может быть полезно, если одно из табличных пространств размещено на диске, который быстрее или медленнее остальной дисковой системы.

1.4. Примеры

Переименование табличного пространства `index_space` в `fast_raid`:

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

Смена владельца табличного пространства `index_space`:

```
ALTER TABLESPACE index_space OWNER TO mary;
```

1.5. Совместимость

Оператор `ALTER TABLESPACE` отсутствует в стандарте SQL.

2. ALTER TEXT SEARCH CONFIGURATION

`ALTER TEXT SEARCH CONFIGURATION` – изменить определение конфигурации текстового поиска.

2.1. Синтаксис

```
ALTER TEXT SEARCH CONFIGURATION имя
```

```
    ADD MAPPING FOR тип_фрагмента [, ... ] WITH  
    имя_словаря [, ... ]
```

```
ALTER TEXT SEARCH CONFIGURATION имя
```

```
    ALTER MAPPING FOR тип_фрагмента [, ... ] WITH  
    имя_словаря [, ... ]
```

```
ALTER TEXT SEARCH CONFIGURATION имя
```

```
    ALTER MAPPING REPLACE старый_словарь WITH  
    новый_словарь
```

```
ALTER TEXT SEARCH CONFIGURATION имя
```

```
    ALTER MAPPING FOR тип_фрагмента [, ... ] REPLACE  
старый_словарь WITH новый_словарь
```

```
ALTER TEXT SEARCH CONFIGURATION имя
```

```
    DROP MAPPING [ IF EXISTS ] FOR тип_фрагмента  
[, ... ]
```

```
ALTER TEXT SEARCH CONFIGURATION имя RENAME TO новое_имя
```

```
ALTER TEXT SEARCH CONFIGURATION имя OWNER TO  
новый_владелец
```

```
ALTER TEXT SEARCH CONFIGURATION имя SET SCHEMA  
новая_схема
```

2.2. Описание

ALTER TEXT SEARCH CONFIGURATION изменяет определение конфигурации текстового поиска. Эта команда позволяет настроить сопоставления типов фрагментов со словарями или сменить владельца или имя конфигурации.

Выполнить ALTER TEXT SEARCH CONFIGURATION может только владелец соответствующей конфигурации.

2.3. Параметры

имя

Имя (возможно, дополненное схемой) существующей конфигурации текстового поиска.

тип_фрагмента

Имя типа фрагмента, выдаваемое при разборе конфигурации.

имя_словаря

Имя словаря текстового поиска, в котором будет искаться указанный тип фрагмента. Если указаны несколько словарей, они просматриваются в порядке перечисления.

старый_словарь

Имя словаря текстового поиска, которое будет заменено в сопоставлении.

новый_словарь

Имя словаря текстового поиска, которое будет подставлено там, где был

старый_словарь.

новое_имя

Новое имя конфигурации текстового поиска.

новый_владелец

Новый владелец конфигурации текстового поиска.

новая_схема

Новая схема конфигурации текстового поиска.

Форма `ADD MAPPING FOR` настраивает список словарей, которые будут просматриваться в поиске указанных типов фрагментов; если сопоставление для каких-либо типов уже задано, возникнет ошибка. Форма `ALTER MAPPING FOR` делает то же самое, но она сначала удаляет существующее сопоставление для этих типов фрагментов. Формы `ALTER MAPPING REPLACE` подставляют новый_словарь вместо старый_словарь везде, где упоминается последний. Это выполняется только для указанных типов фрагментов, когда присутствует `FOR`, либо для всех сопоставлений в конфигурации в противном случае. Форма `DROP MAPPING` удаляет все словари для заданных типов фрагментов, в результате чего фрагменты этих типов будут игнорироваться конфигурацией. Если сопоставлений для заданных типов фрагментов нет, возникает ошибка, если только не добавлено указание `IF EXISTS`.

2.4. Примеры

В следующем примере словарь `english` заменяется на `swedish` везде, где использовался `english` в конфигурации `my_config`.

```
ALTER TEXT SEARCH CONFIGURATION my_config
ALTER MAPPING REPLACE english WITH swedish;
```

2.5. Совместимость

Оператор `ALTER TEXT SEARCH CONFIGURATION` отсутствует в стандарте SQL.

3. ALTER TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH DICTIONARY – изменить определение словаря текстового поиска.

3.1. Синтаксис

```
ALTER TEXT SEARCH DICTIONARY имя (  
    параметр [ = значение ] [, ... ]  
)
```

```
ALTER TEXT SEARCH DICTIONARY имя RENAME TO новое_имя  
ALTER TEXT SEARCH DICTIONARY имя OWNER TO новый_владелец  
ALTER TEXT SEARCH DICTIONARY имя SET SCHEMA новая_схема
```

3.2. Описание

ALTER TEXT SEARCH DICTIONARY изменяет определение словаря текстового поиска. Эта команда позволяет изменить параметры словаря, связанные с шаблонами, или сменить владельца или имя словаря.

Выполнить ALTER TEXT SEARCH DICTIONARY может только владелец словаря.

3.3. Параметры

имя

Имя (возможно, дополненное схемой) существующего словаря текстового поиска.

параметр

Имя параметра шаблона, устанавливаемого для данного словаря.

значение

Новое значение для параметра настройки шаблонов. Если знак равно и значение опущено, предыдущее значение параметра удаляется из словаря, что позволяет вернуться к значению по умолчанию.

новое_имя

Новое имя словаря текстового поиска.

новый_владелец

Новый владелец словаря текстового поиска.

новая_схема

Новая схема словаря текстового поиска.

Параметры настройки шаблонов могут перечисляться в любом порядке.

3.4. Примеры

Команда в следующем примере меняет список стоп-слов словаря на базе Snowball. Другие параметры остаются неизменными.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( StopWords =  
newrussian );
```

Команда в следующем примере меняет параметр, определяющий язык, на dutch, и удаляет параметр, задающий список стоп-слов.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( language = dutch,  
StopWords );
```

Следующая команда «изменяет» определение словаря, на самом деле не меняя ничего.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( dummy );
```

Это работает потому, что код удаления параметра не считает ошибкой отсутствие такого параметра. Этот трюк может быть полезен при изменении файлов конфигурации словаря; ALTER принудит все существующие сеансы перечитать файлы конфигурации, что в противном случае они не сделают никогда, если прочитали конфигурацию ранее.

3.5. Совместимость

Оператор ALTER TEXT SEARCH DICTIONARY отсутствует в стандарте SQL.

4. ALTER TEXT SEARCH PARSER

ALTER TEXT SEARCH PARSER – изменить определение анализатора текстового поиска.

4.1. Синтаксис

```
ALTER TEXT SEARCH PARSER имя RENAME TO новое_имя
```

```
ALTER TEXT SEARCH PARSER имя SET SCHEMA новая_схема
```

4.2. Описание

`ALTER TEXT SEARCH PARSER` изменяет определение анализатора текстового поиска. В настоящее время эта команда позволяет только сменить его имя.

Выполнить `ALTER TEXT SEARCH PARSER` может только суперпользователь.

4.3. Параметры

имя

Имя (возможно, дополненное схемой) существующего анализатора текстового поиска.

новое_имя

Новое имя анализатора текстового поиска.

новая_схема

Новая схема анализатора текстового поиска.

4.4. Совместимость

Оператор `ALTER TEXT SEARCH PARSER` отсутствует в стандарте SQL.

5. ALTER TEXT SEARCH TEMPLATE

`ALTER TEXT SEARCH TEMPLATE` – изменить определение шаблона текстового поиска.

5.1. Синтаксис

`ALTER TEXT SEARCH TEMPLATE имя RENAME TO новое_имя`

`ALTER TEXT SEARCH TEMPLATE имя SET SCHEMA новая_схема`

5.2. Описание

`ALTER TEXT SEARCH TEMPLATE` изменяет определение шаблона текстового поиска. В настоящее время эта команда позволяет только сменить его имя.

Выполнить команду `ALTER TEXT SEARCH TEMPLATE` может только

суперпользователь.

5.3. Параметры

ИМЯ

Имя (возможно, дополненное схемой) существующего шаблона текстового поиска.

новое_имя

Новое имя шаблона текстового поиска.

новая_схема

Новая схема шаблона текстового поиска.

5.4. Совместимость

Оператор ALTER TEXT SEARCH TEMPLATE отсутствует в стандарте SQL.

6. ALTER TRIGGER

ALTER TRIGGER – изменить определение триггера.

6.1. Синтаксис

ALTER TRIGGER имя ON table_name RENAME TO новое_имя

6.2. Описание

ALTER TRIGGER меняет свойства существующего триггера. Предложение RENAME переименовывает данный триггер, не затрагивая его определение.

Изменять свойства триггера может только владелец таблицы, с которой работает триггер.

6.3. Параметры

ИМЯ

Имя существующего триггера, подлежащего изменению.

table_name

Имя таблицы, с которой работает триггер.

новое_имя

Новое имя триггера.

6.4. Замечания

Возможность временно включать или отключать триггер предоставляется командой ALTER TABLE, а не ALTER TRIGGER, так как ALTER TRIGGER не позволяет удобным образом выразить указание включить или отключить все триггеры таблицы сразу.

6.5. Примеры

Переименование существующего триггера:

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

6.6. Совместимость

ALTER TRIGGER – реализованное в Синергия-БД расширение стандарта SQL.

7. ALTER TYPE

ALTER TYPE – изменить определение типа.

7.1. Синтаксис

```
ALTER TYPE имя action [, ... ]  
ALTER TYPE имя OWNER TO новый_владелец  
ALTER TYPE имя RENAME ATTRIBUTE имя_атрибута TO  
новое_имя_атрибута [ CASCADE | RESTRICT ]  
ALTER TYPE имя RENAME TO новое_имя  
ALTER TYPE имя SET SCHEMA новая_схема  
ALTER TYPE имя ADD VALUE [ IF NOT EXISTS ]  
новое_значение_перечисления [ { BEFORE | AFTER }  
существующее_значение_перечисления ]
```

Где возможные варианты action:

```
ADD ATTRIBUTE имя_атрибута тип_данных [ COLLATE  
правило_сортировки ] [ CASCADE | RESTRICT ]  
DROP ATTRIBUTE [ IF EXISTS ] имя_атрибута [ CASCADE  
| RESTRICT ]  
ALTER ATTRIBUTE имя_атрибута [ SET DATA ] TYPE  
тип_данных [ COLLATE правило_сортировки ] [ CASCADE |  
RESTRICT ]
```

7.2. Описание

ALTER TYPE изменяет определение существующего типа. Эта команда имеет несколько разновидностей:

- ADD ATTRIBUTE

Эта форма добавляет в составной тип новый атрибут с тем же синтаксисом, что и CREATE TYPE;

- DROP ATTRIBUTE [IF EXISTS]

Эта форма удаляет атрибут из составного типа. Если указано IF EXISTS и атрибут не существует, это не считается ошибкой. В этом случае выдается только замечание;

- SET DATA TYPE

Эта форма меняет тип атрибута составного типа;

- OWNER

Эта форма меняет владельца типа;

- RENAME

Эта форма меняет имя типа или имя отдельного атрибута составного типа;

- SET SCHEMA

Эта форма переносит тип в другую схему;

- ADD VALUE [IF NOT EXISTS] [BEFORE | AFTER]

Эта форма добавляет новое значение в тип-перечисление. Порядок нового значения в перечислении можно указать, добавив BEFORE (перед) или AFTER (после) с одним из существующих значений. Если такое указание отсутствует, новый элемент добавляется в конец списка значений.

С указанием IF NOT EXISTS, если тип уже содержит новое значение, ошибки не произойдет: будет выдано замечание и ничего больше. Без этого указания, если такое значение уже представлено, возникнет ошибка;

- CASCADE

Автоматически распространять действие операции на типизированные таблицы, имеющий данный тип, и их потомки;

- RESTRICT

Отказать в выполнении операции, если изменяемый тип является типом типизированной таблицы. Это поведение по умолчанию.

Операции `ADD ATTRIBUTE`, `DROP ATTRIBUTE` и `ALTER ATTRIBUTE` можно объединить в один список множественных изменений для параллельного выполнения. Например, в одной команде можно добавить сразу несколько атрибутов и/или изменить тип нескольких атрибутов.

Выполнить `ALTER TYPE` может только владелец соответствующего типа. Чтобы сменить схему типа, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме типа. С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать тип. Однако суперпользователь может сменить владельца типа в любом случае. Чтобы добавить атрибут или изменить тип атрибута, также требуется иметь право `USAGE` для соответствующего типа данных.

7.3. Параметры

ИМЯ

Имя (возможно, дополненное схемой) существующего типа, подлежащего изменению.

новое_имя

Новое имя типа.

новый_владелец

Имя пользователя, назначаемого новым владельцем типа.

новая_схема

Новая схема типа.

имя_атрибута

Имя атрибута, подлежащего добавлению, изменению или удалению.

новое_имя_атрибута

Новое имя атрибута

тип_данных

Тип данных добавляемого атрибута, либо новый тип данных изменяемого атрибута.

новое_значение_перечисления

Новое значение добавляется в список значений перечисления. Как и все элементы перечисления, оно должно заключаться в кавычки.

существующее_значение_перечисления

Существующие значение в перечислении, непосредственно перед или после которого по порядку перечисления будет добавлено новое значение. Как и все элементы перечисления, оно должно заключаться в кавычки.

7.4. Замечания

ALTER TYPE ... ADD VALUE (форму, добавляющую в тип-перечисление новое значение) нельзя выполнять внутри блока транзакции.

Сравнения с добавленными значениями перечисления иногда бывают медленнее сравнений, в которых задействуются только начальные члены типа-перечисления. Обычно это происходит, только если BEFORE или AFTER устанавливает порядок нового элемента не в конце списка. Однако, иногда это наблюдается даже тогда, когда новое значение добавляется в конец списка (это происходит, если счетчик OID «прокручивается» с момента изначального создания типа-перечисления). Это замедление обычно несущественное, но если это важно, вернуть максимальную производительность можно, удалив и создав заново это перечисление, либо выгрузив копию базы данных и загрузив её вновь.

7.5. Примеры

Переименование типа данных:

```
ALTER TYPE electronic_mail RENAME TO email;
```

Смена владельца типа email на joe:

```
ALTER TYPE email OWNER TO joe;
```

Смена схемы типа email на customers:

```
ALTER TYPE email SET SCHEMA customers;
```

Добавление в тип нового атрибута:

```
ALTER TYPE compfoo ADD ATTRIBUTE f3 int;
```

Добавление нового значения в тип-перечисление, в определенное положение

по порядку:

```
ALTER TYPE colors ADD VALUE 'orange' AFTER 'red';
```

7.6. Совместимость

Формы команды, предназначенные для добавления и удаления атрибутов, являются частью стандарта SQL; другие формы относятся к расширениям Синергия-БД.

8. ALTER USER

ALTER USER – изменить роль в базе данных.

8.1. Синтаксис

```
ALTER USER имя [ [ WITH ] параметр [ ... ] ]
```

Здесь параметр:

```
SUPERUSER | NOSUPERUSER  
| CREATEDB | NOCREATEDB  
| CREATEROLE | NOCREATEROLE  
| CREATEUSER | NOCREATEUSER  
| INHERIT | NOINHERIT  
| LOGIN | NOLOGIN  
| REPLICATION | NOREPLICATION  
| CONNECTION LIMIT предел_подключений  
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'  
| VALID UNTIL 'дата_время'
```

```
ALTER USER имя RENAME TO новое_имя
```

```
ALTER USER имя SET параметр_конфигурации { TO | = }  
{ значение | DEFAULT }
```

```
ALTER USER имя SET параметр_конфигурации FROM CURRENT
```

```
ALTER USER имя RESET параметр_конфигурации
```

```
ALTER USER имя RESET ALL
```


8.2. Описание

Оператор `ALTER USER` теперь стал синонимом оператора `ALTER PROCEDURE`.

8.3. Совместимость

Оператор `ALTER USER` является расширением Синергия-БД. В стандарте SQL определение пользователей считается зависимым от реализации.

9. ALTER USER MAPPING

`ALTER USER MAPPING` – изменить определение сопоставления пользователей.

9.1. Синтаксис

```
ALTER USER MAPPING FOR { имя_пользователя | USER |  
CURRENT_USER | PUBLIC }  
    SERVER имя_сервера  
    OPTIONS ( [ ADD | SET | DROP ] параметр ['значение']  
    [, ... ] )
```

9.2. Описание

`ALTER USER MAPPING` изменяет определение сопоставления пользователей.

Владелец стороннего сервера может изменить сопоставление любых пользователей на этом сервере. Кроме того, пользователь может изменить сопоставление для своего собственного имени пользователя, если он наделен правом `USAGE` на данном сервере.

9.3. Параметры

имя_пользователя

Имя пользователя для сопоставления. Значения `CURRENT_USER` и `USER` соответствуют имени текущего пользователя. Значение `PUBLIC` соответствует именам всех текущих и будущих пользователей системы.

имя_сервера

Имя сервера, для которого меняется сопоставление пользователей.

```
OPTIONS ( [ ADD | SET | DROP ] параметр ['значение']
[, ... ] )
```

Эти формы меняют параметры сопоставления пользователей. Новые параметры переопределяют любые определенные ранее. Возможные операции с параметрами: ADD (добавить), SET (установить) и DROP (удалить). По умолчанию подразумевается ADD. Имена параметров должны быть уникальными; кроме того, они проверяются оберткой сторонних данных.

9.4. Примеры

Изменение пароля в сопоставлении пользователя bob на сервере foo:

```
ALTER USER MAPPING FOR bob SERVER foo OPTIONS (user
'bob', password 'public');
```

9.5. Совместимость

ALTER USER MAPPING соответствует стандарту ISO/IEC 9075-9 (SQL/MED). Однако есть небольшое синтаксическое различие: в стандарте ключевое слово FOR опускается. Но так как и в CREATE USER MAPPING, и в DROP USER MAPPING слово FOR находится в аналогичных позициях, а IBM DB2 (еще одна популярная реализация SQL/MED) требует его и для ALTER USER MAPPING, Синергия-БД в этом аспекте отклоняется от стандарта ради согласованности и совместимости.

10. ALTER VIEW

ALTER VIEW – изменить определение представления.

10.1. Синтаксис

```
ALTER VIEW [ IF EXISTS ] имя ALTER [ COLUMN ]
имя_колонки SET DEFAULT выражение
ALTER VIEW [ IF EXISTS ] имя ALTER [ COLUMN ]
имя_колонки DROP DEFAULT
ALTER VIEW [ IF EXISTS ] имя OWNER TO новый_владелец
ALTER VIEW [ IF EXISTS ] имя RENAME TO новое_имя
ALTER VIEW [ IF EXISTS ] имя SET SCHEMA новая_схема
ALTER VIEW [ IF EXISTS ] имя SET
( имя_параметра_представления [=
```

```
значение_параметра_представления] [, ... ] )  
ALTER VIEW [ IF EXISTS ] имя RESET  
( имя_параметра_представления [, ... ] )
```

10.2.Описание

ALTER VIEW изменяет различные дополнительные свойства представления. Для изменения запроса, определяющего представление, используйте команду CREATE OR REPLACE VIEW.

Выполнить ALTER VIEW может только владелец представления. Чтобы сменить схему представления, необходимо также иметь право CREATE в новой схеме. Чтобы сменить владельца, требуется также быть непосредственным или опосредованным членом новой роли, а эта роль должна иметь право CREATE в схеме представления. С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать представление. Однако суперпользователь может сменить владельца представления в любом случае.

10.3.Параметры

имя

Имя (возможно, дополненное схемой) существующего представления.

IF EXISTS

Не считать ошибкой, если представление не существует. В этом случае будет выдано замечание.

SET/DROP DEFAULT

Эти формы устанавливают или удаляют значение по умолчанию в заданной колонке. Значение по умолчанию подставляется в команды INSERT и UPDATE, вносящие данные в представление, до применения каких-либо правил или триггеров в этом представлении. Таким образом, значения по умолчанию в представлении имеют приоритет перед значениями по умолчанию в нижележащих отношениях.

новый_владелец

Имя пользователя, назначаемого новым владельцем представления.

новое_имя

Новое имя представления.

новая_схема

Новая схема представления.

```
SET ( имя_параметра_представления [=
значение_параметра_представления] [, ... ] )
RESET ( имя_параметра_представления [, ... ] )
```

Устанавливает или сбрасывает параметры представления. В настоящее время поддерживаются параметры:

check_option (string)

Изменяет параметр проверки представления. Допустимые значения: local (локальная) или cascaded (каскадная).

security_barrier (boolean)

Изменяет свойство представления, включающее барьер безопасности. Значение должно быть логическим: true или false.

10.4.Замечания

По историческим причинам команду ALTER TABLE можно использовать и с представлениями; но единственно допустимые для работы с представлениями вариации ALTER TABLE равносильны вышеперечисленным командам.

10.5.Примеры

Переименование представления foo в bar:

```
ALTER VIEW foo RENAME TO bar;
```

Добавление значения колонки по умолчанию в изменяемое представление:

```
CREATE TABLE base_table (id int, ts timestamptz);
CREATE VIEW a_view AS SELECT * FROM base_table;
ALTER VIEW a_view ALTER COLUMN ts SET DEFAULT now();
INSERT INTO base_table(id) VALUES(1); -- в ts окажется
значение NULL
INSERT INTO a_view(id) VALUES(2); -- в ts окажется
текущее время
```

10.6.Совместимость

ALTER VIEW – реализованное в Синергия-БД расширение стандарта SQL.

11. ANALYZE

ANALYZE – собрать статистику по базе данных.

11.1.Синтаксис

```
ANALYZE [ VERBOSE ] [ table_name [ ( имя_колонки  
[, ...] ) ] ]
```

11.2.Описание

ANALYZE собирает статистическую информацию о содержимом таблиц в базе данных и сохраняет результаты в системном каталоге pg_statistic. Впоследствии планировщик запросов будет использовать эту статистику для выбора наиболее эффективных планов выполнения запросов.

Без параметров команда ANALYZE анализирует все таблицы в текущей базе данных. Если в параметрах передано имя таблицы, ANALYZE обрабатывает только заданную таблицу. Также эта команда принимает список имен колонок, что позволяет запустить сбор статистики только по этим колонкам.

11.3.Параметры

VERBOSE

Включает вывод сообщений о процессе выполнения.

table_name

Имя (возможно, дополненное схемой) конкретной таблицы, подлежащей анализу. Если опущено, анализироваться будут все обычные (не сторонние) таблицы в текущей базе данных.

имя_колонки

Имя колонки, подлежащей анализу. По умолчанию анализируются все колонки.

11.4.Выводимая информация

С указанием `VERBOSE` команда `ANALYZE` выдает сообщения о процессе анализа, отмечая текущую обрабатываемую таблицу. Также она выводит различные статистические сведения о таблицах.

11.5.Замечания

Сторонние таблицы анализируются только при явном указании и только если соответствующая обертка сторонних данных поддерживает команду `ANALYZE`. Если эта команда не поддерживается, при выполнении `ANALYZE` выводится предупреждение и больше ничего не происходит.

В стандартной конфигурации Синергия-БД работающий демон автоочистки запускает анализ таблиц автоматически, когда они изначально заполняются данными, и периодически, по мере того, как они меняются. Если автоочистка отключена, рекомендуется запускать `ANALYZE` время от времени, либо после кардинальных изменений в таблице. Точная статистика помогает планировщику выбрать наиболее эффективный план запроса и тем самым увеличивает скорость выполнения запроса. Обычно для баз, где данные в основном читаются, выполняют `VACUUM` и `ANALYZE` раз в день, во время наименьшей активности. Этого будет недостаточно, если данные меняются очень активно.

`ANALYZE` запрашивает для целевой таблицы блокировку только на чтение, так что эта команда может выполняться параллельно с другими операциями с таблицей.

Статистика, собираемая командой `ANALYZE`, обычно включает список из нескольких самых частых значений в каждой колонке и гистограмму, отражающую примерное распределение данных во всех колонках. Один или оба этих элемента статистики могут быть опущены, если `ANALYZE` сочтет их неинтересными (например, в колонке уникального ключа нет повторяющихся значений), либо если тип данных колонки не поддерживает соответствующие операторы.

В больших таблицах `ANALYZE` не просматривает все строки, а обрабатывает только небольшую случайную выборку. Это позволяет проанализировать за короткое время даже очень большие таблицы. Однако учтите, что такая статистика будет лишь

приблизительной и может немного меняться при каждом выполнении `ANALYZE`, даже если фактическое содержимое таблицы остается неизменным. Это может приводить к небольшим изменениям в оценках стоимости запросов, выводимых командой `EXPLAIN`. В редких случаях вследствие этой недетерминированности планировщик меняет свой выбор после выполнения `ANALYZE`. Чтобы избежать этого, увеличьте объем статистики, собираемой командой `ANALYZE`, как описано ниже.

Количеством статистики можно управлять, настраивая конфигурационную переменную `default_statistics_target` или устанавливая ориентир статистики на уровне колонок командой `ALTER TABLE ... ALTER COLUMN ... SET STATISTICS`. Ориентир задает максимальное число записей в списке наиболее распространенных значений и максимальное число интервалов в гистограмме. По умолчанию значение ориентира равно 100, но его можно увеличить или уменьшить в поисках баланса между точностью оценок планировщика и временем, требующимся для выполнения `ANALYZE`, а также объемом статистики в таблице `pg_statistic`. Если установить ориентир статистики равным нулю, статистика по таким колонкам собираться не будет. Это может быть полезно для колонок, которые никогда не фигурируют в предложениях `WHERE`, `GROUP BY` и `ORDER BY`, так как планировщик никогда не будет использовать их статистику.

Число строк таблицы, выбираемых для подготовки статистики, определяется наибольшим ориентиром статистики по всем анализируемым колонкам этой таблицы. Увеличение ориентира приводит к пропорциональному увеличению времени и пространства, требуемого для выполнения `ANALYZE`.

Одним из показателей, оцениваемых командой `ANALYZE`, является число различных значений, встречающихся в каждой колонке. Так как рассматривается только подмножество всех строк, эта оценка иногда может быть весьма неточной, даже при самых больших ориентирах статистики. Если эта неточность приводит к плохому выбору плана запроса, более точное значение можно определить вручную и затем задать его командой `ALTER TABLE ... ALTER COLUMN ... SET`

```
(n_distinct = ...).
```

Если у анализируемой таблицы есть один или несколько потомков, ANALYZE соберет статистику дважды: сначала по строкам только родительской таблицы, а затем по строкам родительской и всех дочерних таблиц. Второй набор статистики необходим для планирования запросов, обращающихся ко всему дереву наследования. Демон автоочистки, однако, принимая решение об автоматическом запуске анализа, будет учитывать операции добавления или изменения данных только в самой родительской таблице. Если именно в этой таблице изменение и добавление происходит редко, наследуемая статистика может терять актуальность, если не запускать ANALYZE вручную.

Если анализируемая таблица оказалась пустой, ANALYZE не будет обновлять статистику по этой таблице; в базе сохранится статистика, собранная ранее.

11.6.Совместимость

Оператор ANALYZE отсутствует в стандарте SQL.

12. BEGIN

BEGIN – начать блок транзакции.

12.1.Синтаксис

```
BEGIN [ WORK | TRANSACTION ] [ режим_транзакции  
[, ...] ]
```

Где возможные варианты режим_транзакции:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ |  
READ COMMITTED | READ UNCOMMITTED }  
READ WRITE | READ ONLY  
[ NOT ] DEFERRABLE
```

12.2.Описание

BEGIN начинает блок транзакции, то есть обозначает, что все операторы после команды BEGIN и до явной команды COMMIT или ROLBACK будут выполняться в одной транзакции. По умолчанию (без BEGIN) Синергия-БД выполняет транзакции

в режиме «autocommit» (автофиксация), то есть каждый оператор выполняется в своей отдельной транзакции, которая неявно фиксируется в конце оператора (если оператор был выполнен успешно; в противном случае, транзакция откатывается).

В блоке транзакции операторы выполняются быстрее, так как для запуска/фиксации транзакции производится масса операций, нагружающих процессор и диск. Кроме того, выполнение нескольких операторов в одной транзакции позволяет обеспечить целостность при внесении серии связанных изменений; другие сеансы не видят промежуточное состояние, когда произошли ещё не все связанные изменения.

Если указан уровень изоляции, режим чтения/записи или устанавливается отложенный режим, новая транзакция получает те же характеристики, что и после выполнения SET TRANSACTION.

12.3.Параметры

WORK
TRANSACTION

Необязательные ключевые слова, не оказывают никакого влияния.

12.4.Замечания

START TRANSACTION делает то же, что и BEGIN.

Для завершения блока транзакции используйте COMMIT или ROLBACK.

При попытке выполнить BEGIN внутри уже начатого блока транзакции будет выдано предупреждение, а состояние транзакции не изменится. Для вложения подтранзакций внутри блока транзакций используйте точки сохранения.

Для сохранения обратной совместимости допускается перечисление режимов_транзакции без запятых.

12.5.Примеры

Начало блока транзакции:

BEGIN;

12.6.Совместимость

BEGIN – это языковое расширение Синергия-БД. Эта команда равнозначна соответствующей стандарту SQL команде START TRANSACTION, на справочной странице которой можно найти дополнительные сведения о совместимости.

Значение DEFERRABLE параметра режим_транзакции является языковым расширением Синергия-БД.

По стечению обстоятельств ключевое слово BEGIN имеет другое значение во встраиваемом SQL, поэтому при портировании приложений баз данных рекомендуется внимательно сверить семантику транзакций.

13. CHECKPOINT

CHECKPOINT – записать контрольную точку в журнал транзакций.

13.1.Синтаксис

CHECKPOINT

13.2.Описание

Контрольная точка – это состояние в журнале транзакций, в котором все файлы данных содержат актуальные записи, в соответствии с информацией в журнале. При этом все файлы данных сохраняются на диск.

Команда CHECKPOINT приводит к принудительному выполнению контрольной точки в момент вызова, не дожидаясь периодической контрольной точки, производимой системой по графику. Команда CHECKPOINT не предназначена для применения при обычном использовании базы данных.

Если CHECKPOINT выполняется в процессе восстановления, вместо записи новой контрольной точки производится точка перезапуска.

Выполнять CHECKPOINT могут только суперпользователи.

13.3.Совместимость

Команда CHECKPOINT является языковым расширением Синергия-БД.

14. CLOSE

CLOSE – закрывает курсор.

14.1.Синтаксис

```
CLOSE { имя | ALL }
```

14.2.Описание

CLOSE освобождает ресурсы, связанные с открытым курсором. Когда курсор закрыт, никакие операции с ним невозможны. Закрывать курсор следует, когда он становится ненужным.

Все не удерживаемые открытые курсоры закрываются неявно при завершении транзакции командами COMMIT или ROLLBACK. Удерживаемый курсор закрывается неявно, если транзакция, его создавшая, прерывается командой ROLLBACK. Если создавшая его транзакция завершается успешной фиксацией, удерживаемый курсор остается открытым до явного вызова команды CLOSE или отключения клиента.

14.3.Параметры

ИМЯ

Имя открытого курсора, который будет закрыт.

ALL

Закрывает все открытые курсоры.

14.4.Замечания

В Синергия-БД нет явной команды OPEN для курсора; курсор считается открытым при объявлении. Чтобы объявить курсор, используйте оператор DECLARE.

Получить список всех доступных курсоров можно, обратившись к системному представлению pg_cursors.

Если курсор был закрыт после точки сохранения, а затем произошел откат к этой точке, действие команды CLOSE не отменяется; то есть курсор остается закрытым.

14.5.Примеры

Следующая команда закрывает курсор `liahona`:

```
CLOSE liahona;
```

14.6.Совместимость

Оператор `CLOSE` полностью соответствует стандарту SQL. `CLOSE ALL` является расширением Синергия-БД.

15. CLUSTER

`CLUSTER` – кластеризовать таблицу согласно индексу.

15.1.Синтаксис

```
CLUSTER [VERBOSE] table_name [ USING имя_индекса ]  
CLUSTER [VERBOSE]
```

15.2.Описание

Оператор `CLUSTER` указывает Синергия-БД кластеризовать таблицу, заданную параметром `имя_таблицы`, согласно индексу, заданному параметром `имя_индекса`. Указанный индекс уже должен быть определен в таблице `имя_таблицы`.

В результате кластеризации таблицы её содержимое физически переупорядочивается в зависимости от индекса. Кластеризация является одноразовой операцией: последующие изменения в таблице нарушают порядок кластеризации. Другими словами, система не пытается автоматически сохранять порядок новых или измененных строк в соответствии с индексом. Если такое желание возникает, можно периодически повторять кластеризацию, выполняя команду снова. Кроме того, если для заданной таблицы установить параметр `FILLFACTOR` меньше 100%, это может помочь сохранить порядок кластеризации при изменениях, так как изменяемые строки будут помещаться в ту же страницу, если в ней достаточно места.

Когда таблица кластеризована, Синергия-БД запоминает, по какому именно индексу. Форма `CLUSTER имя_таблицы` повторно кластеризует таблицу по тому

же индексу. Для установки индекса, который будет использоваться для будущих операций кластеризации, или очистки предыдущего значения можно также применить команду `CLUSTER` или формы `SET WITHOUT CLUSTER` команды `ALTER TABLE`.

`CLUSTER` без параметров повторно кластеризует все ранее кластеризованные таблицы в текущей базе данных, принадлежащие пользователю, вызывающему команду, или все такие таблицы, если её вызывает суперпользователь. Эту форму `CLUSTER` нельзя выполнять внутри блока транзакции.

В процессе кластеризации таблицы для нее запрашивается блокировка `ACCESS EXCLUSIVE`. Это препятствует выполнению всех других операций (чтению и записи) с таблицей до завершения `CLUSTER`.

15.3.Параметры

`table_name`

Имя (возможно, дополненное схемой) таблицы.

`имя_индекса`

Имя индекса.

`VERBOSE`

Выводит отчет о процессе кластеризации по мере обработки таблиц.

15.4.Замечания

В случаях, когда происходит обращение к случайным единичным строкам таблицы, фактический порядок данных в этой таблице не важен. Но если обращения к одним данным происходят чаще, чем к другим, и есть индекс, которые собирает их вместе, применение команды `CLUSTER` может быть полезным. Например, когда из таблицы запрашивается диапазон индексированных значений, либо одно индексированное значение, которому соответствуют несколько строк, `CLUSTER` может помочь, так как страница таблицы, найденная по индексу для первой искомой строки, скорее всего, будет содержать и все остальные искомые строки. Таким образом, кластеризация помогает оптимизировать обращения к диску и ускорить запросы.

CLUSTER может переупорядочить таблицу, выполнив либо сканирование указанного индекса, либо (для индексов В-дерева) последовательное сканирование, а затем сортировку. Наилучший по скорости вариант будет выбран, исходя из имеющейся статистической информации и параметров планировщика.

Когда выбирается сканирование индекса, создается временная таблица, содержащая данные целевой таблицы по порядку индекса. Также создаются копии всех индексов таблицы. Таким образом, для этой операции требуется объем дискового пространства не меньше, чем размер таблицы и индексов в сумме.

В случае выбора последовательного сканирования и сортировки создается ещё и временный файл для сортировки, так что пиковым требованием будет удвоенный размер таблицы плюс размер индексов. Этот метод часто быстрее, чем сканирование по индексу, но если требование к дисковому пространству неприемлемо, можно отключить его выбор, временно установив `enable_sort` в `off`.

Перед кластеризацией рекомендуется установить в `maintenance_work_mem` достаточно большое значение (но не больше, чем объем ОЗУ, который вы хотите выделить для операции CLUSTER).

Так как планировщик записывает статистику, связанную с порядком таблиц, для вновь кластеризуемых таблиц рекомендуется запускать ANALYZE. В противном случае планировщик может ошибиться с выбором плана запроса.

Так как CLUSTER запоминает, по каким индексам кластеризованы таблицы, достаточно лишь один раз вручную кластеризовать нужные таблицы, а затем настроить периодический скрипт обслуживания, который будет выполнять CLUSTER без параметров, с тем чтобы эти таблицы регулярно кластеризовались.

15.5.Примеры

Кластеризация таблицы `employees` согласно её индексу `employees_ind`:

```
CLUSTER employees USING employees_ind;
```

Кластеризация таблицы `employees` согласно тому же индексу, что был использован ранее:

```
CLUSTER employees;
```

Кластеризация всех таблиц в базе данных, что были кластеризованы ранее:

CLUSTER;

15.6. Совместимость

Оператор CLUSTER отсутствует в стандарте SQL.

16. COMMENT

COMMENT – задать или изменить комментарий объекта.

16.1. Синтаксис

COMMENT ON

```
{  
    AGGREGATE имя_агрегатной_функции  
( сигнатура_агр_функции ) |  
    CAST (исходный_тип AS целевой_тип) |  
    COLLATION имя_объекта |  
    COLUMN имя_отношения.имя_колонки |  
    CONSTRAINT имя_ограничения ON table_name |  
    CONVERSION имя_объекта |  
    DATABASE имя_объекта |  
    DOMAIN имя_объекта |  
    EXTENSION имя_объекта |  
    EVENT TRIGGER имя_объекта |  
    FOREIGN DATA WRAPPER имя_объекта |  
    FOREIGN TABLE имя_объекта |  
    FUNCTION имя_функции ( [ [ режим_аргумента ]  
[ имя_аргумента ] тип_аргумента [, ...] ] ) |  
    INDEX имя_объекта |  
    LARGE OBJECT oid_большого_объекта |  
    MATERIALIZED VIEW имя_объекта |  
    OPERATOR имя_оператора (тип_слева, тип_справа) |  
    OPERATOR CLASS имя_объекта USING метод_индекса |  
    OPERATOR FAMILY имя_объекта USING метод_индекса |  
    [ PROCEDURAL ] LANGUAGE имя_объекта |
```

```

ROLE имя_объекта |
RULE имя_правила ON table_name |
SCHEMA имя_объекта |
SEQUENCE имя_объекта |
SERVER имя_объекта |
TABLE имя_объекта |
TABLESPACE имя_объекта |
TEXT SEARCH CONFIGURATION имя_объекта |
TEXT SEARCH DICTIONARY имя_объекта |
TEXT SEARCH PARSER имя_объекта |
TEXT SEARCH TEMPLATE имя_объекта |
TRIGGER имя_триггера ON table_name |
TYPE имя_объекта |
VIEW имя_объекта
} IS 'текст'

```

Здесь сигнатура_агр_функции:

```

* |
[ режим_аргумента ] [ имя_аргумента ] тип_аргумента
[ , ... ] |
[ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
[ , ... ] ] ORDER BY [ режим_аргумента ]
[ имя_аргумента ] тип_аргумента [ , ... ]

```

16.2.Описание

COMMENT сохраняет комментарий об объекте базы данных.

Для каждого объекта сохраняется только одна строка, так что для изменения комментария нужно просто выполнить COMMENT ещё раз для того же объекта. Чтобы удалить комментарий, вместо текстовой строки укажите NULL. При удалении объектов комментарии удаляются автоматически.

Для большинства типов объектов комментарий может установить только владелец объекта. Но так как роли не имеют владельцев, COMMENT ON ROLE для ролей суперпользователей разрешено выполнять только суперпользователям, а для обычных ролей – тем, кто имеет право CREATEROLE. Разумеется,

суперпользователи могут задавать комментарии для любых объектов.

Просмотреть комментарии можно в psql, используя семейство команд \d. Имеется возможность получать комментарии и в других пользовательских интерфейсах, используя те же встроенные функции, что использует psql, а именно obj_description, col_description и shobj_description.

16.3.Параметры

имя_объекта
имя_отношения.имя_колонки
имя_агрегатной_функции
имя_ограничения
имя_функции
имя_оператора
имя_правила
имя_триггера

Имя объекта, для которого задается комментарий. Имена таблиц, агрегатных функций, правил сортировки, перекодировок, доменов, сторонних таблиц, функций, индексов, операторов, классов и семейств операторов, последовательностей, объектов текстового поиска, типов и представлений могут быть дополнены именем схемы. При определении комментария для колонки, имя_отношения должно ссылаться на таблицу, представление, составной тип или стороннюю таблицу.

исходный_тип

Имя исходного типа данных для приведения.

целевой_тип

Имя целевого типа данных для приведения.

режим_аргумента

Режим аргумента обычной или агрегатной функции: IN, OUT, INOUT или VARIADIC. По умолчанию подразумевается IN. Заметьте, что COMMENT на самом деле не учитывает аргументы OUT, так как для идентификации функции нужны только типы входных аргументов. Поэтому достаточно перечислить только аргументы IN, INOUT и VARIADIC.

имя_аргумента

Имя аргумента обычной или агрегатной функции. Заметьте, что на самом деле

COMMENT не обращает внимание на имена аргументов, так как для однозначной идентификации функции достаточно только типов аргументов.

тип_аргумента

Тип данных аргумента обычной или агрегатной функции.

oid_большого_объекта

OID большого объекта.

тип_слева

тип_справа

Тип данных аргументов оператора (возможно, дополненный именем схемы). В случае отсутствия аргумента префиксного или постфиксного оператора укажите вместо типа NONE.

PROCEDURAL

Это слово не несет смысловой нагрузки.

текст

Новый комментарий, записанный в виде строковой константы (или NULL для удаления комментария).

16.4.Замечания

В настоящее время механизм безопасности в части просмотра комментариев отсутствует: любой пользователь, подключенный к базе данных, может видеть все комментарии всех объектов базы. Для общих объектов, таких как базы данных, роли и табличные пространства, комментарии хранятся глобально, так что их может видеть любой пользователь, подключенный к любой базе данных в кластере. Поэтому ничего секретного писать в комментариях не следует.

16.5.Примеры

Добавление комментария для таблицы mytable:

```
COMMENT ON TABLE mytable IS 'Это моя таблица.';
```

Удаление его:

```
COMMENT ON TABLE mytable IS NULL;
```

Ещё несколько примеров:

```
COMMENT ON AGGREGATE my_aggregate (double precision) IS
```

```
'Вычисляет дисперсию выборки';  
COMMENT ON CAST (text AS int4) IS 'Конвертирует строки в  
int4';  
COMMENT ON COLLATION «fr_CA» IS 'Канадский французский';  
COMMENT ON COLUMN my_table.my_column IS 'Порядковый  
номер сотрудника';  
COMMENT ON CONVERSION my_conv IS 'Перекодировка в UTF8';  
COMMENT ON CONSTRAINT bar_col_cons ON bar IS  
'Ограничение колонки col';  
COMMENT ON DATABASE my_database IS 'База данных  
разработчиков';  
COMMENT ON DOMAIN my_domain IS 'Домен почтового адреса';  
COMMENT ON EXTENSION hstore IS 'Реализует тип данных  
hstore';  
COMMENT ON FOREIGN DATA WRAPPER mywrapper IS 'моя  
обертка сторонних данных';  
COMMENT ON FOREIGN TABLE my_foreign_table IS 'Информация  
о сотрудниках в другой БД';  
COMMENT ON FUNCTION my_function (timestamp) IS  
'Возвращает число римскими цифрами';  
COMMENT ON INDEX my_index IS 'Обеспечивает уникальность  
по коду сотрудника';  
COMMENT ON LANGUAGE plpython IS 'Поддержка Python для  
храняемых процедур';  
COMMENT ON LARGE OBJECT 346344 IS 'Документ  
планирования';  
COMMENT ON MATERIALIZED VIEW my_matview IS 'Сводка  
истории заказов';  
COMMENT ON OPERATOR ^ (text, text) IS 'Вычисляет  
пересечение двух текстов';  
COMMENT ON OPERATOR - (NONE, integer) IS 'Унарный  
минус';  
COMMENT ON OPERATOR CLASS int4ops USING btree IS  
'Операторы для четырехбайтовых целых (для B-деревьев)';  
COMMENT ON OPERATOR FAMILY integer_ops USING btree IS  
'Все целочисленные операторы (для B-деревьев)';  
COMMENT ON ROLE my_role IS 'Административная группа для  
таблиц бухгалтерии';
```

```
COMMENT ON RULE my_rule ON my_table IS 'Протоколирует
изменения в записях сотрудников';
COMMENT ON SCHEMA my_schema IS 'Данные отдела';
COMMENT ON SEQUENCE my_sequence IS 'Предназначена для
генерации первичных ключей';
COMMENT ON SERVER myserver IS 'Мой сторонний сервер';
COMMENT ON TABLE my_schema.my_table IS 'Данные
сотрудников';
COMMENT ON TABLESPACE my_tablespace IS 'Табличное
пространство для индексов';
COMMENT ON TEXT SEARCH CONFIGURATION my_config IS
'Фильтрация специальных слов';
COMMENT ON TEXT SEARCH DICTIONARY swedish IS 'Стеммер
Snowball для шведского языка';
COMMENT ON TEXT SEARCH PARSER my_parser IS 'Разделяет
текст на слова';
COMMENT ON TEXT SEARCH TEMPLATE snowball IS 'Стеммер
Snowball';
COMMENT ON TRIGGER my_trigger ON my_table IS
'Обеспечивает ссылочную целостность';
COMMENT ON TYPE complex IS 'Тип данных комплексных
чисел';
COMMENT ON VIEW my_view IS 'Представление расходов по
отделам';
```

16.6.Совместимость

Оператор COMMENT отсутствует в стандарте SQL.

17. COMMIT

COMMIT – зафиксировать текущую транзакцию.

17.1.Синтаксис

```
COMMIT [ WORK | TRANSACTION ]
```

17.2.Описание

COMMIT фиксирует текущую транзакцию. Все изменения, произведенные транзакцией, становятся видимыми для других и гарантированно сохраняются в

случае сбоя.

17.3.Параметры

WORK
TRANSACTION

Необязательные ключевые слова, не оказывают никакого влияния.

17.4.Замечания

Для прерывания транзакции используйте ROLLBACK.

При попытке выполнить COMMIT вне транзакции ничего не произойдет, но будет выдано предупреждение.

17.5.Примеры

Следующая команда фиксирует текущую транзакцию и сохраняет все изменения:

COMMIT ;

17.6.Совместимость

В стандарте SQL описаны только две формы: COMMIT и COMMIT WORK. В остальном эта команда полностью соответствует стандарту.

18. COMMIT PREPARED

COMMIT PREPARED – зафиксировать транзакцию, которая ранее была подготовлена для двухфазной фиксации.

18.1.Синтаксис

COMMIT PREPARED id_транзакции

18.2.Описание

COMMIT PREPARED фиксирует транзакцию, находящуюся в подготовленном состоянии.

18.3.Параметры

id_транзакции

Идентификатор транзакции, которая будет зафиксирована.

18.4.Замечания

Зафиксировать подготовленную транзакцию может либо пользователь, выполнявший её изначально, либо суперпользователь. При этом не обязательно работать в том же сеансе, где выполнялась транзакция.

Эту команду нельзя выполнить внутри блока транзакции. Подготовленная транзакция фиксируется немедленно.

Все существующие в текущий момент подготовленные транзакции показываются в системном представлении `pg_prepared_xacts`.

18.5.Примеры

Фиксация транзакции, имеющей идентификатор `foobar`:

```
COMMIT PREPARED 'foobar';
```

18.6.Совместимость

Оператор `COMMIT PREPARED` является расширением Синергия-БД. Он предназначен для использования внешними системами управления транзакциями, некоторые из которых работают по стандартам (например, X/Open XA), но сторона SQL в этих системах не стандартизирована.

19. COPY

`COPY` – копировать данные между файлом и таблицей.

19.1.Синтаксис

```
COPY table_name [ ( имя_колонки [, ...] ) ]  
FROM { 'имя_файла' | PROGRAM 'command' | STDIN }  
[ [ WITH ] ( параметр [, ...] ) ]
```

```
COPY { table_name [ ( имя_колонки [, ...] ) ] |  
( запрос ) }  
TO { 'имя_файла' | PROGRAM 'command' | STDOUT }  
[ [ WITH ] ( параметр [, ...] ) ]
```

Здесь допускается параметр:

```
FORMAT имя_формата
OIDS [ boolean ]
FREEZE [ boolean ]
DELIMITER 'символ_разделитель'
NULL 'маркер_NULL'
HEADER [ boolean ]
QUOTE 'символ_кавычек'
ESCAPE 'символ_экранирования'
FORCE_QUOTE { ( имя_колонки [, ...] ) | * }
FORCE_NOT_NULL ( имя_колонки [, ...] )
FORCE_NULL ( имя_колонки [, ...] )
ENCODING 'имя_кодировки'
```

19.2.Описание

COPY перемещает данные между таблицами Синергия-БД и традиционными файлами. COPY TO копирует содержимое таблицы в файл, а COPY FROM – из файла в таблицу (добавляет данные к тем, что уже содержались в таблице). COPY TO может также скопировать результаты запроса SELECT.

Если указывается список колонок, COPY скопирует в или из файла только данные указанных колонок. Если в таблице есть колонки, отсутствующие в этом списке, COPY FROM заполнит эти колонки значениями по умолчанию.

COPY с именем файла указывает серверу Синергия-БД читать или записывать непосредственно этот файл. Заданный файл должен быть доступен пользователю Синергия-БД (тому пользователю, от имени которого работает сервер) и это имя должно быть определено с точки зрения сервера. Когда указывается параметр PROGRAM, сервер выполняет заданную команду и читает данные из стандартного вывода программы, либо записывает их в стандартный ввод. Команда должна определяться с точки зрения сервера и быть доступной для исполнения пользователю Синергия-БД. Когда указывается STDIN или STDOUT, данные передаются через соединение клиента с сервером.

19.3.Параметры

`table_name`

Имя (возможно, дополненное схемой) существующей таблицы.

`имя_колонки`

Необязательный список колонок, данные которых будут копироваться. Если этот список отсутствует, копируются все колонки таблицы.

`запрос`

Команда `SELECT` или `VALUES`, результаты которой будут скопированы. Заметьте, что запрос должен заключаться в скобки.

`имя_файла`

Путь входного или выходного файла. Путь входного файла может быть абсолютным или относительным, но путь выходного должен быть только абсолютным.

`PROGRAM`

Выполняемая команда. `COPY FROM` читает стандартный вывод команды, а `COPY TO` записывает в её стандартный ввод.

Заметьте, что команда запускается через командную оболочку, так что если требуется передать этой команде какие-либо аргументы, поступающие из недоверенного источника, необходимо аккуратно избавиться от всех спецсимволов, имеющих особое значение в оболочке, либо экранировать их. По соображениям безопасности лучше ограничиться фиксированной строкой команды или как минимум не позволять пользователям вводить в нее произвольное содержимое.

`STDIN`

Указывает, что данные будут поступать из клиентского приложения.

`STDOUT`

Указывает, что данные будут выдаваться клиентскому приложению.

`boolean`

Включает или отключает заданный параметр. Для включения параметра можно написать `TRUE`, `ON` или `1`, а для отключения – `FALSE`, `OFF` или `0`. Значение `boolean` можно опустить, в этом случае подразумевается `TRUE`.

FORMAT

Выбирает формат чтения или записи данных: `text` (текстовый), `csv` (значения, разделенные запятыми, Comma Separated Values) или `binary` (двоичный). По умолчанию выбирается формат `text`.

OIDS

Копирует OID каждой строки. Если присутствует указание `OIDS`, но таблица не содержит колонки `oid`, либо копируется запрос, возникнет ошибка.

FREEZE

Запросы копируют данные с уже замороженными строками, как после выполнения команды `VACUUM FREEZE`. Это позволяет увеличить производительность при начальном добавлении данных. Строки будут замораживаться, только если загружаемая таблица была создана или опустошена в текущей подтранзакции, с ней не связаны открытые курсоры и в данной транзакции нет других снимков.

Заметьте, что все другие сеансы будут немедленно видеть данные, как только они будут успешно загружены. Это нарушает принятые правила видимости MVCC, так что пользователи, включающие этот режим, должны понимать, какие проблемы это может вызвать.

DELIMITER

Задаёт символ, разделяющий колонки в строках файла. По умолчанию это символ табуляции в текстовом формате и запятая в формате CSV. Задаваемый символ должен быть однобайтовым. Для формата `binary` этот параметр не допускается.

NULL

Определяет строку, задающую значение `NULL`. По умолчанию в текстовом формате это `\N` (обратная косая черта и N), а в формате CSV – пустая строка без кавычек. Пустую строку можно использовать и в текстовом формате, если не требуется различать пустые строки и `NULL`. Для формата `binary` этот параметр не допускается.

Примечание. При выполнении `COPY FROM` любые значения, совпадающие с

этой строкой, сохраняются как значение `NULL`, так что при переносе данных важно убедиться в том, что это та же строка, что применялась в `COPY TO`.

HEADER

Указывает, что файл содержит строку заголовка с именами колонок. При выводе первая строка файла будет содержать имена колонок таблицы, а при вводе первая строка просто игнорируется. Этот параметр допускается только для формата `CSV`.

QUOTE

Указывает символ кавычек, используемый для заключения данных в кавычки. По умолчанию это символ двойных кавычек. Задаваемый символ должен быть однобайтовым. Этот параметр поддерживается только для формата `CSV`.

ESCAPE

Задаёт символ, который будет выводиться перед символом данных, совпавшим со значением `QUOTE`. По умолчанию это тот же символ, что и `QUOTE` (то есть, при появлении в данных кавычек, они дублируются). Задаваемый символ должен быть однобайтовым. Этот параметр допускается только для режима `CSV`.

FORCE_QUOTE

Принудительно заключает в кавычки все значения не `NULL` в указанных колонках. Выводимое значение `NULL` никогда не заключается в кавычки. Если указано `*`, в кавычки будут заключаться значения не `NULL` во всех колонках. Этот параметр принимает только команда `COPY TO` и только для формата `CSV`.

FORCE_NOT_NULL

Не сопоставлять значения в указанных колонках с маркером `NULL`. По умолчанию, когда маркер пуст, это означает, что пустые значения будут считаны как строки нулевой длины, а не `NULL`, даже когда они не заключены в кавычки. Этот параметр допускается только в команде `COPY FROM` и только для формата `CSV`.

FORCE_NULL

Сопоставлять значения в указанных колонках с маркером `NULL`, даже если они заключены в кавычки, и в случае совпадения устанавливать значение `NULL`. По умолчанию, когда этот маркер пуст, пустая строка в кавычках будет

преобразовываться в NULL. Этот параметр допускается только в команде COPY FROM и только для формата CSV.

ENCODING

Указывает, что файл имеет кодировку имя_кодировки. Если этот параметр опущен, выбирается текущая кодировка клиента. Подробнее об этом говорится ниже, в примечаниях.

19.4. Выводимая информация

В случае успешного завершения, COPY возвращает метку команды в виде

COPY число

Здесь число – количество скопированных записей.

Примечание. psql выводит эту метку, только если выполнялась не команда COPY ... TO STDOUT или её аналог в psql, метакоманда \copy ... to stdout. Это сделано для того, чтобы метка команды не смешалась с данными, выведенными перед ней.

19.5. Замечания

COPY может использоваться только с обычными таблицами, но не с представлениями. Однако, при необходимости можно скопировать представление так: COPY (SELECT * FROM имя_представления) TO

COPY обрабатывает только явно заданную таблицу, дочерние таблицы при копировании данных не затрагиваются. Поэтому, например COPY таблица TO выводит те же данные, что и запрос SELECT * FROM ONLY table. Для выгрузки всех данных в иерархии наследования можно применить COPY (SELECT * FROM table) TO

В таблице, данные которой читает команда COPY TO, требуется иметь право на выборку данных, а в таблице, куда вставляет значения COPY FROM, требуется право на добавление. При этом, если в команде перечисляются избранные колонки, достаточно иметь права только для них.

Файлы, указанные в команде COPY, читаются или записываются непосредственно сервером, не клиентским приложением. Поэтому они должны

располагаться на сервере или быть доступными серверу, а не клиенту. Они должны быть доступны на чтение или запись пользователю Синергия-БД (пользователю, от имени которого работает сервер), не клиенту. Аналогично, команда, указанная параметром PROGRAM, выполняется непосредственно сервером, а не клиентским приложением, и должна быть доступна на выполнение пользователю Синергия-БД. Выполнять команду COPY с файлом (или командой) разрешено только суперпользователям базы данных, так как она позволяет прочитать и записать любой файл, к которому имеет доступ сервер.

Не путайте команду COPY с реализованной в psql метакомандой \copy. Метакоманда \copy вызывает COPY FROM STDIN или COPY TO STDOUT, а затем работает с данными в файле, доступном клиенту psql. Таким образом, когда применяется команда \copy, доступность файла и права доступа зависят от клиента, а не от сервера.

Путь файла, указываемый в COPY, рекомендуется всегда задавать как абсолютный, а не относительный. Это обязательное условие для команды COPY TO, но COPY FROM позволяет прочитать файл, заданный и относительным путем. Такой путь будет интерпретироваться относительно рабочего каталога серверного процесса (обычно это каталог данных кластера), а не рабочего каталога клиента.

Выполнение команды в PROGRAM может быть ограничено и другими работающими в ОС механизмами контроля доступа, например SELinux.

COPY FROM вызывает все триггеры и обрабатывает все ограничения-проверки в целевой таблице. Однако правила при загрузке данных не вызываются.

При вводе и выводе данных COPY учитывается DateStyle. Для обеспечения переносимости на другие инсталляции Синергия-БД, в которых могут использоваться нестандартные настройки DateStyle, значение DateStyle следует установить равным ISO до вызова COPY TO. Также рекомендуется не выгружать данные с IntervalStyle равным sql_standard, так как сервер с другим значением IntervalStyle может неправильно воспринимать отрицательные интервалы в таких данных.

Входные данные интерпретируются согласно кодировке, заданной параметром `ENCODING`, или текущей кодировке клиента, а выходные кодируются в кодировке `ENCODING` или текущей кодировке клиента, даже если данные не проходят через клиента, а считываются или записываются в файл непосредственно сервером.

`COPY` прекращает операцию при первой ошибке. Это не должно приводить к проблемам в случае с `COPY TO`, но после `COPY FROM` в целевой таблице остаются ранее полученные строки. Эти строки не будут видимыми и доступными, но будут занимать место на диске. Если сбой происходит при копировании большого объема данных, это может приводить к значительным потерям дискового пространства. При желании вернуть потерянный объем, это можно сделать с помощью команды `VACUUM`.

`FORCE_NULL` и `FORCE_NOT_NULL` можно применить одновременно к одной колонке. В результате `NULL`-значения в кавычках будут преобразованы в `NULL`, а `NULL`-значения без кавычек – в пустые строки.

19.6. Форматы файлов

19.6.1. Текстовый формат

Когда применяется формат `text`, читаемые или записываемые данные представляют собой текстовый файл, строка в котором соответствует строке таблицы. Колонки в строке разделяются символом-разделителем. Значения самих колонок – строки, выдаваемые функцией вывода, либо воспринимаемые функцией ввода, соответствующей типу данных колонки. Заданный маркер `NULL` выводится и считывается вместо колонок со значением `NULL`. `COPY FROM` выдает ошибку, если в любой из строк во входном файле оказывается больше или меньше колонок, чем ожидается. С указанием `OIDS` значение `OID` считывается или записывается в первой колонке, предшествующей колонкам с основными данными.

Конец данных может обозначаться одной строкой, содержащей только обратную косую и точку (`\.`). Маркер конца данных не требуется при чтении из файла, так как его роль вполне выполняет конец файла; он необходим только при передаче данных в/из клиентского приложения по протоколу обмена до версии 3.0.

Символы обратной косой черты (\) в данных COPY позволяют экранировать символы данных, которые без них считались бы разделителями строк или колонок. В частности, предваряться обратной косой должны следующие символы, когда они оказываются в значении колонки: сама обратная косая черта, перевод строки, возврат каретки и текущий разделитель.

Маркер NULL передается команде COPY ТО как есть, без добавления обратной косой; COPY FROM, со своей стороны, ищет во вводимых данных маркеры NULL до удаления обратных косых. Таким образом, маркер NULL, например такой как \N, отличается от значения \N в данных (оно должно представляться в виде \\N). Команда COPY FROM распознает спецпоследовательности, описанные в таблице 19.1.

Таблица 19.1 - Спецпоследовательности

Последовательность	Представляет
\b	Забой (ASCII 8)
\f	Подача формы (ASCII 12)
\n	Новая строка (ASCII 10)
\r	Возврат каретки (ASCII 13)
\t	Табуляция (ASCII 9)
\v	Вертикальная табуляция (ASCII 11)
\цифры	Обратная косая с последующими 1-3 восьмеричными цифрами представляет символ с заданным числовым кодом
\хцифры	Обратная косая с последующим х и 1-2 шестнадцатеричными цифрами представляет символ с заданным числовым кодом

В настоящее время COPY ТО никогда не выводит спецпоследовательности с восьмеричными или шестнадцатеричными кодами, однако выводит другие вышеперечисленные спецпоследовательности вместо управляющих символов.

Любой другой символ после обратной косой, отсутствующий в приведенной выше таблице, будет представлять себя. Однако опасайтесь излишнего добавления обратных косых, так как это может привести к случайному образованию строки, обозначающей маркер конца данных (\.) или маркер NULL (\N по умолчанию). Эти строки будут восприняты прежде, чем обработаются спецпоследовательности с обратной косой.

В приложениях, генерирующих данные для COPY, настоятельно рекомендуется конвертировать символы новой строки и возврата каретки в последовательности \n и \r, соответственно. В настоящее время можно

представить возврат каретки в данных как обратная косая и возврат каретки, а перевод строки как обратная косая и перевод строки, однако это может не поддерживаться в будущих версиях. Такие символы также подвержены искажениям, если файл с выводом COPY переносится между разными системами (например, с Unix в Windows и наоборот).

COPY TO завершает каждую строку символом новой строки в стиле Unix («\n»). Серверы, работающие в Microsoft Windows, вместо этого выводят символы возврат каретки/новая строка («\r\n»), но только при выводе COPY в файл на сервере; для согласованности на разных платформах, COPY TO STDOUT всегда передает «\n», вне зависимости от платформы сервера. COPY FROM может воспринимать строки, завершающиеся символами новая строка, перевод каретки, либо возврат каретки+новая строка. Чтобы уменьшить риск ошибки из-за неэкранированных символов новой строки и возврата каретки, которые должны были быть данными, COPY FROM сигнализирует о проблеме, если концы строк во входных данных различаются.

19.6.2. Формат CSV

Этот формат применяется для импорта и экспорта данных в виде списка значений, разделенных запятыми (CSV), с которым могут работать многие другие программы, например электронные таблицы. Вместо правил экранирования значений, введенных в Синергия-БД для текстового формата, этот формат использует стандартный механизм экранирования CSV.

Значения в каждой записи разделяются символами DELIMITER. Если значение содержит символ разделителя, символ QUOTE, маркер NULL, символ возврата каретки или перевода строки, то все значение дополняется спереди и сзади символами QUOTE, а любое вхождение символа QUOTE или спецсимвола (ESCAPE) в данных предваряется спецсимволом. С указанием FORCE_QUOTE в кавычки будут принудительно заключаться любые значения не NULL в указанных колонках.

В формате CSV отсутствует стандартный способ отличить значение NULL от пустой строки. В Синергия-БД команда COPY решает это с помощью кавычек.

Значение NULL выводится в виде строки, задаваемой параметром NULL, и не заключается в кавычки, тогда как значение не NULL, со строкой, задаваемой параметром NULL, заключается. Например, с параметрами по умолчанию NULL записывается в виде пустой строки без кавычек, тогда как пустая строка записывается в двойных кавычках («»). При чтении значений действуют похожие правила. Указание `FORCE_NOT_NULL` позволяет избежать сравнений на NULL во входных данных в заданных колонках, а `FORCE_NULL` – преобразовывать в NULL маркеры NULL, даже заключенные в кавычки.

Так как обратная косая черта не является спецсимволом в формате CSV, маркер конца данных `\.` может быть и значением данных. Во избежание ошибок интерпретации данные `\.`, выводимые в виде единственного элемента строки, автоматически заключаются в кавычки при выводе, а при вводе этот маркер, заключенный в кавычки, не воспринимается как маркер конца данных. При загрузке файла, созданного другой программой, в котором в единственной колонке без кавычек оказалось значение `\.`, потребуется дополнительно заключить это значение в кавычки.

Примечания:

1. В формате CSV все символы являются значимыми. Заключенное в кавычки значение, дополненное пробелами или любыми другими символами, кроме `DELIMITER`, будет включать и эти символы. Это может приводить к ошибкам при импорте данных из системы, дополняющей строки CSV пробельными символами до некоторой фиксированной ширины. В случае возникновения такой проблемы необходимо обработать файл CSV и удалить из него замыкающие пробельные символы, прежде чем загружать данные из него в Синергия-БД.

2. Замечание: Обработчик формата CSV воспринимает и генерирует файлы CSV со значениями в кавычках, которые могут содержать символы возврата каретки и перевода строки. Таким образом, число строк в этих файлах не строго равно числу строк в таблице, как в файлах текстового формата.

3. Замечание: Многие программы генерируют странные и иногда неприемлемые файлы CSV, так что этот формат используется скорее по соглашению, чем по стандарту. Поэтому вам могут встретиться файлы, которые невозможно импортировать, используя этот механизм, а COPY может сформировать такие файлы, что их не смогут обработать другие программы.

19.6.3. Двоичный формат

При выборе формата `binary` все данные сохраняются/считываются в двоичном, а не текстовом виде. Иногда этот формат обрабатывается быстрее, чем текстовый и `CSV`, но он может оказаться непереносимым между разными машинными архитектурами и версиями Синергия-БД. Кроме того, двоичный формат сильно зависит от типов данных; например, он не позволяет вывести данные из колонки `smallint`, а затем прочитать их в колонку `integer`, хотя с текстовым форматом это вполне возможно.

Формат `binary` включает заголовок файла, ноль или более записей, содержащих данные строк, и окончание файла. Для заголовков и данных принят сетевой порядок байт.

19.6.4. Заголовок файла

Заголовок файла содержит 15 байт фиксированных полей, за которыми следует область расширения заголовка переменной длины. Фиксированные поля:

- сигнатура.

Последовательность из 11 байт `PGCOPY\n\377\r\n\0` – заметьте, что нулевой байт является обязательной частью сигнатуры. Эта сигнатура позволяет легко выявить файлы, испорченные при передаче, не сохраняющей все 8 бит данных. Она изменится при прохождении через фильтры, меняющие концы строк, отбрасывающие нулевые байты или старшие биты, либо добавляющие четность;

- поле флагов.

Маска из 32 бит, обозначающая важные аспекты формата файла. Биты нумеруются от 0 (LSB) до 31 (MSB). Учтите, что это поле хранится в сетевом порядке байт (наиболее значащий байт первый), как и все целочисленные поля в этом формате. Биты 16-31 зарезервированы для обозначения критичных особенностей формата; обработчик должен прервать чтение, встретив любой неожиданный бит в этом диапазоне. Биты 0-15 зарезервированы для обозначения особенностей, связанных с обратной совместимостью;

обработчик может просто игнорировать любые неожиданные биты в этом диапазоне. В настоящее время определен только один битовый флаг, остальные должны быть равны 0;

- бит 16.

При 1 в данные включается OID; при 0 – нет;

- длина области расширения заголовка.

Целое 32-битное число, определяющее длину в байтах остального заголовка, не включая само это значение. В настоящее время содержит 0, и сразу за ним следует первая запись. При будущих изменениях формата в заголовок могут быть добавлены дополнительные данные. Обработчик должен просто пропускать все расширенные данные заголовка, о которых ему ничего не известно.

Область расширения заголовка предусмотрена для размещения последовательности самоопределяемых блоков. Поле флагов не должно содержать указаний о том, что содержится в области расширения. Точное содержимое области расширения может быть определено в будущих версиях.

При таком подходе возможно как обратно-совместимое дополнение заголовка, так и не обратно-совместимое (установить старшие биты флагов, и добавить вспомогательные данные в область расширения, если это потребуется).

19.6.5. Записи

Каждая запись начинается с 16-битного целого числа, определяющего количество полей в записи. В настоящее время во всех записях должно быть одинаковое число записей, но так может быть не всегда. Затем, для каждого поля в записи указывается 32-битная длина поля, за которой следует это количество байт с данными поля. Значение длины не включает свой размер, и может быть равно нулю. В качестве особого варианта, -1 обозначает, что в поле содержится NULL. В случае с NULL за длиной не следуют байты данных.

Выравнивание или какие-либо дополнительные данные между полями не вставляются.

В настоящее время предполагается, что все значения данных в файле двоичного формата содержатся в двоичном формате (формате под кодом 1). Возможно, в будущем расширении в заголовок будет добавлено поле, позволяющее задавать другие коды форматов для разных колонок.

Чтобы определить подходящий двоичный формат для фактических данных, обратитесь к исходному коду Синергия-БД, в частности, к функциям `*send` и `*recv` для типов данных каждой колонки (обычно эти функции находятся в каталоге `src/backend/utils/adt/` в дереве исходного кода).

Если в файл включается OID, поле OID следует немедленно за числом, определяющим количество полей. Это поле не отличается от других ничем, кроме того, что оно не учитывается в количестве полей. В частности, для него также задается длина – это позволяет обрабатывать и четырех- и восьмибайтовые OID без особых сложностей, и даже вывести OID, равный NULL, если возникнет потребность в этом.

19.6.6. Окончание файла

Окончание файла состоит из 16-битного целого, содержащего -1. Это позволяет легко отличить его от счетчика полей в записи.

Обработчик, читающий файл, должен выдать ошибку, если число полей в записи не равно -1 или ожидаемому числу колонок. Это обеспечивает дополнительную проверку синхронизации данных.

19.7.Примеры

В следующем примере таблица передается клиенту с разделителем полей «вертикальная черта» (`|`):

```
COPY country TO STDOUT (DELIMITER '|');
```

Копирование данных из файла в таблицу `country`:

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

Копирование в файл только данных стран, название которых начинается с 'A':

```
COPY (SELECT * FROM country WHERE country_name LIKE  
'A%') TO '/usr1/proj/bray/sql/a_list_countries.copy';
```

Для копирования данных в сжатый файл можно направить вывод через

внешнюю программу сжатия:

```
COPY country TO PROGRAM 'gzip >
/usr1/proj/bray/sql/country_data.gz';
```

Пример данных, подходящих для копирования в таблицу из STDIN:

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
```

Примечание. Пробелы в каждой строке на самом деле обозначают символы табуляции.

Ниже приведены те же данные, но выведенные в двоичном формате. Данные показаны после обработки Unix-утилитой `od -c`. Таблица содержит три колонки; первая имеет тип `char (2)`, вторая – `text`, а третья – `integer`. Последняя колонка во всех строках содержит NULL.

```
00000000  P   G   C   O   P   Y   \n 377  \r  \n  \0  \0  \0  \0  \0  \0
00000020  \0  \0  \0  \0 003  \0  \0  \0 002  A   F  \0  \0  \0 013  A
00000040  F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0 003
00000060  \0  \0  \0 002  A   L  \0  \0  \0 007  A   L   B   A   N   I
00000100  A 377 377 377 377  \0 003  \0  \0  \0 002  D   Z  \0  \0  \0
00000120 007  A   L   G   E   R   I   A 377 377 377 377  \0 003  \0  \0
00000140  \0 002  Z   M  \0  \0  \0 006  Z   A   M   B   I   A 377 377
00000160 377 377  \0 003  \0  \0  \0 002  Z   W  \0  \0  \0  \b  Z   I
00000200  M   B   A   B   W   E 377 377 377 377 377 377
```

19.8. Совместимость

Оператор `COPY` отсутствует в стандарте SQL.

20. CREATE AGGREGATE

`CREATE AGGREGATE` – создать агрегатную функцию.

20.1. Синтаксис

```
CREATE AGGREGATE имя ( [ режим_аргумента ]
[ имя_аргумента ] тип_данных_аргумента [ , ... ] ) (
    SFUNC = функция_состояния,
    STYPE = тип_данных_состояния
    [ , SSPACE = размер_данных_состояния ]
    [ , FINALFUNC = функция_завершения ]
```

```

[ , FINALFUNC_EXTRA ]
[ , INITCOND = начальное_условие ]
[ , MSFUNC = функция_состояния_движ ]
[ , MINVFUNC = обратная_функция_движ ]
[ , MSTYPE = тип_данных_состояния_движ ]
[ , MSSPACE = размер_данных_состояния_движ ]
[ , MFINALFUNC = функция_завершения_движ ]
[ , MFINALFUNC_EXTRA ]
[ , MINITCOND = начальное_условие_движ ]
[ , SORTOP = оператор_сортировки ]
)

CREATE AGGREGATE имя ( [ [ режим_аргумента ]
[ имя_аргумента ] тип_данных_аргумента [ , ... ] ]
                        ORDER BY [ режим_аргумента ]
[ имя_аргумента ] тип_данных_аргумента [ , ... ] ) (
    SFUNC = функция_состояния,
    STYPE = тип_данных_состояния
    [ , SSPACE = размер_данных_состояния ]
    [ , FINALFUNC = функция_завершения ]
    [ , FINALFUNC_EXTRA ]
    [ , INITCOND = начальное_условие ]
    [ , HYPOTHETICAL ]
)

```

20.2.Описание

CREATE AGGREGATE создает новую агрегатную функцию. Некоторое количество базовых и часто используемых агрегатных функций включено в дистрибутив. Но если нужно адаптировать их к новым типам или создать недостающие агрегатные функции, это можно сделать с помощью команды CREATE AGGREGATE.

Если указывается имя схемы (например, CREATE AGGREGATE myschema.myagg ...), агрегатная функция создается в указанной схеме. В противном случае она создается в текущей схеме.

Агрегатная функция идентифицируется по имени и типам входных данных. Две агрегатных функции в одной схеме могут иметь одно имя, только если они работают с разными типами данных. Имя и тип(ы) входных данных агрегата не могут совпадать с именем и типами данных любой другой обычной функции в той же схеме. Это же правило действует при перегрузке имен обычных функций.

Простую агрегатную функцию образуют одна или две обычные функции: функция перехода состояния `функция_состояния` и необязательная функция окончательного вычисления `функция_завершения`. Они используются следующим образом:

```
функция_состояния( внутреннее-состояние, следующие-
значения-данных ) ---> следующее-внутреннее-состояние
```

```
функция_завершения( внутреннее-состояние ) --->
агрегатное_значение
```

Синергия-БД создает временную переменную типа `тип_данных_состояния` для хранения текущего внутреннего состояния агрегата. Затем для каждой поступающей строки вычисляются значения аргументов агрегата и вызывается функция перехода состояния с текущим значением состояния и полученными аргументами; эта функция вычисляет следующее внутреннее состояние. Когда таким образом будут обработаны все строки, вызывается завершающая функция, которая должна вычислить возвращаемое значение агрегата. Если функция завершения отсутствует, просто возвращается конечное значение состояния.

Агрегатная функция может определить начальное условие, то есть начальное значение для внутренней переменной состояния. Это значение задается и сохраняется в базе данных в виде строки типа `text`, но оно должно быть допустимым внешним представлением константы типа данных переменной состояния. По умолчанию начальным значением состояния считается `NULL`.

Если функция перехода состояния объявлена как «strict» (строгая), её нельзя вызывать с входными значениями `NULL`. В этом случае агрегатная функция выполняется следующим образом. Строки со значениями `NULL` игнорируются (функция перехода не вызывается и предыдущее значение состояния не меняется) и

если начальное состояние равно `NULL`, то в первой же строке, в которой все входные значения не `NULL`, первый аргумент заменяет значение состояния, а функция перехода вызывается для каждой последующей строки, в которой все входные значения не `NULL`. Это поведение удобно для реализации таких агрегатных функций, как `max`. Заметьте, что такое поведение возможно, только если `тип_данных_состояния` совпадает с первым `типом_данных_аргумента`. Если же эти типы различаются, необходимо задать начальное условие не `NULL` или использовать нестрогую функцию перехода состояния.

Если функция перехода состояния не является строгой, она вызывается безусловно для каждой поступающей строки и должна сама обрабатывать вводимые значения и переменную состояния, равные `NULL`. Это позволяет разработчику агрегатной функции полностью управлять тем, как она воспринимает значения `NULL`.

Если функция завершения объявлена как «strict» (строгая), она не будет вызвана при конечном значении состояния, равном `NULL`; вместо этого автоматически возвращается результат `NULL`. (Разумеется, это вполне нормальное поведение для строгих функций.) Когда функция завершения вызывается, она в любом случае может вернуть значение `NULL`. Например, функция завершения для `avg` возвращает `NULL`, если определяет, что было обработано ноль строк.

Иногда бывает полезно объявить функцию завершения как принимающую не только состояние, но и дополнительные параметры, соответствующие входным данным агрегата. В основном это имеет смысл для полиморфных функций завершения, которым может быть недостаточно знать тип данных только переменной состояния, чтобы вывести тип результата. Эти дополнительные параметры всегда передаются как `NULL` (так что функция завершения не должна быть строгой, когда применяется `FINALFUNC_EXTRA`), но в остальном это обычные параметры. Функция завершения может выяснить фактические типы аргументов в текущем вызове, воспользовавшись системным вызовом `get_fn_expr_argtype`.

Агрегатная функция может дополнительно поддерживать режим движущегося

агрегата. Для применения этого режима требуется указать параметры `MSFUNC`, `MINVFUNC` и `MSTYPE`, а также, возможно, параметры `MSPACE`, `MFINALFUNC`, `MFINALFUNC_EXTRA` и `MINITCOND`. За исключением `MINVFUNC`, эти параметры работают как соответствующие параметры простого агрегата без начальной буквы `M`; они определяют отдельную реализацию агрегата, включающую функцию обратного перехода.

Если в список параметров добавлено указание `ORDER BY`, создается особый тип агрегата, называемый сортирующим агрегатом; с указанием `HYPOTHETICAL` создается гипотезирующий агрегат. Эти агрегаты работают с группами отсортированных значений и зависят от порядка сортировки, поэтому определение порядка сортировки входных данных является неотъемлемой частью их вызова. Кроме того, они могут иметь непосредственные аргументы, которые вычисляются единожды для всей процедуры агрегирования, а не для каждой поступающей строки. Гипотезирующие агрегаты представляют собой подкласс сортирующих агрегатов, в которых непосредственные аргументы должны совпадать, по количеству и типам данных, с агрегируемыми аргументами. Это позволяет добавить значения этих непосредственных аргументов в набор агрегируемых строк в качестве дополнительной «гипотетической» строки.

Агрегаты, работающие подобно `MIN` и `MAX`, иногда можно оптимизировать, заменив сканирование всех строк таблицы обращением к индексу. Если агрегат подлежит такой оптимизации, это можно указать, определив оператор сортировки. Основное требование при этом: агрегат должен выдавать в результате первый элемент по порядку сортировки, задаваемому оператором; другими словами:

```
SELECT agg(col) FROM tab;
```

должно быть равнозначно:

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

Дополнительно предполагается, что агрегат игнорирует значения `NULL` и возвращает `NULL`, только если строк со значениями не `NULL` не нашлось. Обычно оператор `<` является подходящим оператором сортировки для `MIN`, а `>` – для `MAX`. Заметьте, что обращение к индексу может дать эффект, только если заданный

оператор реализует стратегию «меньше» или «больше» в классе операторов индекса В-дерева.

Чтобы создать агрегатную функцию, необходимо иметь право USAGE для типов аргументов, типа(ов) состояния и типа результата, а также право EXECUTE для функций перехода и завершения.

20.3.Параметры

имя

Имя (возможно, дополненное схемой) создаваемой агрегатной функции.

режим_аргумента

Режим аргумента: IN или VARIADIC. (Агрегатные функции не поддерживают выходные аргументы (OUT).) По умолчанию подразумевается IN. Режим VARIADIC может быть указан только последним.

имя_аргумента

Имя аргумента. В настоящее время используется только в целях документирования. Если опущено, соответствующий аргумент будет безымянным.

тип_данных_аргумента

Тип входных данных, с которым работает эта агрегатная функция. Для создания агрегатной функции без аргументов вставьте * вместо списка с определениями аргументов. Пример такой агрегатной функции: count (*).

базовый_тип

В прежнем синтаксисе CREATE AGGREGATE тип входных данных задавался параметром basetype, а не записывался после имени агрегата. Это позволяло указать только один входной параметр. Чтобы определить функцию без аргументов, используя этот синтаксис, в качестве значения basetype нужно указать «ANY» (не *). Создать сортирующий агрегат старый синтаксис не позволял.

функция_состояния

Имя функции перехода состояния, вызываемой для каждой входной строки. Для обычных агрегатных функций с N аргументами, функция_состояния должна принимать N+1 аргумент, первый должен иметь тип тип_данных_состояния, а

остальные – типы соответствующих входных данных. Возвращать она должна значение типа `тип_данных_состояния`. Эта функция принимает текущее значение состояния и текущие значения входных данных, и возвращает следующее значение состояния.

В сортирующих (и в том числе, гипотезирующих) агрегатах функция перехода состояния получает только текущее значение состояния и агрегируемые аргументы, без непосредственных аргументов. Других отличий у нее нет.

`тип_данных_состояния`

Тип данных значения состояния для агрегатной функции.

`размер_данных_состояния`

Средний размер значения состояния агрегата (в байтах). Если этот параметр опущен или равен нулю, применяемая оценка по умолчанию определяется по типу `тип_данных_состояния`. Планировщик использует это значение для оценивания объема памяти, требуемому для агрегатного запроса с группировкой. Планировщик может применить агрегирование по хешу для такого запроса, только если хеш-таблица, судя по оценке, поместится в `work_mem`; таким образом, при больших значениях этого параметра агрегирование по хешу будет менее желательным.

`функция_завершения`

Имя функции завершения, вызываемой для вычисления результата агрегатной функции после обработки всех входных строк. Для обычного агрегата эта функция должна принимать единственный аргумент типа `тип_данных_состояния`. Возвращаемым типом агрегата будет тип, который возвращает эта функция. Если функция_завершения не указана, результатом агрегата будет конечное значение состояния, а типом результата – `тип_данных_состояния`.

В сортирующих (и в том числе, гипотезирующих) агрегатах функция завершения получает не только конечное значение состояния, но и значения всех непосредственных аргументов.

Если команда содержит указание `FINALFUNC_EXTRA`, то в дополнение к конечному значению состояния и всем непосредственным аргументам функция завершения получает добавочные значения `NULL`, соответствующие обычным

(агрегируемым) аргументам агрегата. Это в основном полезно для правильного определения типа результата при создании полиморфной агрегатной функции.

начальное_условие

Начальное значение переменной состояния. Оно должно задаваться строковой константой в форме, пригодной для ввода в тип_данных_состояния. Если не указано, начальным значением состояния будет NULL.

функция_состояния_движ

Имя функции прямого перехода состояния, вызываемой для каждой входной строки в режиме движущегося агрегата. Это точно такая же функция, как и обычная функция перехода, но её первый аргумент и результат имеют тип тип_данных_состояния_движ, который может отличаться от типа тип_данных_состояния.

обратная_функция_движ

Имя функции обратного перехода состояния, применяемой в режиме движущегося агрегата. У этой функции те же типы аргумента и результатов, что и у функции_состояния_движ, но она предназначена не для добавления, а для удаления значения из текущего состояния агрегата. Функция обратного перехода должна иметь ту же характеристику строгости, что и функция прямого перехода.

тип_данных_состояния_движ

Тип данных значения состояния для агрегатной функции в режиме движущегося агрегата.

размер_данных_состояния_движ

Примерный размер значения состояния в режиме движущегося агрегата. Он имеет то же значение, что и размер_данных_состояния.

функция_завершения_движ

Имя функции завершения, вызываемой в режиме движущегося агрегата для вычисления результата агрегатной функции после обработки всех входных строк. Она работает так же, как функция_завершения, но её первый аргумент имеет тип тип_данных_состояния_движ, а дополнительными пустыми аргументами управляет параметр MFINALFUNC_EXTRA. Тип результата, который определяет

функция_завершения_движ, или тип_данных_состояния_движ, должен совпадать с типом результата обычной реализации агрегата.

начальное_условие_движ

Начальное значение переменной состояния в режиме движущегося агрегата. Оно применяется так же, как начальное_условие.

оператор_сортировки

Связанный оператор сортировки для реализации агрегатов, подобных MIN или MAX. Здесь указывается просто имя оператора (возможно, дополненное схемой). Предполагается, что оператор поддерживает те же типы входных данных, что и агрегат (который должен быть обычным и иметь один аргумент).

HYPOTHETICAL

Этот признак, допустимый только для сортирующих агрегатов, указывает, что агрегатные аргументы должны обрабатываться согласно требованиям гипотезирующих агрегатов: то есть последние несколько прямых аргументов должны соответствовать по типам агрегатным аргументам (WITHIN GROUP). Признак HYPOTHETICAL не влияет на поведение во время выполнения, он учитывается только при разрешении типов данных и правил сортировки аргументов.

Параметры CREATE AGGREGATE могут записываться в любом порядке, не обязательно так, как показано выше.

20.4.Замечания

В параметрах, определяющих имена вспомогательных функций, при необходимости можно написать имя схемы, например: SFUNC = public.sum. Однако типы аргументов там не указываются – типы аргументов вспомогательных функций определяются другими параметрами.

Если агрегатная функция поддерживает режим движущегося агрегата, это увеличивает эффективность вычислений, когда она применяется в качестве оконной функции для окна с движущимся началом рамки (то есть, когда начало определяется не как UNBOUNDED PRECEDING). По сути, функция прямого перехода добавляет входные значения к состоянию агрегата, когда они поступают в рамку окна снизу, а

функция обратного перехода снова вычитает их, когда они покидают рамку сверху. Поэтому вычитаются значения в том же порядке, в каком добавлялись. Когда бы ни вызывалась функция обратного перехода, она таким образом получит первое из добавленных, но ещё не удаленных значений аргумента. Функция обратного перехода может рассчитывать на то, что после того, как она удалит самые старые данные, в текущем состоянии останется ещё как минимум одна строка. (Когда это правило могло бы нарушиться, механизм оконных функций просто начинает агрегировать данные заново, а не вызывает функцию обратного перехода.)

Функция прямого перехода для режима движущегося агрегата не может возвращать `NULL` в качестве нового значения состояния. Если `NULL` возвращает функция обратного перехода, это показывает, что она не может произвести обратное вычисление для этих конкретных данных, и что вычисление агрегата следует выполнить заново от начальной позиции текущей рамки. Благодаря этому соглашению, режим движущегося агрегата можно использовать, даже если иногда возникают ситуации, в которых обратный расчет состояния производить непрактично.

Агрегатную функцию можно использовать с движущимися рамками и без реализации движущегося агрегата, но при этом Синергия-БД будет заново агрегировать все данные при каждом перемещении начала рамки. Заметьте, что вне зависимости от того, поддерживает ли агрегатная функция режим движущегося агрегата, Синергия-БД может обойтись без повторных вычислений при сдвиге конца рамки; новые значения просто продолжают добавляться в состояние агрегата. Предполагается, что функция завершения не повреждает значение состояния агрегата, так что вычисление агрегата можно продолжить даже после получения результата для строк в определенной рамке.

Синтаксис сортирующих агрегатных функций позволяет указать `VARIADIC` и в последнем непосредственном параметре, и в последнем агрегатном (`WITHIN GROUP`). Однако в текущей реализации на применение `VARIADIC` накладываются два ограничения. Во-первых, в сортирующих агрегатах можно использовать только `VARIADIC` «any», но не другие типы переменных массивов. Во-вторых, если

последним непосредственным аргументом является VARIADIC «any», то допускается только один агрегатный аргумент и это тоже должен быть VARIADIC «any». В представлении, используемом в системных каталогах, эти два параметра объединяются в один элемент VARIADIC «any», так как в pg_proc нельзя представить функцию с несколькими параметрами VARIADIC. Если агрегатная функция является гипотезирующей, непосредственные аргументы, соответствующие параметру VARIADIC «any», будут гипотетическими; любые предшествующие параметры представляют дополнительные непосредственные аргументы, которые могут не соответствовать агрегатным.

В настоящее время сортирующие агрегатные функции не поддерживают режим движущегося агрегата, так как их нельзя применять в качестве оконных функций.

20.5.Совместимость

Оператор CREATE AGGREGATE является языковым расширением Синергия-БД. В стандарте SQL не предусмотрено создание пользовательских агрегатных функций.

21. CREATE CAST

CREATE CAST – создать приведение.

21.1.Синтаксис

```
CREATE CAST (исходный_тип AS целевой_тип)
    WITH FUNCTION имя_функции (тип_аргумента [, ...])
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

```
CREATE CAST (исходный_тип AS целевой_тип)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

```
CREATE CAST (исходный_тип AS целевой_тип)
    WITH INOUT
```

21.2.Описание

CREATE CAST создает новое приведение. Приведение определяет, как выполнить преобразование из одного типа в другой.

Пример.

```
SELECT CAST(42 AS float8);
```

преобразует целочисленную константу 42 к типу float8, вызывая ранее определенную функцию, в данном случае float8(int4). Если подходящее приведение не определено, возникнет ошибка преобразования.

Два типа могут быть двоично-сводимыми; это означает, что преобразование может быть выполнено «бесплатно», без вызова какой-либо функции. Для этого требуется, чтобы соответствующие значения имели одинаковое внутреннее представление. Например, типы text и varchar двоично-сводимые в обе стороны. Отношение двоичной сводимости не обязательно симметрично. Например, приведение типа xml к типу text в текущей реализации можно выполнить бесплатно, но для преобразования в обратном направлении требуется функция, выполняющая как минимум синтаксическую проверку. Два типа, двоично-сводимые в обе стороны, также называются двоично-совместимыми.

Приведение можно определить как преобразование ввода/вывода, используя указание WITH INOUT. В этом случае для приведения одного типа к другому вызывается функция вывода исходного типа данных, а выданная ей строка передается функции ввода целевого типа. Во многих случаях эта возможность избавляет от необходимости писать для преобразования всех типов отдельные функции приведения. Преобразование ввода/вывода работает так же, как и обычное приведение с функцией; отличается только реализация.

По умолчанию, приведение можно вызвать, только записав его явно, то есть применив конструкцию CAST(x AS имя_типа) или x::имя_типа.

Если приведение помечено AS ASSIGNMENT, его можно вызывать неявно, присваивая значение колонке с целевым типом данных. Например, если foo.fl –

колонка типа `text`, то команда:

```
INSERT INTO foo (f1) VALUES (42);
```

будет допустимой, если приведение типа `integer` к `text` помечено `AS ASSIGNMENT`, и не будет в противном случае. Для описания такого типа приведений мы обычно используем термин приведение присваивания.

Если приведение помечено `AS IMPLICIT`, оно будет вызываться неявно в любом контексте, будь то присваивание или внутреннее преобразование в выражении. Обычно мы называем приведение такого типа неявным приведением. Например, рассмотрите этот запрос:

```
SELECT 2 + 4.0;
```

При разборе запроса константам сначала назначаются типы `integer` и `numeric`. Однако в системных каталогах нет оператора `integer + numeric`, хотя есть оператор `numeric + numeric`. Таким образом, запрос выполнится успешно, если существует преобразование типа `integer` к `numeric` с пометкой `AS IMPLICIT` – и на самом деле это так. Анализатор запроса применит неявное приведение и запрос будет обработан, как если бы он был записан в виде:

```
SELECT CAST ( 2 AS numeric ) + 4.0;
```

Системные каталоги также содержат приведение типа `numeric` к `integer`. Если бы это приведение тоже было бы помечено `AS IMPLICIT` (на самом деле это не так), анализатору запроса пришлось бы выбирать между предыдущим вариантом и приведением константы `numeric` к типу `integer` с последующим применением оператора `integer + integer`. Не имея возможности выбрать лучший вариант, анализатор бы не смог разрешить запрос и объявил бы его неоднозначным. Именно благодаря тому, что только одно из двух приведений сделано неявным, анализатор приходит к пониманию, что предпочтительным является преобразование выражения `numeric-и-integer` в `numeric`; отдельного встроенного знания об этом нет.

Определяя, объявлять ли приведения неявными, разумно проявлять консерватизм. При чрезмерном количестве способов неявного приведения Синергия-БД может выбирать неожиданные интерпретации команд, или вовсе не сможет выполнить команды из-за наличия множества возможных интерпретаций. Как

правило, следует делать приведение неявно вызываемым только для преобразований, сохраняющих информацию, между типами в одной общей категории типов. Например, приведение `int2` к `int4` разумно сделать неявным, но приведение `float8` к `int4`, возможно, лучше сделать только приведением присваивания. Приведения типов разных категорий, например, `text` к `int4`, лучше делать только явными.

Примечание. Иногда ради удобства или соответствия стандартам требуется ввести множество неявных преобразований для нескольких типов, что приводит к неизбежной неоднозначности. Чтобы анализатор запроса мог обеспечить желаемое поведение в таких случаях, он дополнительно принимает во внимание категории типов и предпочитаемые типы. Подробнее это описано в `CREATE TYPE`.

Чтобы создать приведение, необходимо быть владельцем одного (исходного или целевого) типа и иметь право `USAGE` для другого типа. Создать двоично-сводимое приведение могут только суперпользователи. Это ограничение введено потому, что преобразование данных с ошибочным двоичным сведением может легко вызывать сбой сервера.

21.3.Параметры

`исходный_тип`

Имя исходного типа данных для приведения.

`целевой_тип`

Имя целевого типа данных для приведения.

`имя_функции(тип_аргумента [, ...])`

Функция, вызываемая для выполнения приведения. Имя функции может быть дополнено схемой, в противном случае для поиска функции просматривается путь поиска. Тип данных результата должен соответствовать целевому типу приведения. Аргументы функции рассматриваются ниже.

`WITHOUT FUNCTION`

Обозначает, что исходный тип сводится к целевому на двоичном уровне, так что функция для приведения не требуется.

`WITH INOUT`

Обозначает, что приведение выполняется как преобразование ввода/вывода, то

есть вызывается функция вывода исходного типа данных, а её результат-строка передается функции ввода целевого типа.

AS ASSIGNMENT

Обозначает, что приведение может вызываться неявно в контексте присваивания.

AS IMPLICIT

Обозначает, что приведение может вызываться неявно в любом контексте.

Функции, реализующие приведение, могут иметь от одного до трех аргументов. Тип первого аргумента должен быть идентичен или двоично-сводимым к исходному типу приведения. Второй аргумент, если он есть, должен иметь тип `integer`; в нем передается модификатор типа, связанный с целевым типом, или `-1`, если он отсутствует. Третий аргумент, если он есть, должен иметь тип `boolean`; в нем передается `true`, если приведение выполняется явно, либо `false` в противном случае. Это довольно экстравагантно, но стандарт SQL предусматривает разное поведение для явного и неявного приведения в некоторых случаях. Этот аргумент предназначен для функций, которые должны реализовывать такие приведения. Однако создавать собственные типы данных, для которых это имело бы значение, не рекомендуется.

Возвращаемый тип функции приведения должен быть идентичным или двоично-сводимым к целевому типу приведения.

Обычно исходный и целевой типы в приведении различаются, однако можно объявить приведение одного типа к такому же, если функция, реализующая преобразование, имеет более одного аргумента. Это используется для представления в системных каталогах функций, сводящих разные длины типов. Реализующая такое приведение функция будет сводить значение типа к значению с определенным модификатором, заданному вторым аргументом.

Когда исходный и целевой типы приведения различаются и функция принимает более одного аргумента, преобразование типа из одного в другой и сведение к нужной длине может выполняться за один шаг. Если же соответствующей записи не находится, приведение к типу с определенным модификатором

выполняется в два этапа: сначала выполняется преобразование типа, а затем применяется модификатор типа.

Приведение типа домена или к типу домена в настоящее время не осуществляется. При попытке выполнить такое приведение вместо него выполняется приведение, связанное с базовым типом домена.

21.4.Замечания

Для удаления приведений, созданных пользователем, применяется DROP CAST.

Помните, что когда требуется преобразовывать типы в обе стороны, необходимо явно описать два приведения.

Обычно не требуется создавать приведения между пользовательскими типами и стандартными строковыми типами (`text`, `varchar` и `char(n)`), а также пользовательскими типами, относящимися к категории строковых). Для них Синергия-БД предоставляет автоматическое преобразование ввода/вывода. Автоматические приведения к строковым типам считаются приведениями присваивания, а автоматические приведения строковых типов к другим могут быть только явными. Это поведение можно переопределить, создав собственное приведение, заменяющее автоматическое, но обычно это нужно, только чтобы сделать вызов более удобным, чем стандартное только присваивание или явное указание. Возможен и другой повод для такого переопределения – желание создать приведение, работающее не так, как функция ввода/вывода типа; но это настолько удивительно, что следует дважды подумать, хороша ли эта идея. На самом деле у небольшого количества встроенных типов имеются подобные специфические приведения, в основном из-за требований стандарта SQL.

Хотя это и не обязательно, но рекомендуется следовать старому соглашению называть функции, реализующие приведение, по целевому типу данных. Многие привыкли выполнять преобразование типов данных, записывая его в стиле функций, т.е. `имя_типа(x)`. Эта запись на самом деле ни больше ни меньше как просто вызов функции, реализующей приведение; такой вызов не воспринимается как именно

приведение. Если называть функции, не следуя этому соглашению, это может оказаться неожиданным для пользователей. Так как Синергия-БД позволяет перегружать одно и то же имя функции с разными типами аргументов, ничто не мешает создать множество функций приведения разных типов к одному, названных по имени этого целевого типа.

Примечания:

1. В предыдущем абзаце допущено некоторое упрощение: есть два случая, когда конструкция с вызовом функции выполняется как приведение, без сопоставления с фактической функцией. Если вызову функции имя(х) в точности не соответствует существующая функция, но имеется тип данных имя и в `pg_cast` есть двоично-сводимое приведение типа `x` к этому типу, такой вызов будет воспринят как приведение. Это исключение введено, чтобы двоично-сводимое приведение можно было вызывать, используя синтаксис функций, несмотря на то, что никакой функции преобразования у него нет. Аналогично, если запись приведения в `pg_cast` отсутствует, но в случае приведения это было бы преобразование в/из строкового типа, такой вызов будет выполнен как преобразование ввода/вывода. Это исключение позволяет вызывать преобразование ввода/вывода, используя синтаксис вызова функции.

2. Но есть исключение и из этого исключения: преобразование ввода/вывода из составных типов в строковые нельзя вызвать в виде функции, его необходимо записать как явное приведение (используя `CAST` или запись `::`). Это исключение было добавлено, потому что после введения автоматически предоставляемых преобразований ввода/вывода, оказалось слишком легко случайно вызвать такое приведение, тогда как имела в виду ссылка на колонку или функцию.

21.5.Примеры

Создание приведения присваивания типа `bigint` к типу `int4` с помощью функции `int4(bigint)`:

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint)
AS ASSIGNMENT;
```

Это приведение уже предопределено в системе.

21.6.Совместимость

Команда `CREATE CAST` соответствует стандарту SQL.

22. CREATE COLLATION

`CREATE COLLATION` – создать правило сортировки.

22.1.Синтаксис

```
CREATE COLLATION имя (  
    [ LOCALE = локаль, ]  
    [ LC_COLLATE = категория_сортировки, ]  
    [ LC_STYPE = категория_типов_символов ]  
)  
CREATE COLLATION имя FROM существующее_правило
```

22.2.Описание

CREATE COLLATION определяет новое правило сортировки, используя параметры локали операционной системы, либо копируя существующее правило.

Чтобы создать правило сортировки, необходимо иметь право CREATE в целевой схеме.

22.3.Параметры

имя

Имя правила сортировки, возможно, дополненное схемой. Если схема не указана, правило сортировки создается в текущей схеме. Заданное имя правила должно быть уникальным в этой схеме. Системные каталоги могут содержать правила сортировки с одним именем, но предназначенные для разных кодировок, однако они будут игнорироваться, если их кодировка не совпадает с кодировкой базы данных.

локаль

Это краткая запись для одновременной установки LC_COLLATE и LC_STYPE. Если указан этот вариант, задать любой из этих параметров отдельно нельзя.

категория_сортировки

Указанная локаль операционной системы устанавливается в качестве категории локали LC_COLLATE. Эта локаль должна быть применимой к кодировке текущей базы данных.

категория_типов_символов

Указанная локаль операционной системы устанавливается в качестве

категории локали LC_STYPE. Эта локаль должна быть применимой к кодировке текущей базы данных.

существующее_правило

Имя копируемого существующего правила сортировки. Новое правило сортировки получит те же свойства, что и существующее, но будет независимым объектом.

22.4.Замечания

Для удаления созданных пользователем правил сортировки применяется команда DROP COLLATION.

22.5.Примеры

Создание правила сортировки из локали операционной системы fr_FR.utf8 (предполагается, что кодировка текущей базы данных – UTF8):

```
CREATE COLLATION french (LOCALE = 'fr_FR.utf8');
```

Создание правила сортировки из уже существующего:

```
CREATE COLLATION german FROM «de_DE»;
```

Иногда удобно использовать в приложениях имена правил сортировки, не зависящие от операционной системы.

22.6.Совместимость

Оператор CREATE COLLATION определен в стандарте SQL, но его действие ограничено копированием существующего правила сортировки. Синтаксис создания нового правила сортировки представляет собой расширение Синергия-БД.

23. CREATE CONVERSION

CREATE CONVERSION – создать перекодировку

23.1.Синтаксис

```
CREATE [ DEFAULT ] CONVERSION имя
```

```
FOR исходная_кодировка TO целевая_кодировка FROM  
имя_функции
```

23.2.Описание

CREATE CONVERSION определяет новую перекодировку наборов символов. Кроме того, перекодировки, помеченные как DEFAULT, могут применяться для автоматического преобразования кодировки между клиентом и сервером. Для такого применения должны быть определены две перекодировки: из кодировки А в В и из кодировки В в А.

Чтобы создать перекодировку, необходимо иметь право EXECUTE для реализующей функции и право CREATE в целевой схеме.

23.3.Параметры

DEFAULT

Предложение DEFAULT показывает, что эта перекодировка должна использоваться по умолчанию для преобразования заданной исходной кодировки в целевую. Для каждой пары кодировок может быть только одна перекодировка по умолчанию.

имя

Имя перекодировки, возможно, дополненное схемой. Если схема не указана, перекодировка создается в текущей схеме. Имя перекодировки должно быть уникально в этой схеме.

исходная_кодировка

Имя исходной кодировки.

целевая_кодировка

Имя целевой кодировки.

имя_функции

Функция, выполняющая перекодирование. Имя функции может быть дополнено схемой, в противном случае для поиска функции просматривается путь поиска.

Функция должна иметь следующую сигнатуру:

```
conv_proc(  
    integer, -- идентификатор исходной кодировки  
    integer, -- идентификатор целевой кодировки
```

```

    cstring, -- исходная строка (строка, завершающаяся
0, как в C)
    internal, -- целевая строка (заполняется строкой,
завершающейся 0, как в C)
    integer -- длина исходной строки
) RETURNS void;
```

23.4.Замечания

Для удаления перекодировок, созданных пользователем, применяется DROP CONVERSION.

Набор прав, требуемых для создания перекодировки, может измениться в будущих версиях.

23.5.Примеры

Создание перекодировки из кодировки UTF8 в LATIN1 с использованием функции myfunc:

```

CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM
myfunc;
```

23.6.Совместимость

Оператор CREATE CONVERSION является расширением Синергия-БД. В стандарте SQL отсутствует оператор CREATE CONVERSION, но есть очень похожий по предназначению и синтаксису оператор CREATE TRANSLATION.

24. CREATE DATABASE

CREATE DATABASE – создать базу данных.

24.1.Синтаксис

```

CREATE DATABASE имя
    [ [ WITH ] [ OWNER [=] имя_пользователя ]
      [ TEMPLATE [=] шаблон ]
      [ ENCODING [=] кодировка ]
      [ LC_COLLATE [=] категория_сортировки ]
      [ LC_STYPE [=] категория_типов_символов ]
```



```
[ TABLESPACE [=] табл_пространство ]
```

```
[ CONNECTION LIMIT [=] предел_подключений ] ]
```

24.2.Описание

Команда `CREATE DATABASE` создает базу данных Синергия-БД.

Чтобы создать базу данных, необходимо быть суперпользователем или иметь специальное право `CREATEDB`.

По умолчанию новая база данных создается копированием стандартной системной базы данных `template1`. Задать другой шаблон можно, добавив указание `TEMPLATE имя`. В частности, написав `TEMPLATE template0`, можно создать девственно чистую базу данных, содержащую только стандартные объекты, предопределенные установленной версией Синергия-БД. Это бывает полезно, когда копировать в новую базу любые дополнительные объекты, добавленные локально в `template1`, нежелательно.

24.3.Параметры

имя

Имя создаваемой базы данных.

имя_пользователя

Имя пользователя (роли), назначаемого владельцем новой базы данных, либо `DEFAULT`, чтобы владельцем стал пользователь по умолчанию (а именно, пользователь, выполняющий команду). Чтобы создать базу данных и сделать её владельцем другую роль, необходимо быть непосредственным или опосредованным членом этой роли, либо суперпользователем.

шаблон

Имя шаблона, из которого будет создаваться новая база данных, либо `DEFAULT`, чтобы выбрать шаблон по умолчанию (`template1`).

кодировка

Кодировка символов в новой базе данных. Укажите строковую константу (например, `'SQL_ASCII'`) или целочисленный номер кодировки, либо `DEFAULT`, чтобы выбрать кодировку по умолчанию (а именно, кодировку шаблона).

категория_сортировки

Порядок сортировки (LC_COLLATE), который будет использоваться в новой базе данных. Этот параметр определяет порядок сортировки строк, например, в запросах с ORDER BY, а также порядок индексов по текстовым колонкам. По умолчанию используется порядок сортировки, установленный в шаблоне. Дополнительные ограничения описаны ниже.

категория_типов_символов

Классификация символов (LC_TYPE), которая будет применяться в новой базе данных. Этот параметр определяет принадлежность символов категориям, например: строчные, заглавные, цифры и т.п. По умолчанию используется классификация символов, установленная в шаблоне. Дополнительные ограничения описаны ниже.

табл_пространство

Имя табличного пространства, связываемого с новой базой данных, или DEFAULT для использования табличного пространства шаблона. Это табличное пространство будет использоваться по умолчанию для объектов, создаваемых в этой базе.

предел_подключений

Максимальное количество одновременных подключений к этой базе данных. Значение -1 (по умолчанию) снимает ограничение.

Дополнительные параметры могут записываться в любом порядке, не обязательно так, как показано выше.

24.4.Замечания

CREATE DATABASE нельзя выполнять внутри блока транзакции.

Ошибки, содержащие сообщение «не удалось инициализировать каталог базы данных», чаще всего связаны с нехваткой прав в каталоге данных, заполнением диска или другими проблемами в файловой системе.

Для удаления базы данных применяется DROP DATABASE.

Программа createdb представляет собой оболочку этой команды, созданную

ради удобства.

Конфигурационные параметры уровня базы данных (устанавливаемые командой `ALTER DATABASE`) из шаблона в новую базу данных не копируются.

Хотя с помощью этой команды можно скопировать любую базу данных, а не только `template1`, указав её имя в качестве имени шаблона, она не предназначена (пока) для использования в качестве универсального средства вроде «COPY DATABASE». Принципиальным ограничением является невозможность копирования базы данных шаблона, если установлены другие подключения к ней. `CREATE DATABASE` выдает ошибку, если при запуске команды есть другие подключения к этой базе; в противном случае новые подключения к базе блокируются до завершения команды `CREATE DATABASE`.

Кодировка символов, указанная для новой базы данных, должна быть совместима с выбранными параметрами локали (`LC_COLLATE` и `LC_CTYPE`). Если выбрана локаль C (или равнозначная ей `POSIX`), допускаются все кодировки, но для других настроек правильно будет работать только одна кодировка. `CREATE DATABASE` позволяет суперпользователям указать кодировку `SQL_ASCII` вне зависимости от настроек локали, но этот вариант считается устаревшим и может привести к ошибочному поведению строковых функций, если в базе хранятся данные в кодировке, несовместимой с заданной локалью.

Настройки локали и кодировка должны соответствовать тем, что установлены в шаблоне, если только это не `template0`. Это ограничение объясняется тем, что другие базы данных могут содержать данные в кодировке, отличной от заданной, или индексы, порядок сортировки которых определяются параметрами `LC_COLLATE` и `LC_CTYPE`. При копировании таких данных получится база, которая будет испорченной согласно новым настройкам. Однако `template0` определенно не содержит какие-либо данные или индексы, зависящие от кодировки или локали.

Ограничение `CONNECTION LIMIT` действует только приблизительно; если одновременно запускаются два сеанса, тогда как в базе остается только одно «свободное место», может так случиться, что будут отклонены оба подключения.

Кроме того, это ограничение не распространяется на суперпользователей.

24.5.Примеры

Создание базы данных:

```
CREATE DATABASE lusiadas;
```

Создание базы данных sales, принадлежащей пользователю salesapp, с табличным пространством по умолчанию salespace:

```
CREATE DATABASE sales OWNER salesapp TABLESPACE  
salespace;
```

Создание базы данных music с кодировкой ISO-8859-1:

```
CREATE DATABASE music ENCODING 'LATIN1' TEMPLATE  
template0;
```

В этом примере предложение TEMPLATE template0 будет необходимым, только если кодировка template1 отличается от ISO-8859-1. Заметьте, что при смене кодировки может потребоваться также выбрать другие параметры LC_COLLATE и LC_CTYPE.

24.6.Совместимость

Оператор CREATE DATABASE отсутствует в стандарте SQL. Базы данных равнозначны каталогам, а их создание в стандарте определяется реализацией.

25. CREATE DOMAIN

CREATE DOMAIN – создать домен.

25.1.Синтаксис

```
CREATE DOMAIN имя [ AS ] тип_данных  
[ COLLATE правило_сортировки ]  
[ DEFAULT выражение ]  
[ ограничение [ ... ] ]
```

Здесь ограничение:

```
[ CONSTRAINT имя_ограничения ]  
{ NOT NULL | NULL | CHECK (выражение) }
```

25.2.Описание

CREATE DOMAIN создает новый домен. Домен по сути представляет собой тип данных с дополнительными условиями (ограничивающими допустимый набор значений). Владелец домена становится пользователь его создавший.

Если задается имя схемы (например, CREATE DOMAIN myschema.mydomain ...), домен создается в указанной схеме, в противном случае – в текущей. Имя домена должно быть уникальным среди имен типов и доменов, существующих в этой схеме.

Домены полезны для абстрагирования и вынесения общих характеристик разных полей в единое место для упрощения сопровождения. Например, в нескольких таблицах может присутствовать колонка, содержащая электронный адрес, и для всех требуются одинаковые ограничения CHECK, проверяющие синтаксис адреса. В этом случае лучше определить домен, а не задавать для каждой таблицы отдельные ограничения.

Чтобы создать домен, необходимо иметь право USAGE для нижележащего типа.

25.3.Параметры

имя

Имя (возможно, дополненное схемой) создаваемого домена.

тип_данных

Нижележащий тип данных домена (может включать определение массива с этим типом).

правило_сортировки

Необязательное указание правила сортировки для домена. Если это указание отсутствует, используется правило сортировки по умолчанию нижележащего типа данных. Указать COLLATE можно, только если нижележащий тип данных является сортируемым.

DEFAULT выражение

Предложение DEFAULT определяет значение по умолчанию для колонок,

типом данных которых является этот домен. Значением может быть любое выражение без переменных (подзапросы также не допускаются). Тип данных этого выражения должен соответствовать типу данных домена. Если значение по умолчанию не указано, им будет значение NULL.

Значение по умолчанию будет использоваться в любой операции добавления строк, в которой не задано значение для этой колонки. Если значение по умолчанию установлено для конкретной колонки, оно будет переопределять значение по умолчанию, связанное с доменом. В свою очередь, значение по умолчанию для домена переопределяет любое значение по умолчанию, связанное с нижележащим типом данных.

`CONSTRAINT имя_ограничения`

Имя ограничения. Если не указано явно, имя будет сгенерировано системой.

`NOT NULL`

Такой домен как правило не принимает значения NULL. Однако домен с таким ограничением все же может принимать значение NULL, если ему присваивается такой же тип домена, который оказался равным NULL, например, в результате `LEFT OUTER JOIN` или `INSERT INTO tab (domcol) VALUES ((SELECT domcol FROM tab WHERE false))`.

`NULL`

Этот домен может содержать значение NULL. Это свойство домена по умолчанию.

Это предложение предназначено только для совместимости с нестандартными базами данных SQL. Использовать его в новых приложениях не рекомендуется.

`CHECK (выражение)`

Предложения `CHECK` задают ограничения целостности или проверки, которым должны удовлетворять значения домена. Каждое ограничение должно представлять собой выражение, выдающее результат типа `Boolean`. Проверяемое значение в этом выражении обозначается ключевым словом `VALUE`.

В настоящее время выражения `CHECK` не могут содержать переменные, кроме `VALUE`, и подзапросы.

25.4.Примеры

В этом примере создается тип данных `us_postal_code` (почтовый индекс США), который затем используется в определении таблицы. Для проверки значения на соответствие формату почтовых индексов США применяется проверка с регулярными выражениями:

```
CREATE DOMAIN us_postal_code AS TEXT
CHECK (
    VALUE ~ '^\\d{5}$'
OR VALUE ~ '^\\d{5}-\\d{4}$'
);
```

```
CREATE TABLE us_snail_addy (
    address_id SERIAL PRIMARY KEY,
    street1 TEXT NOT NULL,
    street2 TEXT,
    street3 TEXT,
    city TEXT NOT NULL,
    postal us_postal_code NOT NULL
);
```

25.5.Совместимость

Команда `CREATE DOMAIN` соответствует стандарту SQL.

26. CREATE EVENT TRIGGER

`CREATE EVENT TRIGGER` – создать событийный триггер.

26.1.Синтаксис

```
CREATE EVENT TRIGGER имя
ON событие
[ WHEN переменная_фильтра IN (filter_value [, ... ])
[ AND ... ] ]
EXECUTE PROCEDURE имя_функции()
```

26.2.Описание

CREATE EVENT TRIGGER создает новый событийный триггер. Функция триггера выполняется, когда происходит указанное событие и удовлетворяется связанное с триггером условие WHEN (если такое имеется). Владельцем триггера становится пользователь его создавший.

26.3.Параметры

имя

Имя, назначаемое новому триггеру. Это имя должно быть уникальным в базе данных.

событие

Имя события, при котором срабатывает триггер и вызывается заданная функция.

переменная_фильтра

Имя переменной, применяемой для фильтрации событий. Это указание позволяет ограничить срабатывание триггера подмножеством случаев, в которых он поддерживается. В настоящее время единственно возможное значение параметра переменная_фильтра – TAG.

значение_фильтра

Список значений связанного параметра переменная_фильтра, для которых должен срабатывать триггер. Для переменной TAG это список меток команд (например, 'DROP FUNCTION').

имя_функции

Заданная пользователем функция, объявленная как функция без аргументов и возвращающая тип event_trigger.

26.4.Замечания

Создавать событийные триггеры могут только суперпользователи.

Если ошибочный событийный триггер заблокировал работу с базой данных так, что даже удалить его нельзя, перезапустите сервер в однопользовательском режиме и это можно будет сделать.

26.5.Примеры

Триггер, запрещающий выполнение любой команды DDL:

```
CREATE OR REPLACE FUNCTION abort_any_command()  
  RETURNS event_trigger  
  LANGUAGE plpgsql  
  AS $$  
BEGIN  
  RAISE EXCEPTION 'команда % отключена', tg_tag;  
END;  
$$;  
  
CREATE EVENT TRIGGER abort_ddl ON ddl_command_start  
  EXECUTE PROCEDURE abort_any_command();
```

26.6.Совместимость

Оператор CREATE EVENT TRIGGER отсутствует в стандарте SQL.

27. CREATE EXTENSION

CREATE EXTENSION – установить расширение.

27.1.Синтаксис

```
CREATE EXTENSION [ IF NOT EXISTS ] имя_расширения  
  [ WITH ] [ SCHEMA имя_схемы ]  
  [ VERSION версия ]  
  [ FROM старая_версия ]
```

27.2.Описание

CREATE EXTENSION загружает в текущую базу данных новое расширение.

Расширение с таким именем не должно быть уже загружено.

Загрузка расширения по сути сводится к запуску скрипта расширения. Этот скрипт обычно создает новые SQL-объекты, такие как функции, типы данных, операторы и методы поддержки индексов. CREATE EXTENSION дополнительно записывает идентификаторы всех добавляемых объектов, так что впоследствии их

можно удалить, выполнив команду `DROP EXTENSION`.

Для загрузки расширения требуются те же права, что необходимы для создания составляющих его объектов. Для большинства расширений это означает, что необходимы права владельца базы данных или суперпользователя. Пользователь, запускающий `CREATE EXTENSION`, становится владельцем самого расширения (это требуется для последующих проверок доступа), а также владельцем всех объектов, созданных скриптом расширения.

27.3.Параметры

`IF NOT EXISTS`

Не считать ошибкой, если расширение с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее расширение как-то соотносится с тем, которое могло бы быть создано из указанного скрипта.

имя_расширения

Имя устанавливаемого расширения. Синергия-БД создаст расширение, используя инструкции из файла `SHAREDIR/extension/имя_расширения.control`.

имя_схемы

Имя схемы, в которую будут установлены объекты расширения (подразумевается, что расширение позволяет управлять размещением своих объектов). Указанная схема должна уже существовать. Если имя не указано и в управляющем файле расширения оно так же не задано, для создания объектов используется текущая схема.

Помните, что само расширение не считается принадлежащим какой-либо схеме; имена расширений не дополняются схемой и потому должны быть уникальными во всей базе данных. Однако объекты, принадлежащие расширениям, могут относиться к схемам.

версия

Версия устанавливаемого расширения. Ее можно записать в виде идентификатора или строкового значения. По умолчанию версия считывается из

управляющего файла расширения.

старая_версия

Указание FROM старая_версия может быть добавлено тогда и только тогда, когда устанавливаемое расширение заменяет модуль «старого стиля», представляющий собой просто набор объектов, не упакованный в расширение. С этим указанием CREATE EXTENSION запускает альтернативный установочный скрипт, собирающий все существующие объекты в расширение, а не создающий новые. Учтите, что SCHEMA при этом определяет схему, содержащую эти существующие объекты.

Значение, задаваемое в качестве старой_версии, определяется автором расширения и может меняться, если в расширение нужно преобразовать не одну версию модуля в старом стиле.

27.4.Замечания

Прежде чем вы сможете выполнить CREATE EXTENSION и загрузить расширение в базу данных, необходимо правильно установить сопутствующие файлы расширения.

Расширения, доступные для установки в данный момент, можно найти в системном представлении pg_available_extensions или pg_available_extension_versions.

27.5.Примеры

Установка расширения hstore в текущую базу данных:

```
CREATE EXTENSION hstore;
```

27.6.Совместимость

CREATE EXTENSION является расширением Синергия-БД.

28. CREATE FOREIGN DATA WRAPPER

CREATE FOREIGN DATA WRAPPER – создать новую обертку сторонних данных.

28.1. Синтаксис

```
CREATE FOREIGN DATA WRAPPER имя  
    [ HANDLER функция_обработчик | NO HANDLER ]  
    [ VALIDATOR функция_проверки | NO VALIDATOR ]  
    [ OPTIONS ( параметр 'значение' [, ... ] ) ]
```

28.2. Описание

CREATE FOREIGN DATA WRAPPER создает обертку сторонних данных. Владелец обертки становится создавший её пользователь.

Имя обертки сторонних данных должно быть уникальным в базе данных.

Создавать обертки сторонних данных могут только суперпользователи.

28.3. Параметры

имя

Имя создаваемой обертки сторонних данных.

HANDLER функция_обработчик

В аргументе функция_обработчик указывается имя ранее зарегистрированной функции, которая будет вызываться для получения функций, реализующих обращения к сторонним таблицам. Функция-обработчик не принимает аргументы и возвращает результат типа `fdw_handler`.

Обертку сторонних таблиц можно создать и без функции-обработчика, но через такую обертку нельзя будет использовать сторонние таблицы, хотя объявить их вполне возможно.

VALIDATOR функция_проверки

В аргументе функция_проверки указывается имя ранее зарегистрированной функции, которая будет вызываться для проверки общих параметров, передаваемых обертке сторонних данных, а также параметров сторонних серверов, сопоставлений пользователей и сторонних таблиц, доступных через эту обертку. Если функция проверки не задана или указано NO VALIDATOR, параметры не будут проверяться во время создания объектов. Обертка сторонних данных может игнорировать или не принимать неверные указания параметров во

время выполнения, в зависимости от реализации. Функция проверки должна принимать два аргумента: первый типа `text[]` (в нем содержится массив параметров, хранящихся в системном каталоге), а второй типа `oid` (в нем указывается OID системного каталога с этими параметрами). Возвращаемое значение игнорируется; функция проверки должна сообщать о неверных параметрах, вызывая системную функцию `ereport (ERROR)`.

```
OPTIONS ( параметр 'значение' [, ... ] )
```

Это предложение определяет параметры для создаваемой обертки сторонних данных. Набор допустимых параметров и значений для каждой обертки свой, контроль их правильности осуществляет функция проверки сторонних данных. Имена параметров должны быть уникальными.

28.4.Замечания

Функциональность Синергия-БД по работе со сторонними данными продолжает активно развиваться. На данный момент выполняется только примитивная оптимизация запросов (и по большей части это тоже делает обертка), так что в этом направлении есть поле для улучшения производительности.

28.5.Примеры

Создание бесполезной обертки сторонних данных `dummy`:

```
CREATE FOREIGN DATA WRAPPER dummy;
```

Создание обертки сторонних данных `file` с функцией-обработчиком `file_fdw_handler`:

```
CREATE FOREIGN DATA WRAPPER file HANDLER
file_fdw_handler;
```

Создание обертки сторонних данных `mywrapper` с параметрами:

```
CREATE FOREIGN DATA WRAPPER mywrapper
OPTIONS (debug 'true');
```

28.6.Совместимость

`CREATE FOREIGN DATA WRAPPER` соответствует стандарту ISO/IEC 9075-9 (SQL/MED), за исключением того, что предложения `HANDLER` и `VALIDATOR`

стандартом не предусмотрены, а предложения LIBRARY и LANGUAGE, напротив, не реализованы в Синергия-БД.

Учтите, однако, что функциональность SQL/MED в целом ещё не обеспечивается.

29. CREATE FOREIGN TABLE

CREATE FOREIGN TABLE – создать стороннюю таблицу.

29.1. Синтаксис

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name ( [  
    имя_колонки тип_данных [ OPTIONS ( параметр  
    'значение' [, ... ] ) ] [ COLLATE правило_сортировки ]  
    [ ограничение_колонки [ ... ] ]  
    [, ... ]  
] )
```

```
    SERVER имя_сервера  
    [ OPTIONS ( параметр 'значение' [, ... ] ) ]
```

Здесь ограничение_колонки:

```
[ CONSTRAINT имя_ограничения ]  
{ NOT NULL |  
  NULL |  
  DEFAULT выражение_по_умолчанию }
```

29.2. Описание

CREATE FOREIGN TABLE создает новую стороннюю таблицу в текущей базе данных. Владелец таблицы будет пользователь, выполнивший эту команду.

Если указано имя схемы (например, CREATE FOREIGN TABLE myschema.mytable ...), таблица будет создана в этой схеме. В противном случае она создается в текущей схеме. Имя сторонней таблицы должно отличаться от имен других сторонних и обычных таблиц, последовательностей, индексов, представлений и материализованных представлений, существующих в этой схеме.

CREATE FOREIGN TABLE также автоматически создает составной тип данных, соответствующий одной строке сторонней таблицы. Таким образом, имя

сторонней таблицы не может совпадать с именем существующего типа в этой же схеме.

Чтобы создать стороннюю таблицу, необходимо иметь право USAGE для стороннего сервера, а также право USAGE для всех типов колонок, содержащихся в таблице.

29.3.Параметры

IF NOT EXISTS

Не считать ошибкой, если отношение с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее отношение как-то соотносится с тем, которое могло бы быть создано.

table_name

Имя (возможно, дополненное схемой) создаваемой таблицы.

имя_колонки

Имя колонки, создаваемой в новой таблице.

тип_данных

Тип данных колонки (может включать определение массива с этим типом).

NOT NULL

Данная колонка не принимает значения NULL.

NULL

Данная колонка может содержать значения NULL (по умолчанию).

Это предложение предназначено только для совместимости с нестандартными базами данных SQL. Использовать его в новых приложениях не рекомендуется.

DEFAULT выражение_по_умолчанию

Предложение DEFAULT задает значение по умолчанию для колонки, в определении которой оно присутствует. Значение задается выражением без переменных (подзапросы и перекрестные ссылки на другие колонки текущей таблицы в нем не допускаются). Тип данных выражения, задающего значение по умолчанию, должен соответствовать типу данных колонки.

Это выражение будет использоваться во всех операциях добавления данных, в которых не задается значение данной колонки. Если значение по умолчанию не

определено, таким значением будет NULL.

имя_сервера

Имя существующего стороннего сервера, предоставляющего данную стороннюю таблицу.

OPTIONS (параметр 'значение' [, ...])

Параметры, связываемые с новой сторонней таблицей или одной из её колонок. Допустимые имена и значения параметров у каждой обертки сторонних данных свои; они контролируются функцией проверки, связанной с этой оберткой. Имена параметров не должны повторяться (хотя параметр таблицы и параметр колонки вполне могут иметь одно имя).

29.4.Примеры

Создание сторонней таблицы films, которая будет доступна через сервер film_server:

```
CREATE FOREIGN TABLE films (  
    code          char(5) NOT NULL,  
    title         varchar(40) NOT NULL,  
    did           integer NOT NULL,  
    date_prod     date,  
    kind          varchar(10),  
    len           interval hour to minute  
)  
SERVER film_server;
```

29.5.Совместимость

Команда CREATE FOREIGN TABLE в основном соответствует стандарту SQL; однако, как и CREATE TABLE, она допускает ограничения NULL и сторонние таблицы с нулём столбцов. нулем колонок. Возможность задавать значения по умолчанию для столбцов также является расширением Синергия-БД. Наследование таблиц, в форме, определённой в Синергия-БД, стандарту не соответствует.

30. CREATE FUNCTION

CREATE FUNCTION – создать функцию.

30.1. Синтаксис

```
CREATE [ OR REPLACE ] FUNCTION
    имя ( [ [ режим_аргумента ] [ имя_аргумента ]
тип_аргумента [ { DEFAULT | = } выражение_по_умолчанию ]
[ , ... ] ) )
    [ RETURNS тип_результата
      | RETURNS TABLE ( имя_колонки тип_колонки
[ , ... ] ) ]
    { LANGUAGE имя_языка
      | WINDOW
      | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
      | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT
| STRICT
      | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ]
SECURITY DEFINER
      | COST стоимость_выполнения
      | ROWS строк_в_результате
      | SET параметр_конфигурации { TO значение | =
значение | FROM CURRENT }
      | AS 'определение'
      | AS 'объектный_файл', 'объектный_символ'
    } ...
    [ WITH ( атрибут [ , ... ] ) ]
```

30.2. Описание

Команда CREATE FUNCTION определяет новую функцию. CREATE OR REPLACE FUNCTION создает новую функцию, либо заменяет определение уже существующей. Чтобы определить функцию, необходимо иметь право USAGE для соответствующего языка.

Если указано имя схемы, функция создается в заданной схеме, в противном случае – в текущей. Имя новой функции должно отличаться от имен существующих

функций с такими же типами аргументов в этой схеме. Однако, функции с аргументами разных типов могут иметь одно имя (это называется перегрузкой).

Чтобы заменить текущее определение существующей функции, используйте команду `CREATE OR REPLACE FUNCTION`. Но учтите, что она не позволяет изменить имя или аргументы функции (если попытаться сделать это, на самом деле будет создана новая, независимая функция). Кроме того, `CREATE OR REPLACE FUNCTION` не позволит изменить тип результата существующей функции. Чтобы сделать это, придется удалить функцию и создать её заново. (Это означает, что если функция имеет выходные параметры (OUT), то изменить типы параметров OUT можно, только удалив функцию.)

Когда команда `CREATE OR REPLACE FUNCTION` заменяет существующую функцию, владелец и права доступа к этой функции не меняются. Все другие свойства функции получают значения, задаваемые командой явно или по умолчанию. Чтобы заменить функцию, необходимо быть её владельцем (или быть членом роли-владельца).

Если вы удалите и затем вновь создадите функцию, новая функция станет другой сущностью, отличной от старой; вам потребуется так же удалить существующие правила, представления, триггеры и т.п., ссылающиеся на старую функцию. Поэтому, чтобы изменить определение функции, сохраняя ссылающиеся на нее объекты, следует использовать `CREATE OR REPLACE FUNCTION`. Кроме того, многие дополнительные свойства существующей функции можно изменить с помощью `ALTER FUNCTION`.

Владельцем функции становится пользователь её создавший.

Чтобы создать функцию, необходимо иметь право `USAGE` для типов её аргументов и возвращаемого типа.

30.3.Параметры

имя

Имя (возможно, дополненное схемой) создаваемой функции.

режим_аргумента

Режим аргумента: IN (входной), OUT (выходной), INOUT (входной и выходной) или VARIADIC (переменный). По умолчанию подразумевается IN. За единственным аргументом VARIADIC могут следовать только аргументы OUT. Кроме того, аргументы OUT и INOUT нельзя использовать с предложением RETURNS TABLE.

имя_аргумента

Имя аргумента. Некоторые языки (включая SQL и PL/pgSQL) позволяют использовать это имя в теле функции. Для других языков это имя служит просто дополнительным описанием, если говорить о самой функции; однако вы можете указывать имена аргументов при вызове функции для улучшения читаемости. Имя выходного аргумента в любом случае имеет значение, так как оно определяет имя колонки в типе результата. Если вы опустите имя выходного аргумента, система выберет для него имя по умолчанию.

тип_аргумента

Тип(ы) данных аргументов функции (возможно, дополненный схемой), при наличии аргументов. Тип аргументов может быть базовым, составным или доменным, либо это может быть ссылка на колонку таблицы.

В зависимости от языка реализации также может допускаться указание «псевдотипов», например, `cstring`. Псевдотипы показывают, что фактический тип аргумента либо определен не полностью, либо существует вне множества обычных типов SQL.

Ссылка на тип колонки записывается в виде `имя_таблицы.имя_колонки%TYPE`. Иногда такое указание бывает полезно, так как позволяет создать функцию, независимую от изменений в определении таблицы.

выражение_по_умолчанию

Выражение, используемое для вычисления значения по умолчанию, если параметр не задан явно. Результат выражения должен сводиться к типу соответствующего параметра. Значения по умолчанию могут иметь только входные параметры (включая INOUT). Для всех входных параметров, следующих за параметром с определенным значением по умолчанию, также должны быть

определены значения по умолчанию.

тип_результата

Тип возвращаемых данных (возможно, дополненный схемой). Это может быть базовый, составной или доменный тип, либо ссылка на тип колонки таблицы. В зависимости от языка реализации здесь также могут допускаться «псевдотипы», например `cstring`. Если функция не должна возвращать значение, в качестве типа результата указывается `void`.

В случае наличия параметров `OUT` или `INOUT`, предложение `RETURNS` можно опустить. Если оно присутствует, оно должно согласовываться с типом результата, выводимым из выходных параметров: в качестве возвращаемого типа указывается `RECORD`, если выходных параметров несколько, либо тип единственного выходного параметра.

Указание `SETOF` показывает, что функция возвращает множество, а не единственный элемент.

Ссылка на тип колонки записывается в виде
`имя_таблицы.имя_колонки%TYPE`.

имя_колонки

Имя выходной колонки в записи `RETURNS TABLE`. По сути это ещё один способ объявить именованный выходной параметр (`OUT`), но `RETURNS TABLE` также подразумевает и `RETURNS SETOF`.

тип_колонки

Тип данных выходной колонки в записи `RETURNS TABLE`.

имя_языка

Имя языка, на котором реализована функция. Это может быть `sql`, `c`, `internal`, либо имя процедурного языка, определенного пользователем, например, `plpgsql`. Стиль написания этого имени в апострофах считается устаревшим и требует точного совпадения регистра.

WINDOW

Указание `WINDOW` показывает, что создается не простая, а оконная функция. В настоящее время это имеет смысл только для функций, написанных на `C`. Атрибут

WINDOW нельзя изменить, модифицируя впоследствии определение функции.

IMMUTABLE
STABLE
VOLATILE

Эти атрибуты информируют оптимизатор запросов о поведении функции. Одновременно можно указать не более одного атрибута. Если никакой атрибут не задан, по умолчанию подразумевается VOLATILE.

Характеристика IMMUTABLE (постоянная) показывает, что функция не может модифицировать базу данных и всегда возвращает один и тот же результат при определенных значениях аргументов; то есть, она не обращается к базе данных и не использует информацию, не переданную ей явно в списке аргументов. Если функция имеет такую характеристику, любой её вызов с аргументами-константами можно немедленно заменить значением функции.

Характеристика STABLE (стабильная) показывает, что функция не может модифицировать базу данных и в рамках одного сканирования таблицы она всегда возвращает один и тот же результат для определенных значений аргументов, но этот результат может быть разным в разных операторах SQL. Это подходящий выбор для функций, результаты которых зависят от содержимого базы данных и настраиваемых параметров (например, текущего часового пояса). Но этот вариант не подходит для триггеров AFTER, желающих прочитать строки, измененные текущей командой. Также заметьте, что функции семейства `current_timestamp` также считаются стабильными, так как их результаты не меняются внутри транзакции.

Характеристика VOLATILE (изменчивая) показывает, что результат функции может меняться даже в рамках одного сканирования таблицы, так что её вызовы нельзя оптимизировать. Изменчивы в этом смысле относительно немногие функции баз данных, например: `random()`, `currval()` и `timeofday()`. Но заметьте, что любая функция с побочными эффектами должна быть классифицирована как изменчивая, даже если её результат вполне предсказуем, чтобы её вызовы не были сооптимизированы; пример такой функции: `setval()`.

LEAKPROOF

Характеристика `LEAKPROOF` (герметичная) показывает, что функция не имеет побочных эффектов. Она не раскрывает информацию о своих аргументах, кроме как возвращая результат. Например, функция, которая выдает сообщение об ошибке с некоторыми, но не всеми значениями аргументов, либо выводит значения аргументов в сообщении об ошибке, не является герметичной. Планировщик запросов может поместить герметичные (и только такие) функции в представления, созданные с барьером безопасности (с указанием `security_barrier`). Это свойство может установить только суперпользователь.

```
CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT
```

`CALLED ON NULL INPUT` (по умолчанию) показывает, что функция будет вызвана как обычно, если среди её аргументов оказываются значения `NULL`. В этом случае ответственность за проверку значений `NULL` и соответствующую их обработку ложится на разработчика функции.

Указание `RETURNS NULL ON NULL INPUT` или `STRICT` показывает, что функция всегда возвращает `NULL`, получив `NULL` в одном из аргументов. Такая функция не будет вызываться с аргументами `NULL`, вместо этого автоматически будет полагаться результат `NULL`.

```
[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER
```

Характеристика `SECURITY INVOKER` (безопасность вызывающего) показывает, что функция будет выполняться с правами пользователя, вызвавшего ее. Этот вариант подразумевается по умолчанию. Вариант `SECURITY DEFINER` (безопасность определившего) определяет, что функция выполняется с правами пользователя, создавшего ее.

Ключевое слово `EXTERNAL` (внешняя) допускается для соответствия стандарту SQL, но является необязательным, так как, в отличие от SQL, эта характеристика распространяется на все функции, а не только внешние.

```
СТОИМОСТЬ_выполнения
```

Положительное число, задающее примерную стоимость выполнения функции,

в единицах `cpu_operator_cost`. Если функция возвращает множество, это число задает стоимость для одной строки. Если стоимость не указана, для функций на С и внутренних функций она считается равной 1 единице, а для функций на всех других языках – 100 единицам. При больших значениях планировщик будет стараться не вызывать эту функцию чаще, чем это необходимо.

`строка_в_результате`

Положительное число, задающее примерное число строк, которое будет ожидать планировщик на выходе этой функции. Это указание допустимо, только если функция объявлена как возвращающая множество. Предполагаемое по умолчанию значение – 1000 строк.

`параметр_конфигурации`
`значение`

Предложение SET определяет, что при вызове функции указанный параметр конфигурации должен принять заданное значение, а затем восстановить свое предыдущее значение при завершении функции. Предложение SET FROM CURRENT сохраняет в качестве значения, которое будет применено при входе в функцию, текущее значение указанного параметра.

Если в определение функции добавлено SET, то действие команды SET LOCAL, выполняемой внутри функции для того же параметра, ограничивается телом функции: предыдущее значение параметра неизменно будет восстановлено при завершении функции. Однако обычная команда SET (без LOCAL) переопределяет предложение SET, как и предыдущую команду SET LOCAL: действие такой команды будет сохранено и после завершения функции, если только не произойдет откат транзакции.

`определение`

Строковая константа, определяющая реализацию функции; её значение зависит от языка. Это может быть имя внутренней функции, путь к объектному файлу, команда SQL или код функции на процедурном языке.

Часто бывает полезно заключать определение функции в доллары, а не в традиционные апострофы. Если не использовать доллары, все апострофы и

обратные косые черты в определении функции придется экранировать, дублируя их.

`объектный_файл, объектный_символ`

Эта форма предложения `AS` применяется для динамически загружаемых функций на языке `C`, когда имя функции в коде `C` не совпадает с именем функции в `SQL`. Строка `объектный_файл` задает имя файла, содержащего динамически загружаемый объект, а `объектный_символ` – символ скомпонованной функций, то есть имя функции в исходном коде на языке `C`. Если `объектный_символ` опущен, предполагается, что он совпадает с именем определяемой `SQL`-функции.

Если повторные вызовы `CREATE FUNCTION` ссылаются на один и тот же `объектный_файл`, файл загружается в рамках сеанса только один раз. Чтобы выгрузить и загрузить файл снова (например, в процессе разработки), начните новый сеанс.

атрибут

Исторически сложившийся способ передавать дополнительную информацию о функции. В этой записи могут присутствовать следующие атрибуты:

`isStrict`

Равнозначно указанию `STRICT` или `RETURNS NULL ON NULL INPUT`.

`isCachable`

Свойство `isCachable` – устаревший эквивалент `IMMUTABLE`; оно все ещё поддерживается ради обратной совместимости.

Имена атрибутов являются регистронезависимыми.

30.4. Перегрузка

Синергия-БД допускает перегрузку функций; то есть, позволяет использовать одно имя для нескольких различных функций, если у них различаются типы входных аргументов. Однако имена функций в коде `C` должны различаться, так что перегружаемым функциям `C` нужно давать различные имена в `C` (например, включить в имя функции на `C` типы аргументов).

Две функции считаются совпадающими, если они имеют одинаковые имена и типы входных аргументов, параметры `OUT` игнорируются. Таким образом, например,

эти объявления вызовут конфликт:

```
CREATE FUNCTION foo(int) ...
```

```
CREATE FUNCTION foo(int, out text) ...
```

Функции, имеющие разные типы аргументов, не будут считаться конфликтующими в момент создания, но предоставленные для них значения по умолчанию могут вызвать конфликт в момент использования. Например, рассмотрите следующие определения:

```
CREATE FUNCTION foo(int) ...
```

```
CREATE FUNCTION foo(int, int default 42) ...
```

Вызов `foo(10)` завершится ошибкой из-за неоднозначности в выборе вызываемой функции.

30.5.Замечания

В объявлении аргументов функции и возвращаемого значения допускается полный синтаксис описания типа SQL. Однако модификаторы типа в скобках (например, поле точности для типа `numeric`) команда `CREATE FUNCTION` не учитывает. Так что, например, `CREATE FUNCTION foo (varchar(10)) ...` создаст такую же функцию, что и `CREATE FUNCTION foo (varchar) ....`

При замене существующей функции с помощью `CREATE OR REPLACE FUNCTION` есть ограничения на изменения имен параметров. В частности, нельзя изменить имя, уже назначенное любому входному параметру (хотя можно добавить имена ранее безымянным параметрам). Также, если у функции более одного выходного параметра, нельзя изменять имена выходных параметров, так как это приведет к изменению имен колонок анонимного составного типа, описывающего результат функции. Эти ограничения позволяют гарантировать, что существующие вызовы функции не перестанут работать после её замены.

Если функция объявлена как `STRICT` с аргументом `VARIADIC`, при оценивании строгости проверяется, что весь переменный массив в целом не `NULL`. Если же в этом массиве содержатся элементы `NULL`, функция будет вызываться.

30.6.Примеры

Ниже приведено несколько простых вводных примеров.

```
CREATE FUNCTION add(integer, integer) RETURNS integer
  AS 'select $1 + $2;'
  LANGUAGE SQL
  IMMUTABLE
  RETURNS NULL ON NULL INPUT;
```

Функция увеличения целого числа на 1, использующая именованный аргумент, на языке PL/pgSQL:

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS
integer AS $$
  BEGIN
    RETURN i + 1;
  END;
$$ LANGUAGE plpgsql;
```

Функция, возвращающая запись с несколькими выходными параметрами:

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
  AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
  LANGUAGE SQL;
```

```
SELECT * FROM dup(42);
```

То же самое можно сделать более развернуто, явно объявив составной тип:

```
CREATE TYPE dup_result AS (f1 int, f2 text);
```

```
CREATE FUNCTION dup(int) RETURNS dup_result
  AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
  LANGUAGE SQL;
```

```
SELECT * FROM dup(42);
```

Ещё один способ вернуть несколько колонок – применить функцию TABLE:

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
  AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
```

```
LANGUAGE SQL;
```

```
SELECT * FROM dup(42);
```

Однако пример с TABLE отличается от предыдущих, так как в нем функция на самом деле возвращает не одну, а набор записей.

Разработка защищенных функций SECURITY DEFINER.

Так как функция SECURITY DEFINER выполняется с правами пользователя, создавшего ее, необходимо позаботиться о том, чтобы её нельзя было использовать не по назначению. В целях безопасности, в пути search_path следует исключить любые схемы, доступные на запись недоверенным пользователям. Это не позволит злонамеренным пользователям создать свои объекты, которые замаскируют объекты, используемые функцией. Особенно важно в этом отношении отключить схему временных таблиц, которая по умолчанию просматривается первой, а право записи в нее по умолчанию имеют все. Соответствующую защиту можно организовать, поместив временную схему в конец списка поиска. Для этого следует сделать pg_temp последней записью в search_path. Такую защиту демонстрирует следующая функция:

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN;
BEGIN
    SELECT (pwd = $2) INTO passed
    FROM   pwds
    WHERE  username = $1;

    RETURN passed;
END;
$$ LANGUAGE plpgsql
SECURITY DEFINER
-- Установить безопасный путь поиска: сначала
доверенная схема(a), затем 'pg_temp'.
SET search_path = admin, pg_temp;
```

Также следует помнить о том, что по умолчанию право выполнения для создаваемых функций имеет роль PUBLIC (за подробностями обратитесь к GRANT). Однако часто требуется разрешить доступ к функциям, работающим в контексте определившего, только некоторым пользователям. Для этого необходимо отозвать стандартные права PUBLIC и затем дать права на выполнение индивидуально. Чтобы не образовалось окно, в котором новая функция будет недоступна совсем, создайте её и назначьте права в одной транзакции, например:

```
BEGIN;
CREATE FUNCTION check_password(uname TEXT, pass
TEXT) ... SECURITY DEFINER;
REVOKE ALL ON FUNCTION check_password(uname TEXT, pass
TEXT) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION check_password(uname TEXT,
pass TEXT) TO admins;
COMMIT;
```

30.7. Совместимость

Команда CREATE FUNCTION определена в SQL:1999 и более поздних стандартах. Версия, реализованная в Синергия-БД, близка к стандартизированной, но соответствует ей не полностью. В частности, непереносимы атрибуты, а также различные языки реализаций.

Для совместимости с другими СУБД режим_аргумента можно записать после имя_аргумента или перед ним, но стандарту соответствует только первый вариант.

Для определения значений по умолчанию для параметров стандарт SQL поддерживает только синтаксис с ключевым словом DEFAULT. Синтаксис со знаком = используется в T-SQL и Firebird.

31. CREATE GROUP

CREATE GROUP – создать роль в базе данных.

31.1. Синтаксис

```
CREATE GROUP имя [ [ WITH ] параметр [ ... ] ]
```

Здесь параметр:

```
SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| CREATEUSER | NOCREATEUSER
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
| VALID UNTIL 'дата_время'
| IN ROLE имя_роли [, ...]
| IN GROUP имя_роли [, ...]
| ROLE имя_роли [, ...]
| ADMIN имя_роли [, ...]
| USER имя_роли [, ...]
| SYSID uid
```

31.2.Описание

Оператор CREATE GROUP теперь является синонимом оператора CREATE ROLE.

31.3.Совместимость

Оператор CREATE GROUP отсутствует в стандарте SQL.

32. CREATE INDEX

CREATE INDEX – создать индекс.

32.1.Синтаксис

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ имя ] ON
table_name [ USING метод ]
( { имя_колонки | ( выражение ) } [ COLLATE
правило_сортировки ] [ класс_операторов ] [ ASC | DESC ]
[ NULLS { FIRST | LAST } ] [, ...] )
[ WITH ( параметр_хранения = значение [, ...] ) ]
[ TABLESPACE табл_пространство ]
```

32.2.Описание

CREATE INDEX создает индексы по указанной колонке(ам) заданного отношения, которым может быть таблица или материализованное представление. Индексы применяются в первую очередь для оптимизации производительности базы данных (хотя при неправильном использовании возможен и противоположный эффект).

Ключевое поле для индекса задается как имя колонки или выражение, заключенное в скобки. Если метод индекса поддерживает многоколоночные индексы, допускается указание нескольких полей.

Поле индекса может быть выражением, вычисляемым из значений одной или нескольких колонок в строке таблицы. Это может быть полезно для получения быстрого доступа к данным по некоторому преобразованию исходных значений. Например, индекс, построенный по выражению `upper(col)`, позволит использовать поиск по индексу в предложении `WHERE upper(col) = 'JIM'`.

Синергия-БД предоставляет следующие методы индексов: В-дерево, хеш, GiST, SP-GiST и GIN. Пользователи могут определить и собственные методы индексов, но это довольно сложная задача.

Если в команде присутствует предложение WHERE, она создает частичный индекс. Такой индекс содержит записи только для части таблицы, обычно более полезной для индексации, чем остальная таблица. Например, если таблица содержит информацию об оплаченных и неоплаченных счетах, при этом последних сравнительно немного, но именно эта часть таблицы наиболее востребована, то увеличить быстродействие можно, создав индекс только по этой части. Ещё одно возможное применение WHERE — добавив UNIQUE, обеспечить уникальность в подмножестве таблицы.

Выражение в предложении WHERE может ссылаться только на колонки нижележащей таблицы, но не обязательно ограничиваться теми, по которым строится индекс. В настоящее время в WHERE также нельзя использовать

подзапросы и агрегатные выражения. Это же ограничение распространяется и на выражения в полях индексов.

Все функции и операторы, используемые в определении индекса, должны быть «постоянными», то есть, их результаты должны зависеть только от аргументов, но не от внешних факторов (например, содержимого другой таблицы или текущего времени). Это ограничение обеспечивает определенность поведения индекса. Чтобы использовать в выражении индекса или в предложении WHERE собственную функцию, не забудьте пометить её при создании как постоянную.

32.3.Параметры

UNIQUE

Указывает, что система должна контролировать повторяющиеся значения в таблице при создании индекса (если в таблице уже есть данные) и при каждом добавлении данных. Попытки вставить или изменить данные, при которых будет нарушена уникальность индекса, будут завершаться ошибкой.

CONCURRENTLY

С этим указанием Синергия-БД построит индекс, не устанавливая никаких блокировок, которые бы предотвращали добавление, изменение или удаление записей в таблице; тогда как по умолчанию операция построения индекса блокирует запись (но не чтение) в таблице до своего завершения.

ИМЯ

Имя создаваемого индекса. Указание схемы при этом не допускается; индекс всегда относится к той же схеме, что и родительская таблица. Если имя опущено, Синергия-БД формирует подходящее имя по имени родительской таблицы и именам индексируемых колонок.

table_name

Имя (возможно, дополненное схемой) индексируемой таблицы.

метод

Имя применяемого метода индекса. Возможные варианты: btree, hash, gist, spgist и gin. По умолчанию подразумевается метод btree.

ИМЯ_КОЛОНКИ

Имя колонки таблицы.

выражение

Выражение с одной или несколькими колонками таблицы. Обычно выражение должно записываться в скобках, как показано в синтаксисе команды. Однако скобки можно опустить, если выражение записано в виде вызова функции.

правило_сортировки

Имя правила сортировки, применяемого для индекса. По умолчанию используется правило сортировки, заданное для индексируемой колонки, либо полученное для результата выражения индекса. Индексы с нестандартными правилами сортировки могут быть полезны для запросов, включающих выражения с такими правилами.

класс_операторов

Имя класса операторов.

ASC

Указывает порядок сортировки по возрастанию (подразумевается по умолчанию).

DESC

Указывает порядок сортировки по убыванию.

NULLS FIRST

Указывает, что значения NULL после сортировки оказываются перед остальными. Это поведение по умолчанию с порядком сортировки DESC.

NULLS LAST

Указывает, что значения NULL после сортировки оказываются после остальных. Это поведение по умолчанию с порядком сортировки ASC.

параметр_хранения

Имя специфичного для индекса параметра хранения.

табл_пространство

Табличное пространство, в котором будет создан индекс. Если не определено, выбирается default_tablespace, либо temp_tablespaces, при создании индекса временной таблицы.

предикат

Выражение ограничения для частичного индекса.

32.4. Параметры хранения индекса

Необязательное предложение `WITH` определяет параметры хранения для индекса. У каждого метода индекса есть свой набор допустимых параметров хранения. Следующий параметр принимают методы В-дерево, хеш, GiST и SP-GiST:

`FILLFACTOR`

Фактор заполнения для индекса определяет в процентном отношении, насколько плотно метод индекса будет заполнять страницы индекса. Для В-деревьев концевые страницы заполняются до этого процента при начальном построении индекса и позже, при расширении индекса вправо (добавлении новых наибольших значений ключа). Если страницы впоследствии оказываются заполненными полностью, они будут разделены, что приводит к постепенному снижению эффективности индекса. Для В-деревьев по умолчанию используется фактор заполнения 90, но его можно поменять на любое целое значение от 10 до 100. Фактор заполнения, равный 100, полезен для статических таблиц и помогает уменьшить физический размер таблицы, но для интенсивно изменяемых таблиц лучше использовать меньшее значение, чтобы разделять страницы приходилось реже. С другими методами индекса фактор заполнения действует по-другому, но примерно в том же ключе; значение фактора заполнения по умолчанию для разных методов разное.

Индексы GiST дополнительно принимают этот параметр:

`BUFFERING`

Определяет, будет ли при построении индекса использоваться буферизация. Со значением `OFF` она отключена, с `ON` – включена, а с `AUTO` – отключена вначале, но может затем включиться на ходу, как только размер индекса достигнет значения `effective_cache_size`. По умолчанию подразумевается `AUTO`.

Индексы GIN принимают другой параметр:

`FASTUPDATE`

Этот параметр управляет механизмом быстрого обновления. Он имеет

логическое значение: ON включает быстрое обновление, OFF отключает его.

Значение по умолчанию – ON.

Примечание. Выключение FASTUPDATE в ALTER INDEX предотвращает помещение добавляемых в дальнейшем строк в список записей, ожидающих индексации, но записи, добавленные в этот список ранее, в нем остаются. Чтобы очистить очередь операций, надо затем выполнить VACUUM для этой таблицы.

32.5.Параллельное построение индексов

Создание индекса может мешать обычной работе с базой данных. Обычно Синергия-БД блокирует индексируемую таблицу от записи и выполняет всю операцию построения индекса за одно сканирование таблицы. Другие транзакции могут продолжать читать таблицу, но при попытке вставить, изменить или удалить строки в таблице они будут заблокированы до завершения построения индекса. Это может оказать нежелательное влияние на работу производственной базы данных. Индексация очень больших таблиц может занимать много часов, и даже для маленьких таблиц построение индекса может заблокировать записывающие процессы на время, неприемлемое для производственной системы.

Синергия-БД поддерживает построение индексов без блокировки записи. Этот метод выбирается указанием CONCURRENTLY команды CREATE INDEX. Когда он используется, Синергия-БД должна выполнить два сканирования таблицы, а кроме того, должна дожидаться завершения всех существующих транзакций, которые потенциально могут использовать этот индекс. Таким образом, эта процедура требует проделать в сумме больше действий и выполняется значительно дольше, чем обычное построение индекса. Однако благодаря тому, что этот метод позволяет продолжать обычную работу с базой во время построения индекса, он оказывается полезным в производственной среде. Хотя разумеется, дополнительная нагрузка на процессор и подсистему ввода/вывода, создаваемая при построении индекса, может привести к замедлению других операций.

При параллельном построении индекса он попадает в системный каталог в одной транзакции, затем ещё два сканирования таблицы выполняются в двух других транзакциях. Любая транзакция, активная в момент начала второго сканирования

таблицы, может заблокировать параллельное создание индекса до своего окончания, даже если эта транзакция обратилась к таблице уже после начала второго сканирования. Процедура параллельного создания индекса последовательно ожидает завершения всех старых транзакций.

Если при сканировании таблицы возникает проблема, например взаимоблокировка или нарушение уникальности в уникальном индексе, команда `CREATE INDEX` завершится ошибкой, но оставит после себя «нерабочий» индекс. Этот индекс будет игнорироваться при чтении данных, так как он может быть неполным; однако с ним могут быть связаны дополнительные операции при изменениях. В `psql` встроенная команда `\d` помечает такой индекс как `INVALID`:

```
postgres=# \d tab
          Table «public.tab»
   Column |   Type   | Modifiers
-----+-----+-----
   col    | integer |
Indexes:
    «idx» btree (col) INVALID
```

Рекомендуемый в таких случаях способ исправления ситуации – удалить индекс и затем попытаться снова выполнить `CREATE INDEX CONCURRENTLY`. Кроме того, можно перестроить его с помощью команды `REINDEX`. Но так как `REINDEX` не поддерживает параллельный режим, вряд ли этот вариант будет желательным.

Ещё одна сложность, с которой можно столкнуться при параллельном построении уникального индекса, заключается в том, что ограничение уникальности уже влияет на другие транзакции, когда второе сканирование таблицы только начинается. Это значит, что нарушения ограничения могут проявляться в других запросах до того, как индекс становится доступным для использования и даже тогда, когда создать индекс в итоге не удастся. Кроме того, если при втором сканировании происходит ошибка, «нерабочий» индекс оставляет в силе свое ограничение уникальности и дальше.

Метод параллельного построения поддерживает также индексы выражений и частичные индексы. Ошибки, произошедшие при вычислении этих выражений, могут привести к такому же поведению, как в вышеописанных случаях с нарушением ограничений уникальности.

Обычное построение индекса допускает одновременное построение других индексов обычным методом, но параллельное построение в один момент времени допускается только одно. В обоих случаях, никакие изменения типов в схеме таблицы в это время не разрешаются. Другое отличие состоит в том, что в блоке транзакции может быть выполнена обычная команда `CREATE INDEX`, но не `CREATE INDEX CONCURRENTLY`.

32.6.Замечания

Операции с хеш-индексами в настоящее время не проходят через WAL, так что после аварийной остановки базы данных может потребоваться перестроить хеш-индексы командой `REINDEX`. Кроме того, изменения в хеш-индексах после начальной копии не переносятся при потоковой или файловой репликации, так что в последующих запросах они будут давать неправильные ответы. По этим причинам настоятельно рекомендуется не использовать их.

В настоящее время многоколоночные индексы поддерживают только методы В-дерево, GiST и GIN. По умолчанию такой индекс может включать до 32 полей. Этот предел можно изменить, пересобрав Синергия-БД. Уникальные индексы поддерживает только В-дерево.

Для каждой колонки индекса можно задать класс операторов. Этот класс определяет, какие операторы будут использоваться индексом для этой колонки. Например, индекс типа В-дерево по четырехбайтовым целым будет использовать класс `int4_ops`; этот класс операторов включает функции сравнения для таких значений. На практике обычно достаточно использовать класс операторов по умолчанию для типа данных колонки. Существование классов операторов объясняется в первую очередь тем, что для некоторых типов данных можно предложить более одного осмысленного порядка сортировки. Например, может

возникнуть желание отсортировать комплексные числа как по абсолютному значению, так и по вещественной части. Это можно сделать, определив два класса операторов для типа данных и выбрав подходящий класс при создании индекса.

Для методов индекса, поддерживающих сканирование по порядку (в настоящее время это поддерживает только В-дерево), можно изменить порядок сортировки индекса, добавив необязательные предложения ASC, DESC, NULLS FIRST или NULLS LAST. Так как упорядоченный индекс можно сканировать вперед или назад, обычно не имеет смысла создавать индекс по убыванию (DESC) для одной колонки – этот порядок сортировки можно получить и с обычным индексом. Эти параметры имеют смысл при создании многоколоночных индексов так, что они будут соответствовать порядку сортировки, указанному в запросе со смешанным порядком, например `SELECT ... ORDER BY x ASC, y DESC`. Параметры NULLS полезны, когда требуется реализовать поведение «NULL внизу», изменив стандартное «NULL вверху», в запросах, зависящих от индексов, чтобы избежать дополнительной сортировки.

Для большинства методов индексов скорость создания индекса зависит от значения `maintenance_work_mem`. При увеличении этого значения индекс будет создаваться быстрее, пока оно не превысит действительно доступный объем памяти, что приведет к использованию подкачки. Для хеш-индексов значение `effective_cache_size` также влияет на время создания индекса: Синергия-БД будет использовать один из двух различных методов создания индекса, в зависимости от того, больше или меньше оцениваемый размер индекса, чем `effective_cache_size`. Для наилучших результатов этот параметр нужно устанавливать в зависимости от объема свободной памяти, но так, чтобы сумма `maintenance_work_mem` и `effective_cache_size` была меньше объема ОЗУ за вычетом объема, необходимого другим программам.

32.7.Примеры

Создание индекса в виде В-дерева по колонке `title` в таблице `films`:

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

Создание индекса по выражению `lower(title)`, позволяющего эффективно выполнять регистронезависимый поиск:

```
CREATE INDEX ON films ((lower(title)));
```

В этом примере мы решили опустить имя индекса, чтобы имя выбрала система, например `films_lower_idx`.

Создание индекса с нестандартным правилом сортировки:

```
CREATE INDEX title_idx_german ON films (title COLLATE  
«de_DE»);
```

Создание индекса с нестандартным порядком значений NULL:

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS  
FIRST);
```

Создание индекса с нестандартным фактором заполнения:

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH  
(fillfactor = 70);
```

Создание индекса GIN с отключенным механизмом быстрого обновления:

```
CREATE INDEX gin_idx ON documents_table USING gin  
(locations) WITH (fastupdate = off);
```

Создание индекса по колонке `code` в таблице `films` и размещение его в табличном пространстве `indexspace`:

```
CREATE INDEX code_idx ON films (code) TABLESPACE  
indexspace;
```

Создание индекса GiST по координатам точек, позволяющего эффективно использовать операторы `box` с результатом функции преобразования:

```
CREATE INDEX pointloc  
ON points USING gist (box(location,location));  
SELECT * FROM points  
WHERE box(location,location) && '(0,0),(1,1)::box;
```

Создание индекса без блокировки записи в таблицу:

```
CREATE INDEX CONCURRENTLY sales_quantity_index ON  
sales_table (quantity);
```

32.8.Совместимость

CREATE INDEX является языковым расширением Синергия-БД. Средства обеспечения индексов в стандарте SQL не описаны.

33. CREATE LANGUAGE

CREATE LANGUAGE – создать процедурный язык.

33.1.Синтаксис

```
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE имя
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ]
LANGUAGE имя
        HANDLER обработчик_вызова [ INLINE
обработчик_внедренного_кода ] [ VALIDATOR
функция_проверки ]
```

33.2.Описание

CREATE LANGUAGE регистрирует в базе данных Синергия-БД новый процедурный язык. Впоследствии на этом языке можно будет определять новые функции и триггерные процедуры.

CREATE LANGUAGE по сути связывает имя языка с функциями-обработчиками, исполняющими код, написанный на этом языке.

Команда CREATE LANGUAGE имеет две формы. В первой форме пользователь задает только имя создаваемого языка, и Синергия-БД получает подходящие параметры, обращаясь к системному каталогу pg_pltemplate. Во второй форме пользователь указывает параметры языка вместе с его именем. Вторая форма позволяет создать язык, не определенный заранее в pg_pltemplate, но этот подход считается устаревшим.

Когда сервер находит в каталоге pg_pltemplate запись для заданного имени языка, он будет использовать данные из каталога, даже если параметры языка указаны в команде. Это поведение упрощает загрузку экспортированных ранее файлов, которые, скорее всего, будут содержать устаревшую информацию о функциях поддержки языка.

Обычно, для того чтобы зарегистрировать новый язык, необходимо иметь право суперпользователя Синергия-БД. Однако владелец базы данных может зарегистрировать новый язык в этой базе данных, если язык определен в каталоге `pg_pltemplate` и помечен как допускающий создание владельцами БД (`tmpldbacreate` содержит `true`). По умолчанию доверенные языки могут создаваться владельцами базы данных, но это можно изменить, внося коррективы в `pg_pltemplate`. Создатель языка становится его владельцем и может впоследствии удалить или переименовать его, либо назначить другого владельца.

`CREATE OR REPLACE LANGUAGE` создаст новый язык, либо заменит существующее определение. Если язык уже существует, его параметры изменяются в соответствии со значениями, указанными в команде или полученными из `pg_pltemplate`, при этом владелец языка и права доступа к нему не меняются, и все существующие функции, написанные на этом языке, по-прежнему считаются рабочими. Помимо обладания обычными правами, требующимися для создания языка, пользователь должен быть суперпользователем или владельцем существующего языка. Вариант с `REPLACE` в основном предназначен для случаев, когда язык уже существует. Если для языка есть запись в `pg_pltemplate`, то `REPLACE` фактически ничего не меняет в существующем определении, кроме исключительного случая, когда запись в `pg_pltemplate` была изменена после создания языка.

33.3.Параметры

`TRUSTED`

Указание `TRUSTED` (доверенный) означает, что язык не дает пользователю доступ к данным, к которым он не имел бы доступа без него. Если при регистрации языка это ключевое слово опущено, использовать этот язык для создания новых функций смогут только суперпользователи Синергия-БД.

`PROCEDURAL`

Это слово не несет смысловой нагрузки.

ИМЯ

Имя процедурного языка. Оно должно быть уникальным среди всех языков в базе данных.

В целях обратной совместимости имя можно заключить в апострофы.

`HANDLER` обработчик_вызова

Здесь `обработчик_вызова` – имя ранее зарегистрированной функции, которая будет вызываться для исполнения процедур (функций) на этом языке. Обработчик вызова для процедурного языка должен быть написан на компилируемом языке, например, на C, с соглашениями о вызовах версии 1 и зарегистрирован в Синергия-БД как функция без аргументов, возвращающая фиктивный тип `language_handler`, который просто показывает, что эта функция – обработчик вызова.

`INLINE` обработчик_внедренного_кода

Здесь `обработчик_внедренного_кода` – имя ранее зарегистрированной функции, которая будет вызываться для выполнения анонимного блока кода (команды `DO`) на этом языке. Если `обработчик_внедренного_кода` не определен, данный язык не будет поддерживать анонимные блоки кода. Этот обработчик должен принимать один аргумент типа `internal`, содержащий внутреннее представление команды `DO`, и обычно возвращает тип `void`. Значение, возвращаемое этим обработчиком, игнорируется.

`VALIDATOR` функция_проверки

Здесь `функция_проверки` – имя ранее зарегистрированной функции, которая будет вызываться при создании новой функции на этом языке и проверять ее. Если функция проверки не задана, функции на этом языке при создании проверяться не будут. Функция проверки должна принимать один аргумент типа `oid` с идентификатором создаваемой функции и обычно возвращает `void`.

Функция проверки обычно проверяет тело создаваемой функции на синтаксические ошибки, но может также проанализировать и другие характеристики функции, например, поддержку определенных типов аргументов этим языком. Значение, возвращаемое этой функцией, игнорируется, об ошибках она должна сигнализировать, вызывая `ereport()`.

Параметр TRUSTED и имена вспомогательных функций игнорируются, если на сервере для указанного языка имеется запись в pg_pltemplate.

33.4.Замечания

Для удаления процедурных языков следует использовать DROP LANGUAGE.

В системном каталоге pg_language записывается информация о языках, установленных в данный момент. Список установленных языков также показывает команда \dL в psql.

Чтобы создавать функции на процедурном языке, пользователь должен иметь право USAGE для этого языка. По умолчанию для доверенных языков право USAGE дается роли PUBLIC (т.е. всем), однако при желании его можно отозвать.

Процедурные языки являются локальными по отношению к базам данных. Тем не менее, язык можно установить в базу данных template1, в результате чего он будет автоматически доступен во всех базах данных, создаваемых из нее впоследствии.

Обработчик вызова, обработчик встроенного кода (при наличии) и функция проверки (при наличии) должны уже существовать, если на сервере нет записи для этого языка в pg_pltemplate. Но если такая запись уже есть, функции могут не существовать – в случае отсутствия в базе данных они будут определены автоматически. Это может привести к сбою в CREATE LANGUAGE, если в установленной системе не найдется разделяемая библиотека, реализующая этот язык.

33.5.Примеры

Предпочитаемый способ создания любых процедурных языков довольно прост:

```
CREATE LANGUAGE plperl;
```

Для языка, не представленного в каталоге pg_pltemplate, требуется выполнить следующие действия:

```
CREATE FUNCTION plsample_call_handler() RETURNS  
language_handler
```

```

AS '$libdir/plsample'
LANGUAGE C;
CREATE LANGUAGE plsample
HANDLER plsample_call_handler;

```

33.6.Совместимость

CREATE LANGUAGE является расширением Синергия-БД.

34. CREATE MATERIALIZED VIEW

CREATE MATERIALIZED VIEW – создать материализованное представление.

34.1.Синтаксис

```

CREATE MATERIALIZED VIEW table_name
[ (имя_колонки [, ...] ) ]
[ WITH ( параметр_хранения [= значение] [, ... ] ) ]
[ TABLESPACE табл_пространство ]
AS запрос
[ WITH [ NO ] DATA ]

```

34.2.Описание

CREATE MATERIALIZED VIEW определяет материализованное представление запроса. Заданный запрос выполняется и наполняет представление в момент вызова команды (если только не указано WITH NO DATA). Обновить представление позже можно, выполнив REFRESH MATERIALIZED VIEW.

Команда CREATE MATERIALIZED VIEW подобна CREATE TABLE AS, за исключением того, что она запоминает запрос, порождающий представление, так что это представление можно позже обновить по требованию. Материализованные представления сходны с таблицами во многом, но не во всем; например, не поддерживаются временные материализованные представления и автоматическая генерация OID.

34.3.Параметры

table_name

Имя (возможно, дополненное схемой) создаваемого материализованного представления.

имя_колонки

Имя колонки в создаваемом материализованном представлении. Если имена колонок не заданы явно, они определяются по именам колонок результата запроса.

WITH (параметр_хранения [= значение] [, ...])

Это предложение задает дополнительные параметры хранения для создаваемого материализованного представления. Все параметры, которые поддерживает CREATE TABLE, поддерживает так же и CREATE MATERIALIZED VIEW, за исключением OIDS.

TABLESPACE табл_пространство

Здесь табл_пространство – имя табличного пространства, в котором будет создано материализованное представление. Если оно не задано, выбирается default_tablespace.

запрос

Команда SELECT, TABLE или VALUES. Эта команда будет выполняться в контексте с ограничениями по безопасности; в частности, будут запрещены вызовы функций, которые сами создают временные таблицы.

WITH [NO] DATA

Это предложение указывает, будет ли материализованное представление наполняться в момент создания. Если материализованное представление не наполняется, оно помечается как нечитаемое, так что к нему нельзя будет обращаться до выполнения REFRESH MATERIALIZED VIEW.

34.4.Совместимость

CREATE MATERIALIZED VIEW является расширением Синергия-БД.

35. CREATE OPERATOR

CREATE OPERATOR – создать оператор.

35.1. Синтаксис

```
CREATE OPERATOR имя (  
    PROCEDURE = имя_функции  
    [, LEFTARG = тип_слева ] [, RIGHTARG = тип_справа ]  
    [, COMMUTATOR = коммут_оператор ] [, NEGATOR =  
    обратный_оператор ]  
    [, RESTRICT = процедура_ограничения ] [, JOIN =  
    процедура_соединения ]  
    [, HASHES ] [, MERGES ]  
)
```

35.2. Описание

CREATE OPERATOR определяет новый оператор, имя. Владелец оператора становится пользователь, его создавший. Если указано имя схемы, оператор создается в ней, в противном случае – в текущей схеме.

Имя оператора образует последовательность из нескольких символов (не более чем NAMEDATALEN-1, по умолчанию 63) из следующего списка:

+ - * / < > = ~ ! @ # % ^ & | ` ?

Однако выбор имени ограничен ещё следующими условиями:

1. Сочетания символов -- и /* не могут присутствовать в имени оператора, так как они будут обозначать начало комментария.
2. Многосимвольное имя оператора не может заканчиваться знаками + или -, если только оно не содержит также один из этих символов:

~ ! @ # % ^ & | ` ?

Например, @- – допустимое имя оператора, а *- – нет. Благодаря этому ограничению, Синергия-БД может разбирать корректные SQL-запросы без пробелов между компонентами.

3. Использование => в качестве имени оператора считается устаревшим и может быть вовсе запрещено в будущих выпусках.

Оператор != отображается в <> при вводе, так что эти два имени всегда равнозначны.

Необходимо определить либо LEFTARG, либо RIGHTARG, а для бинарных

операторов оба аргумента. Для правых унарных операторов должен быть определен только LEFTARG, а для левых унарных – только RIGHTARG.

Процедура имя_функции должна быть уже определена с помощью CREATE FUNCTION и иметь соответствующее число аргументов (один или два) указанных типов.

Другие предложения определяют дополнительные характеристики оптимизации.

Чтобы создать оператор, необходимо иметь право USAGE для типов аргументов и результата, а также право EXECUTE для нижележащей функции. Если указывается коммутативный или обратный оператор, нужно быть его владельцем.

35.3.Параметры

имя

Имя определяемого оператора. Допустимые в нем символы перечислены ниже. Указанное имя может быть дополнено схемой, например так: CREATE OPERATOR myschema.+ (...). Если схема не указана, оператор создается в текущей схеме. При этом два оператора в одной схеме могут иметь одно имя, если они работают с разными типами данных. Такое определение операторов называется перегрузкой.

имя_функции

Функция, реализующая этот оператор.

тип_слева

Тип данных левого операнда оператора, если он есть. Этот параметр опускается для левых унарных операторов.

тип_справа

Тип данных правого операнда оператора, если он есть. Этот параметр опускается для правых унарных операторов.

коммут_оператор

Оператор, коммутирующий для данного.

обратный_оператор

Оператор, обратный для данного.

процедура_ограничения

Функция оценки избирательности ограничения для данного оператора.

процедура_соединения

Функция оценки избирательности соединения для этого оператора.

HASHES

Показывает, что этот оператор поддерживает соединение по хешу.

MERGES

Показывает, что этот оператор поддерживает соединение слиянием.

Чтобы задать имя оператора с указанием схемы в коммут_оператор или другом дополнительном аргументе, применяется синтаксис OPERATOR(), например:

```
COMMUTATOR = OPERATOR(mySchema.===) ,
```

35.4.Замечания

Задать лексический приоритет оператора в команде CREATE OPERATOR невозможно, так как обработка приоритетов жестко зашита в анализаторе.

Устаревшие параметры SORT1, SORT2, LTCMP и GTCMP ранее использовались для определения имен операторов сортировки, связанных с оператором, применяемым при соединении слиянием. Теперь это не требуется, так как информацию о связанных операторах теперь дают семейства операторов В-дерева. Если в команде отсутствует явное указание MERGES, все эти параметры игнорируются.

Для удаления пользовательских операторов из базы данных применяется DROP OPERATOR, а для изменения их свойств – ALTER OPERATOR.

35.5.Примеры

Следующая команда определяет новый оператор, проверяющий равенство площадей, для типа box:

```
CREATE OPERATOR === (  
    LEFTARG = box,  
    RIGHTARG = box,  
    PROCEDURE = area_equal_procedure,
```

```

COMMUTATOR = ==,
NEGATOR = !=,
RESTRICT = area_restriction_procedure,
JOIN = area_join_procedure,
HASHES, MERGES
);

```

35.6. Совместимость

CREATE OPERATOR является языковым расширением Синергия-БД. Средства определения пользовательских операторов в стандарте SQL не описаны.

36. CREATE OPERATOR CLASS

CREATE OPERATOR CLASS – создать класс операторов.

36.1. Синтаксис

```

CREATE OPERATOR CLASS имя [ DEFAULT ] FOR TYPE
тип_данных
    USING метод_индекса [ FAMILY имя_семейства ] AS
    {
        OPERATOR номер_стратегии имя_оператора
        [ ( тип_операнда, тип_операнда ) ] [ FOR SEARCH | FOR
        ORDER BY семейство_сортировки ]
        | FUNCTION номер_опорной_функции [ ( тип_операнда [ ,
        тип_операнда ] ) ] имя_функции ( тип_аргумента [, ...] )
        | STORAGE тип_хранения
    } [, ... ]

```

36.2. Описание

CREATE OPERATOR CLASS создает класс операторов. Класс операторов устанавливает, как данный тип будет использоваться в индексе, определяя, какие операторы исполняют конкретные роли или «стратегии» для этого типа данных и метода индекса. Также класс операторов определяет вспомогательные процедуры, которые будет задействовать метод индекса в случае выбора данного класса для колонки индекса. Все операторы и функции, используемые классом операторов, должны существовать до создания этого класса.

Если указывается имя схемы, класс операторов создается в указанной схеме, в противном случае – в текущей. Два класса операторов в одной схеме могут иметь одинаковые имена, только если они предназначены для разных методов индекса.

Владельцем класса операторов становится пользователь, создавший его. В настоящее время создавать классы операторов могут только суперпользователи. Это ограничение введено потому, что ошибочное определение класса может вызвать нарушения или даже сбой в работе сервера.

CREATE OPERATOR CLASS в настоящее время не проверяет, включает ли определение класса операторов все операторы и функции, требуемые для метода индекса, и образуют ли они целостный набор. Ответственность за правильность определения класса операторов лежит на пользователе.

Связанные классы операторов могут быть сгруппированы в семейства операторов. Чтобы поместить класс в существующее семейство, добавьте параметр FAMILY в CREATE OPERATOR CLASS. Без этого параметра новый класс помещается в семейство, имеющее то же имя, что и класс (если такое семейство не существует, оно создается).

36.3.Параметры

имя

Имя создаваемого класса операторов, возможно, дополненное схемой.

DEFAULT

Если присутствует это указание, класс операторов становится классом по умолчанию для своего типа данных. Для определенного типа данных и метода индекса можно определить не больше одного класса операторов по умолчанию.

тип_данных

Тип данных колонки, для которого предназначен этот класс операторов.

метод_индекса

Имя индексного метода, для которого предназначен этот класс операторов.

имя_семейства

Имя существующего семейства операторов, в которое будет добавлен этот класс. Если не указано, подразумевается семейство с тем же именем, что и класс

(если такое семейство не существует, оно создается).

номер_стратегии

Номер стратегии индексного метода для оператора, связанного с данным классом операторов.

имя_оператора

Имя (возможно, дополненное схемой) оператора, связанного с данным классом операторов.

тип_операнда

В предложении OPERATOR это тип данных операнда, либо ключевое слово NONE, характеризующее левый унарный или правый унарный оператор. Типы операндов обычно можно опустить, когда они совпадают с типом данных класса операторов.

В предложении FUNCTION это тип данных операнда, который должна поддерживать эта функция, если он отличается от входного типа данных функции (для функций сравнения в B-деревьях и хеш-функций) или типа данных класса (для функций поддержки сортировки в B-деревьях и всех функций в классах операторов GiST, SP-GiST и GIN). Обычно предполагаемые по умолчанию типы оказываются верными, так что тип_операнда указывать в FUNCTION не нужно, кроме функций сортировки B-дерева, предназначенных для сравнений разных типов данных.

семейство_сортировки

Имя (возможно, дополненное схемой) существующего семейства операторов btree, описывающего порядок сортировки, связанный с оператором сортировки.

Если не указано ни FOR SEARCH (для поиска), ни FOR ORDER BY (для сортировки), подразумевается FOR SEARCH .

номер_опорной_функции

Номер опорной процедуры индексного метода для функции, связанной с данным классом операторов.

имя_функции

Имя (возможно, дополненное схемой) функции, которая является опорной процедурой индексного метода для данного класса операторов.

тип_аргумента

Тип данных параметра функции.

тип_хранения

Тип данных, фактически сохраняемых в индексе. Обычно это тип данных колонки, но некоторые методы индекса (в настоящее время, GiST и GIN) могут работать с отличным от него типом. Предложение STORAGE может присутствовать, только если метод индекса позволяет использовать другой тип данных.

Предложения OPERATOR, FUNCTION и STORAGE могут указываться в любом порядке.

36.4.Замечания

Так как механизмы индексов не проверяют права доступа к функциям, прежде чем вызывать их, включение функций или операторов в класс операторов по сути дает всем право на выполнение их. Обычно это не проблема для таких функций, какие бывают полезны в классе операторов.

Операторы не должны реализовываться в функциях на языке SQL. SQL-функция вероятнее всего будет встроена в вызывающий запрос, что помешает оптимизатору понять, что этот запрос соответствует индексу.

36.5.Примеры

Команда в следующем примере определяет класс операторов индекса GiST для типа данных `_int4` (массива из `int4`).

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR          3          &&,
        OPERATOR          6          = (anyarray, anyarray),
        OPERATOR          7          @>,
        OPERATOR          8          <@,
        OPERATOR          20         @@ (_int4, query_int),
        FUNCTION           1          g_int_consistent
(internal, _int4, int, oid, internal),
        FUNCTION           2          g_int_union (internal,
```

```
internal),
        FUNCTION          3          g_int_compress
(internal),
        FUNCTION          4          g_int_decompress
(internal),
        FUNCTION          5          g_int_penalty (internal,
internal, internal),
        FUNCTION          6          g_int_picksplit
(internal, internal),
        FUNCTION          7          g_int_same (_int4,
_int4, internal);
```

36.6.Совместимость

CREATE OPERATOR CLASS является расширением Синергия-БД. Команда CREATE OPERATOR CLASS отсутствует в стандарте SQL.

37. CREATE OPERATOR FAMILY

CREATE OPERATOR FAMILY – создать семейство операторов

37.1.Синтаксис

CREATE OPERATOR FAMILY имя USING метод_индекса

37.2.Описание

CREATE OPERATOR FAMILY создает новое семейство операторов. Семейство операторов определяет коллекцию связанных классов операторов и, возможно, некоторые дополнительные операторы и вспомогательные функции, совместимые с этими классами, но не требующиеся для функционирования каких-либо индексов. Операторы и функции, требующиеся для работы индексов, следует группировать в соответствующем классе операторов, а не «слабо» связывать в семействе операторов. Обычно операторы с операндами одного типа привязываются к классам операторов, тогда как операторы для двух разных типов могут быть слабо связаны в семействе, содержащем классы операторов для обоих типов.

Создаваемое семейство операторов изначально не содержит ничего. Оно должно наполняться последующими командами CREATE OPERATOR CLASS,

добавляющими в него классы операторов, и, возможно, командами ALTER OPERATOR FAMILY, добавляющими «слабосвязанные» операторы и соответствующие вспомогательные функции.

Если указывается имя схемы, семейство операторов создается в указанной схеме, в противном случае – в текущей. Два семейства операторов в одной схеме могут иметь одинаковые имена, только если они предназначены для разных методов индекса.

Владельцем семейства операторов становится пользователь, создавший его. В настоящее время создавать семейства операторов могут только суперпользователи. Это ограничение введено потому, что ошибочное определение семейства может вызвать нарушения или даже сбой в работе сервера.

37.3.Параметры

имя

Имя создаваемого семейства операторов, возможно, дополненное схемой.

метод_индекса

Имя индексного метода, для которого предназначено это семейство операторов.

37.4.Совместимость

CREATE OPERATOR FAMILY является расширением Синергия-БД. Команда CREATE OPERATOR FAMILY отсутствует в стандарте SQL.

38. CREATE POLICY

CREATE POLICY – создать политику для таблицы.

38.1.Синтаксис

```
CREATE POLICY имя ON table_name
    [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { имя_роли | PUBLIC | CURRENT_USER |
SESSION_USER } [, ...] ]
    [ USING ( выражение_использования ) ]
```

[WITH CHECK (выражение_проверки)]

38.2.Описание

Команда `CREATE POLICY` определяет новую политику для таблицы. Заметьте, что защита на уровне строк также должна быть включена для таблицы командой `ALTER TABLE`, чтобы созданные политики действовали.

Политика дает разрешение на выборку, добавление, изменение или удаление строк, удовлетворяющих соответствующему выражению политики. Существующие строки таблицы проверяются по выражению, указанному в `USING`, тогда как строки, которые могут быть созданы командами `INSERT` или `UPDATE` проверяются по выражению, указанному в `WITH CHECK`. Когда выражение `USING` истинно для заданной строки, эта строка видна пользователю, а если ложно или выдает `NULL`, строка не видна. Когда выражение `WITH CHECK` истинно для заданной строки, эта строка добавляется, а если ложно или выдает `NULL`, происходит ошибка.

Вообще система будет применять фильтры, устанавливаемые политиками безопасности, до условий, заданных в самом запросе, чтобы предотвратить незапланированную утечку защищаемых данных через пользовательские функции, которые могут быть недоверенными. Однако функции и операторы, помеченные системой (или системным администратором) как `LEAKPROOF` (герметичные) могут вычисляться до условий политики, так как они считаются доверенными.

Для операторов `INSERT` и `UPDATE` выражения `WITH CHECK` применяются после срабатывания триггеров `BEFORE`, но до того, как данные будут модифицированы. Таким образом, триггер `BEFORE ROW` может изменить данные, подлежащие добавлению, и повлиять на результат условия политики безопасности. Выражения `WITH CHECK` обрабатываются до каких-либо других ограничений.

Имена политик задаются на уровне таблицы. Таким образом, одно имя политики можно использовать в нескольких разных таблицах и в каждой дать отдельное, подходящее этой таблице определение политики.

Политики можно применять к определенным командам или избранным ролям. По умолчанию создаваемые политики применяются ко всем командам и ролям, если

нет других указаний. Если к заданному оператору применяются несколько политик, они будут объединены логическим сложением (хотя политики ON CONFLICT DO UPDATE и INSERT таким образом не складываются, а применяются как отмечено на каждой стадии выполнения ON CONFLICT).

Кроме того, для команд, допускающих наличие политик USING и WITH CHECK (ALL и UPDATE), в отсутствии политики WITH CHECK политика USING будет определять и видимость строк (обычное предназначение USING), и разрешение на добавление строк (предназначение WITH CHECK).

38.3.Параметры

имя

Имя создаваемой политики. Оно должно отличаться от имен других политик для этой таблицы.

table_name

Имя (возможно, дополненное схемой) существующей таблицы (или представления), к которой применяется эта политика.

command

Команда, к которой применяется политика. Допустимые варианты: ALL, SELECT, INSERT, UPDATE и DELETE. ALL (все) подразумевается по умолчанию. Особенности их применения описаны ниже.

имя_роли

Роли, к которым применяется политика. По умолчанию подразумевается PUBLIC, то есть политика применяется ко всем ролям.

выражение_использования

Любое условное выражение SQL (возвращающее boolean). Это выражение не должно содержать агрегатные или оконные функции. Заданное выражение будет добавляться к запросам, обращающимся к таблице, если для нее включена защита на уровне строк, и строки, для которых это выражение истинно, будут видимыми. Строки, для которых это выражение ложно или выдает NULL, не будут видны пользователю.

выражение_проверки

Любое условное выражение SQL (возвращающее `boolean`). Это выражение не должно содержать агрегатные или оконные функции. Заданное выражение будет применяться с запросами `INSERT` и `UPDATE` к целевой таблице, если включена защита строк, и операция будет разрешена только для тех строк, для которых это выражение истинно. Если выражение ложно или выдает `NULL` для записей, вставляемых или полученных в результате изменения, происходит ошибка.

38.4. Политики по командам

`ALL`

Указание `ALL` для политики означает, что она применяется ко всем командам, вне зависимости от типа. Если существует политика `ALL` и другие более детализированные политики, тогда и политика `ALL`, и более детализированная политика (или политики) объединяются логическим сложением, как обычно при наложении политик. Кроме того, политики `ALL` с выражением `USING` будут применяться и к стороне выборки, и к стороне изменения данных, если определена только политика `USING`.

Например, если выполняется `UPDATE`, политика `ALL` будет применяться и для проверки строк, которые оператор `UPDATE` сможет выбрать для изменения (по выражению `USING`), и для проверки строк, выданных оператором `UPDATE`, с целью определить, какие из них разрешено добавить в эту таблицу (по выражению `WITH CHECK`, если оно определено, или по выражению `USING` в противном случае). Если команда `INSERT` или `UPDATE` попытается добавить строки, не удовлетворяющие условию `ALL WITH CHECK`, вся команда будет прервана. Заметьте, что если указано только предложение `USING`, это выражение применяется и в качестве `USING`, и в качестве `WITH CHECK`.

`SELECT`

Указание `SELECT` для политики означает, что она будет применяться к командам `SELECT`. С такой политикой будут возвращены только те записи из отношения, которые пройдут проверку политики `SELECT`, даже если в этом

отношении есть и другие записи. Политика `SELECT` принимает только выражение `USING`, так как она действует только когда из отношения получаются записи.

`INSERT`

Указание `INSERT` для политики означает, что она применяется к командам `INSERT`. Если вставляемые строки не проходят проверку политики, происходит ошибка нарушения политики и вся команда `INSERT` прерывается. Политика `INSERT` принимает только выражение `WITH CHECK`, так как она действует только когда в отношение добавляются записи.

Заметьте, что `INSERT` с указанием `ON CONFLICT DO UPDATE` требует, чтобы для всех добавляемых в отношение строк выполнялись выражения `WITH CHECK` всех политик `INSERT`, только по пути `INSERT`.

`UPDATE`

Указание `UPDATE` для политики означает, что она будет применяться для команд `UPDATE` (или дополнительных предложений `ON CONFLICT DO UPDATE` команд `INSERT`). Так как `UPDATE` подразумевает извлечение существующей записи и внесение изменений в некоторые её поля (возможно не все), политика `UPDATE` принимает и выражение `USING`, и выражение `WITH CHECK`. Выражение `USING` будет применяться, чтобы определить, какие записи увидит и будет обрабатывать команда `UPDATE`, а выражение `WITH CHECK` определяет, какие строки могут быть внесены в отношение (как в политике `INSERT`). Если для каких-то строк результирующие значения не проходят проверку `WITH CHECK`, произойдет ошибка и вся команда будет прервана. Заметьте, что если указано только предложение `USING`, это предложение будет применяться и в качестве `USING`, и в качестве `WITH CHECK`.

Заметьте однако, что `INSERT` с `ON CONFLICT DO UPDATE` требует, чтобы выражение `USING` политики `UPDATE` всегда действовало как выражение `WITH CHECK`. Эта политика `UPDATE` должна всегда выполняться, когда выбирается путь `UPDATE`. Любая существующая строка, из-за которой выбирается путь `UPDATE`, должна удовлетворять проверкам (политик `UPDATE` или `ALL`) `USING`

(объединенным логическим сложением), которые всегда действуют как WITH CHECK в данном контексте. Путь UPDATE никогда не пропускается, в случае невыполнения условия выдается ошибка. Наконец, окончательная строка, добавляемая в отношение, должна проходить все проверки WITH CHECK, которые должны быть пройдены и при обычном UPDATE.

DELETE

Указание DELETE для политики означает, что она будет применяться только к командам DELETE. Команда DELETE увидит только те строки, которые удовлетворяют условию политики. При этом могут существовать строки, видимые через SELECT, но невидимые для DELETE, если они не удовлетворяют выражению USING для DELETE. И наоборот, могут существовать строки, которые не видны при применении политики SELECT, но могут быть удалены, если они удовлетворяют политике DELETE USING. Политика DELETE принимает только выражение USING, так как она действует только когда из отношения получают записи для удаления.

38.5.Замечания

Чтобы создать или изменить политики для таблицы, нужно быть её владельцем.

Учтите, что политики применяются к явно выполняемым запросам к таблицам СУБД, но не применяются, когда система выполняет внутренние проверки ссылочной целостности или проверяет ограничения. Это означает, что существуют косвенные пути проверить существование заданного значения. Например, можно попытаться вставить повторяющееся значение в колонку, образующую первичный ключ или имеющую ограничение уникальности. Если при этом произойдет ошибка, пользователь может заключить, что это значение уже существует. В данном случае предполагается, что политика разрешает пользователю вставлять записи, которые он может не видеть. Подобный прием также возможен, если пользователь может вставлять записи в таблицу, которая ссылается на другую, иным образом не видимую. Существование значения можно определить, вставив его в подчиненную таблицу, при этом успешный результат операции будет признаком того, что это

значение есть в главной таблице. Эти изъяны можно устранить, либо тщательно разработав политики, которые вовсе не позволят пользователям выполнять операции добавления, изменения и удаления, по результатам которых можно узнать о значениях в таблицах, не видимых иным образом, либо используя генерируемые значения (например, суррогатные ключи).

И ещё одно замечание относительно взаимоотношений пользователя и выражений политик: так как эти выражения добавляются непосредственно в запрос пользователя, они выполняются с правами пользователя, запускающего исходный запрос. Таким образом, пользователи, на которых распространяется заданная политика, должны иметь права для обращения ко всем таблицам и функциям, задействованным в выражении, иначе им просто будет отказано в доступе при попытке обращения к целевой таблице (если для нее включена защита на уровне строк). Однако это не влияет на работу представлений – как и с обычными запросами и представлениями, проверки разрешений и политики для нижележащих таблиц представления будут выполняться с правами владельца представления, при этом будут действовать политики, распространяющиеся на этого владельца.

39. CREATE ROLE

CREATE ROLE – создать роль в базе данных.

39.1. Синтаксис

CREATE ROLE имя [[WITH] параметр [...]]

Здесь параметр:

```
SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| CREATEUSER | NOCREATEUSER
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| BYPASSRLS | NOBYPASSRLS
| CONNECTION LIMIT предел_подключений
```

```
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'  
| VALID UNTIL 'дата_время'  
| IN ROLE имя_роли [, ...]  
| IN GROUP имя_роли [, ...]  
| ROLE имя_роли [, ...]  
| ADMIN имя_роли [, ...]  
| USER имя_роли [, ...]  
| SYSID uid
```

39.2.Описание

CREATE ROLE добавляет новую роль в кластер баз данных Синергия-БД. Роль – это сущность, которая может владеть объектами и иметь определенные права в базе; роль может представлять «пользователя», «группу» или и то, и другое, в зависимости от варианта использования. Чтобы выполнить эту команду, необходимо быть суперпользователем или иметь право CREATEROLE.

Учтите, что роли определяются на уровне кластера баз данных, так что они действуют во всех базах в кластере.

39.3.Параметры

ИМЯ

Имя создаваемой роли.

SUPERUSER

NOSUPERUSER

Эти предложения определяют, будет ли эта роль «суперпользователем», который может переопределить все ограничения доступа в базе данных. Статус суперпользователя несет опасность и назначать его следует только в случае необходимости. Создать нового суперпользователя может только суперпользователь. В отсутствие этих предложений по умолчанию подразумевается NOSUPERUSER.

CREATEDB

NOCREATEDB

Эти предложения определяют, сможет ли роль создавать базы данных. Указание CREATEDB дает новой роли это право, а NOCREATEDB запрещает роли

создавать базы данных. По умолчанию подразумевается NOCREATEDB.

CREATEROLE
NOCREATEROLE

Эти предложения определяют, сможет ли роль создавать новые роли (т.е. выполнять CREATE ROLE). Роль с правом CREATEROLE может также изменять и удалять другие роли. По умолчанию подразумевается NOCREATEROLE.

CREATEUSER
NOCREATEUSER

Эти предложения являются устаревшими, но по прежнему принимаются, как альтернативное написание SUPERUSER и NOSUPERUSER. Учтите, что они не равнозначны паре CREATEROLE, как можно было бы подумать!

INHERIT
NOINHERIT

Эти предложения определяют, будет ли роль «наследовать» права ролей, членом которых она является. Роль с атрибутом INHERIT может автоматически использовать в базе данных любые права, назначенные всем ролям, в которые она включена, непосредственно или опосредованно. Без INHERIT членство в другой роли позволяет только выполнить SET ROLE и переключиться на эту роль; правами, назначенными другой роли, можно будет пользоваться только после этого. По умолчанию подразумевается INHERIT.

LOGIN
NOLOGIN

Эти предложения определяют, разрешается ли новой роли вход на сервер; то есть, может ли эта роль стать начальным авторизованным именем при подключении клиента. Можно считать, что роль с атрибутом LOGIN соответствует пользователю. Роли без этого атрибута бывают полезны для управления доступом в базе данных, но это не пользователи в обычном понимании. По умолчанию подразумевается вариант NOLOGIN, за исключением вызова CREATE ROLE в виде CREATE USER.

REPLICATION
NOREPLICATION

Эти предложения определяют, сможет ли новая роль запускать потоковую репликацию или переключать систему в режим резервного копирования и из него.

Роль с атрибутом `REPLICATION` получает очень широкие полномочия, поэтому его следует назначать, только если роль фактически применяется для репликации. По умолчанию подразумевается `NOREPLICATION`.

`BYPASSRLS`
`NOBYPASSRLS`

Эти предложения определяют, будет ли роль обходить политики защиты строк. По умолчанию подразумевается `NOBYPASSRLS`. Заметьте, что команда `pg_dump` по умолчанию устанавливает параметр `row_security` в значение `OFF`, чтобы выгрузить все содержимое таблицы. Если у пользователя, запустившего `pg_dump`, недостаточно привилегий, будет возвращена ошибка. Суперпользователь и владелец выгружаемой таблицы всегда обходят политики защиты строк.

`CONNECTION LIMIT` предел_подключений

Если роли разрешен вход, этот параметр определяет максимальное число одновременных подключений, разрешенных для этой роли. Значение `-1` (по умолчанию) отключает ограничение.

`PASSWORD` password

Задает пароль роли. Пароль полезен только для ролей с атрибутом `LOGIN`, но задать его можно и для ролей без такого атрибута. Если проверка подлинности по паролю не будет использоваться, этот параметр можно опустить. При указании пустого значения будет задан пароль `NULL`, что не позволит данному пользователю пройти проверку подлинности по паролю. При желании пароль `NULL` можно установить явно, указав `PASSWORD NULL`.

`ENCRYPTED`
`UNENCRYPTED`

Эти ключевые слова определяют, будет ли пароль храниться в системных каталогах в зашифрованном виде. При отсутствии явного указания поведение по умолчанию определяется конфигурационным параметром `password_encryption`. Если пароль представлен в виде MD5-хеша, он сохраняется как есть, вне зависимости от того, присутствует ли указание `ENCRYPTED` или `UNENCRYPTED` (так как система не может расшифровать зашифрованный пароль). Это позволяет выгружать/загружать зашифрованные пароли при экспорте/импорте данных.

Учтите, что старые клиенты могут не поддерживать механизм проверки подлинности с MD5, необходимый для использования паролей, хранящихся в зашифрованном виде.

```
VALID UNTIL 'дата_время'
```

Предложение `VALID UNTIL` устанавливает дату и время, после которого пароль роли перестает действовать. Если это предложение отсутствует, срок действия пароля будет неограниченным.

```
IN ROLE имя_роли
```

В предложении `IN ROLE` перечисляются одна или несколько существующих ролей, в которые будет немедленно включена новая роль. Заметьте, что добавить новую роль с правами администратора таким образом нельзя; для этого надо отдельно выполнить команду `GRANT`.

```
IN GROUP имя_роли
```

`IN GROUP` – устаревшее написание предложения `IN ROLE`.

```
ROLE имя_роли
```

В предложении `ROLE` перечисляются одна или несколько существующих ролей, которые автоматически становятся членами создаваемой роли. По сути таким образом новая роль становится «группой».

```
ADMIN имя_роли
```

Предложение `ADMIN` подобно `ROLE`, но перечисленные в нем роли включаются в новую роль с атрибутом `WITH ADMIN OPTION`, что дает им право включать в эту роль другие роли.

```
USER имя_роли
```

Предложение `USER` является устаревшим написанием предложения `ROLE`.

```
SYSID uid
```

Предложение `SYSID` игнорируется, но принимается для обратной совместимости.

39.4.Замечания

Для изменения атрибутов роли применяется `ALTER PROCEDURE`.

39.5.Примеры

Создание роли, для которой разрешен вход, но не задан пароль:

```
CREATE ROLE jonathan LOGIN;
```

Создание роли с паролем:

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

CREATE USER действует так же, как CREATE ROLE, но подразумевает ещё и атрибут LOGIN.

Создание роли с паролем, действующим до конца 2004 г., то есть пароль перестает действовать в первую же секунду 2005 г.:

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID  
UNTIL '2005-01-01';
```

Создание роли, которая может создавать базы данных и управлять ролями:

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

39.6.Совместимость

Оператор CREATE ROLE описан в стандарте SQL, но стандарт требует поддержки только следующего синтаксиса:

```
CREATE ROLE имя [ WITH ADMIN имя_роли ]
```

Возможность создавать множество начальных администраторов и все другие параметры CREATE ROLE относятся к расширениям Синергия-БД.

В стандарте SQL определяются концепции пользователей и ролей, но в нем они рассматриваются как отдельные сущности, а все команды создания пользователей считаются внутренней спецификой СУБД. В Синергия-БД мы решили объединить пользователей и роли в единую сущность, так что роли получили дополнительные атрибуты, не описанные в стандарте.

Поведение, наиболее близкое к описанному в стандарте SQL, можно получить, если создавать пользователей с атрибутом NOINHERIT, а роли – с атрибутом INHERIT.

40. CREATE RULE

CREATE RULE – создать правило перезаписи.

40.1. Синтаксис

```
CREATE [ OR REPLACE ] RULE имя AS ON событие  
    TO table_name [ WHERE условие ]  
    DO [ ALSO | INSTEAD ] { NOTHING | command |  
    ( command ; command ... ) }
```

Здесь допускается событие:

```
SELECT | INSERT | UPDATE | DELETE
```

40.2. Описание

CREATE RULE создает правило, применяемое к указанной таблице или представлению. CREATE OR REPLACE RULE либо создает новое правило, либо заменяет существующее с тем же именем для той же таблицы.

Система правил Синергия-БД позволяет определить альтернативное действие, заменяющее операции добавления, изменения или удаления данных в таблицах базы данных. Грубо говоря, правило описывает дополнительные команды, которые будут выполняться при вызове определенной команды для определенной таблицы. Кроме того, правило INSTEAD может заменить заданную команду другой, либо сделать, чтобы она не выполнялась вовсе. Правила также применяются для реализации SQL-запросов. Важно понимать, что правило это фактически механизм преобразования команд (макрос). Заданное преобразование имеет место до начала выполнения команды. Когда требуется выполнить некоторую операцию независимо для каждой физической строки, скорее всего, для этого нужно применять триггер, а не правило.

В настоящее время правила ON SELECT должны быть безусловными, с характеристикой INSTEAD (вместо исходного), и их действия должны состоять из единственной команды SELECT. Таким образом, правило ON SELECT по сути превращает таблицу в представление, чьим видимым содержимым являются строки, возвращаемые командой SELECT, заданной в правиле, а не данные, хранящиеся в таблице (если они есть). Вообще же для этой цели лучшим стилем считается пользоваться командой CREATE VIEW, а не создавать реальную таблицу и определять затем правило ON SELECT для нее.

С помощью правил можно создать иллюзию изменяемого представления,

определив правила ON INSERT, ON UPDATE и ON DELETE (либо только те, которых достаточно для решения поставленной задачи) и заменив операции изменения данных в представлении соответствующими действиями с другими таблицами. Если требуется поддерживать оператор INSERT RETURNING и подобные ему, в каждое из этих правил обязательно нужно поместить подходящее предложение RETURNING.

Использование правил с условиями для сложных изменений представлений связано с одним ограничением: для каждого действия, которое вы хотите разрешить для представления, необходимо определить безусловное правило INSTEAD. Если определено только условное правило, или правило не типа INSTEAD, система отвергнет попытки выполнить изменения, предполагая, что в некоторых случаях изменения могут свестись к операциям с фиктивной нижележащей таблицей. При желании обработать все полезные случаи изменений в условных правилах, добавьте безусловное правило DO INSTEAD NOTHING, чтобы система понимала, что ей никогда не придется изменять нижележащую таблицу. Затем создайте условные правила без свойства INSTEAD; в тех случаях, когда они будут применяться, их действия будут добавлены к действию по умолчанию INSTEAD NOTHING. (Однако, этот способ в настоящее время не подходит для реализации запросов RETURNING.)

Примечание. Представления, достаточно простые для реализации автоматического обновления, могут быть изменяемыми без пользовательских правил. Хотя вы, тем не менее, можете создать явное правило, обычно автоматическое преобразование будет работать лучше такого правила.

Другая, заслуживающая рассмотрения, альтернатива правилам – триггеры INSTEAD OF.

40.3.Параметры

ИМЯ

Имя создаваемого правила. Оно должно отличаться от имен любых других правил для той же таблицы. При наличии нескольких правил для одной таблицы и одного типа события они применяются в алфавитном порядке.

СОБЫТИЕ

Одно из следующих событий: SELECT, INSERT, UPDATE или DELETE.

table_name

Имя (возможно, дополненное схемой) существующей таблицы (или представления), к которой применяется это правило.

условие

Любое выражение условия SQL (возвращающее boolean). Это выражение не может ссылаться на какие-либо таблицы, кроме как на NEW и OLD, и не может содержать агрегатные функции.

INSTEAD

INSTEAD указывает, что заданные команды должны выполняться вместо исходной команды.

ALSO

ALSO указывает, что заданные команды должны выполняться в дополнение к исходной команде.

Если ни ALSO, ни INSTEAD не указано, по умолчанию подразумевается ALSO.

command

Команда или команды, составляющие действие правила. Здесь допустимы команды: SELECT, INSERT, UPDATE, DELETE и NOTIFY.

В параметрах условие и команда можно использовать имена специальных таблиц NEW и OLD для обращения к значениям в соответствующей таблице. NEW (новая) принимается в правилах ON INSERT и ON UPDATE и обозначает ссылку на новую строку, добавляемую или изменяемую. OLD (старая) принимается в правилах ON UPDATE и ON DELETE и обозначает ссылку на существующую строку, изменяемую или удаляемую.

40.4.Замечания

Чтобы создать или изменить правила для таблицы, нужно быть её владельцем.

В правило для INSERT, UPDATE или DELETE для представления можно добавить предложение RETURNING, выдающее колонки представления. Это предложение будет генерировать результат, если правило сработает при выполнении

команды INSERT RETURNING, UPDATE RETURNING или DELETE RETURNING. Когда правило срабатывает при выполнении команды без RETURNING, предложение RETURNING этого правила игнорируется. В текущей реализации только безусловные правила INSTEAD могут содержать RETURNING; более того, допускается максимум одно предложение RETURNING среди всех правил для некоторого события. (Благодаря этому ограничению, только одно предложение RETURNING может быть выбрано для вычисления результатов.) Запросы с RETURNING к данному представлению не будут выполняться, если ни одно из определенных для него правил не содержит предложение RETURNING.

Очень важно следить за тем, чтобы правила не за цикливались. Например, два следующих определения правил будут приняты Синергия-БД, но при попытке выполнить команду SELECT Синергия-БД сообщит об ошибке из-за рекурсивного расширения правила:

```
CREATE RULE «_RETURN» AS
  ON SELECT TO t1
  DO INSTEAD
    SELECT * FROM t2;
```

```
CREATE RULE «_RETURN» AS
  ON SELECT TO t2
  DO INSTEAD
    SELECT * FROM t1;
```

```
SELECT * FROM t1;
```

В настоящее время, если действие правила содержит команду NOTIFY, эта команда будет выполняться безусловно, то есть, NOTIFY будет выдаваться, даже если не найдется никаких строк, к которым бы применялось правило. Например, в следующем примере:

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO ALSO
  NOTIFY mytable;
```

```
UPDATE mytable SET name = 'foo' WHERE id = 42;
```

одно событие NOTIFY будет отправлено при выполнении команды UPDATE, даже если никакие строки не соответствуют условию `id = 42`. Это недостаток текущей реализации, который может быть исправлен в будущих версиях.

40.5. Совместимость

Оператор `CREATE RULE` является языковым расширением Синергия-БД, как и вся система перезаписи запросов.

41. CREATE SCHEMA

`CREATE SCHEMA` – создать схему.

41.1. Синтаксис

```
CREATE SCHEMA имя_схемы [ AUTHORIZATION  
имя_пользователя ] [ элемент_схемы [ ... ] ]
```

```
CREATE SCHEMA AUTHORIZATION имя_пользователя  
[ элемент_схемы [ ... ] ]
```

```
CREATE SCHEMA IF NOT EXISTS имя_схемы [ AUTHORIZATION  
имя_пользователя ]
```

```
CREATE SCHEMA IF NOT EXISTS AUTHORIZATION  
имя_пользователя
```

41.2. Описание

`CREATE SCHEMA` создает новую схему в текущей базе данных. Имя схемы должно отличаться от имен других существующих схем в текущей базе данных.

Схема по сути представляет собой пространство имен: она содержит именованные объекты (таблицы, типы данных, функции и операторы), имена которых могут совпадать с именами других объектов, существующих в других схемах. Для обращения к объекту нужно либо «дополнить» его имя именем схемы в виде префикса, либо установить путь поиска, включающий требуемую схему. Команда `CREATE`, в которой указывается неполное имя объекта, создает объект в текущей схеме (схеме, стоящей первой в пути поиска; узнать её позволяет функция `current_schema`).

Команда `CREATE SCHEMA` может дополнительно включать подкоманды, создающие объекты в новой схеме. Эти подкоманды по сути воспринимаются как отдельные команды, выполняемые после создания схемы, за исключением того, что с предложением `AUTHORIZATION` все создаваемые объекты будут принадлежать указанному в нем пользователю.

41.3.Параметры

`имя_схемы`

Имя создаваемой схемы. Если оно опущено, именем схемы будет `имя_пользователя`. Это имя не может начинаться с `pg_`, так как такие имена зарезервированы для системных схем.

`имя_пользователя`

Имя пользователя (роли), назначаемого владельцем новой схемы. Если опущено, по умолчанию владельцем будет пользователь, выполняющий команды. Чтобы назначить владельцем создаваемой схемы другую роль, необходимо быть непосредственным или опосредованным членом этой роли, либо суперпользователем.

`элемент_схемы`

Оператор SQL, определяющий объект, создаваемый в новой схеме. В настоящее время `CREATE SCHEMA` может содержать только подкоманды `CREATE TABLE`, `CREATE VIEW`, `CREATE INDEX`, `CREATE SEQUENCE`, `CREATE TRIGGER` и `GRANT`. Создать объекты других типов можно отдельными командами после создания схемы.

`IF NOT EXISTS`

Не делать ничего (только выдать замечание), если схема с таким именем уже существует. Когда используется это указание, эта команда не может содержать подкоманды `элемент_схемы`.

41.4.Замечания

Чтобы создать схему, пользователь должен иметь право `CREATE` в текущей базе данных. На суперпользователей это условие не распространяется.

41.5.Примеры

Создание схемы:

```
CREATE SCHEMA myschema;
```

Создание схемы для пользователя joe; схема так же получит имя joe:

```
CREATE SCHEMA AUTHORIZATION joe;
```

Создание схемы с именем test, владельцем которой будет пользователь joe, если только схема test ещё не существует. Является ли владельцем существующей схемы пользователь joe, значения не имеет.

```
CREATE SCHEMA IF NOT EXISTS test AUTHORIZATION joe;
```

Создание схемы, в которой сразу создаются таблица и представление:

```
CREATE SCHEMA hollywood
```

```
    CREATE TABLE films (title text, release date, awards  
text[])
```

```
    CREATE VIEW winners AS
```

```
        SELECT title, release FROM films WHERE awards IS  
NOT NULL;
```

Заметьте, что отдельные подкоманды не завершаются точкой с запятой.

Следующие команды приводят к тому же результату другим способом:

```
CREATE SCHEMA hollywood;
```

```
CREATE TABLE hollywood.films (title text, release date,  
awards text[]);
```

```
CREATE VIEW hollywood.winners AS
```

```
    SELECT title, release FROM hollywood.films WHERE  
awards IS NOT NULL;
```

41.6.Совместимость

Стандарт SQL также допускает в команде CREATE SCHEMA предложение DEFAULT CHARACTER SET и дополнительные типы подкоманд, которые Синергия-БД в настоящее время не принимает.

В стандарте SQL говорится, что подкоманды в CREATE SCHEMA могут следовать в любом порядке. Однако текущая реализация в Синергия-БД не воспринимает все возможные варианты ссылок вперед в подкомандах, поэтому иногда возникает необходимость переупорядочить подкоманды, чтобы исключить

такие ссылки.

Согласно стандарту SQL, владелец схемы всегда владеет всеми объектами в ней, но Синергия-БД допускает размещение в схемах объектов, принадлежащих не владельцу схемы. Такая ситуация возможна, только если владелец схемы даст право CREATE в этой схеме кому-либо другому, либо объекты в ней будет создавать суперпользователь.

Указание IF NOT EXISTS является расширением Синергия-БД.

42. CREATE SEQUENCE

CREATE SEQUENCE – создать генератор последовательности.

42.1.Синтаксис

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ]
ИМЯ
    [ AS тип_данных ]
    [ INCREMENT [ BY ] шаг ]
    [ MINVALUE мин_значение | NO MINVALUE ] [ MAXVALUE
макс_значение | NO MAXVALUE ]
    [ START [ WITH ] начало ] [ CACHE кеш ] [ [ NO ]
CYCLE ]
    [ OWNED BY { имя_таблицы.имя_столбца | NONE } ]
```

42.2.Описание

CREATE SEQUENCE создает генератор последовательности. Эта операция включает создание и инициализацию специальной таблицы ИМЯ, содержащей одну строку. Владелец генератора будет пользователь, выполняющий эту команду.

Если указано имя схемы, последовательность создается в заданной схеме, в противном случае – в текущей. Временные последовательности существуют в специальной схеме, так что при создании таких последовательностей имя схемы задать нельзя. Имя последовательности должно отличаться от имен других последовательностей, таблиц, индексов, представлений или сторонних таблиц, уже существующих в этой схеме.

После создания последовательности работать с ней можно, вызывая функции

`nextval`, `currval` и `setval`.

Хотя непосредственно изменить значение последовательности нельзя, получить её параметры и текущее состояние можно таким запросом:

```
SELECT * FROM name;
```

В частности, поле `last_value` последовательности будет содержать последнее значение, выделенное для какого-либо сеанса. Ко времени вывода это значение может стать неактуальным, если другие сеансы активно вызывают `nextval`.

42.3.Параметры

`TEMPORARY` или `TEMP`

Если указано, объект последовательности создается только для данного сеанса и автоматически удаляется при завершении сеанса. Существующая постоянная последовательность с тем же именем не будут видна (в этом сеансе), пока существует временная, однако к ней можно обратиться, дополнив имя указанием схемы.

`IF NOT EXISTS`

Не считать ошибкой, если отношение с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее отношение как-то соотносится с последовательностью, которая могла бы быть создана – это может быть даже не последовательность.

имя

Имя создаваемой последовательности (возможно, дополненное схемой).

тип_данных

Необязательное предложение `AS тип_данных` задаёт тип данных для последовательности. Допустимые типы: `smallint`, `integer` и `bigint`. По умолчанию устанавливается тип `bigint`. От типа данных зависят принимаемые по умолчанию минимальное и максимальное значения последовательности.

шаг

Необязательное предложение `INCREMENT BY шаг` определяет, какое число будет добавляться к текущему значению последовательности для получения нового

значения. С положительным шагом последовательность будет возрастающей, а с отрицательным – убывающей. Значение по умолчанию: 1.

мин_значение
NO MINVALUE

Необязательное предложение MINVALUE мин_значение определяет наименьшее число, которое будет генерировать последовательность. Если это предложение опущено либо указано NO MINVALUE, используется значение по умолчанию: 1 для возрастающей последовательности или минимальное значение типа данных – для убывающей.

макс_значение
NO MAXVALUE

Необязательное предложение MAXVALUE макс_значение определяет наибольшее число, которое будет генерировать последовательность. Если это предложение опущено либо указано NO MAXVALUE, используется значение по умолчанию: максимальное значение типа данных для возрастающей последовательности или -1 – для убывающей.

начало

Необязательное предложение START WITH начало позволяет запустить последовательность с любого значения. По умолчанию началом считается мин_значение для возрастающих последовательностей и макс_значение для убывающих.

кеш

Необязательное предложение CACHE кеш определяет, сколько чисел последовательности будет выделяться и сохраняться в памяти для ускорения доступа к ним. Минимальное значение равно 1 (за один раз генерируется только одно значение, т.е. кеширования нет), и оно же предполагается по умолчанию.

CYCLE
NO CYCLE

Параметр CYCLE позволяет зациклить последовательность при достижении макс_значения или мин_значения для возрастающей или убывающей последовательности, соответственно. Когда этот предел достигается, следующим

числом этих последовательностей будет соответственно мин_значение или макс_значение.

Если указывается NO CYCLE, при каждом вызове nextval после достижения предельного значения будет возникать ошибка. Если указания CYCLE и NO CYCLE отсутствуют, по умолчанию предполагается NO CYCLE.

OWNED BY имя_таблицы.имя_колонки
OWNED BY NONE

Предложение OWNED BY позволяет связать последовательность с определенной колонкой таблицы так, чтобы при удалении этой колонки (или всей таблицы) последовательность удалялась автоматически. Указанная таблица должна иметь того же владельца и находиться в той же схеме, что и последовательность. Подразумеваемое по умолчанию предложение OWNED BY NONE указывает, что такая связь не устанавливается.

42.4.Замечания

Для удаления последовательности применяется команда DROP SEQUENCE.

Последовательности основаны на арифметике bigint, так что их значения не могут выходить за диапазон восьмибайтовых целых (-9223372036854775808 .. 9223372036854775807).

Так как вызовы nextval и setval никогда не откатываются, объекты последовательностей не подходят, если требуется обеспечить непрерывное назначение номеров последовательностей. Непрерывное назначение можно организовать, используя исключительную блокировку таблицы со счётчиком; однако это решение будет гораздо дороже, чем применение объектов последовательностей, особенно когда последовательные номера будут затребоваться сразу многими транзакциями.

Если значение параметра кеш больше единицы, и объект последовательности используется параллельно в нескольких сеансах, результат может оказаться не вполне ожидаемым. Каждый сеанс будет выделять и кешировать несколько очередных значений последовательности при одном обращении к объекту

последовательности и соответственно увеличивать последнее_значение этого объекта. Затем при следующих кеш-1 вызовах `nextval` в этом сеансе будет просто возвращать заготовленные значения, не касаясь объекта последовательности. В результате, все числа, выделенные, но не использованные в сеансе, будут потеряны при завершении сеанса, что приведет к образованию «дырок» в последовательности.

Более того, хотя разным сеансам гарантированно выделяются различные значения последовательности, если рассмотреть все сеансы в целом, порядок этих значений может быть нарушен. Например, при значении кеш, равном 10, сеанс А может зарезервировать значения 1..10 и получить `nextval=1`, затем сеанс В может зарезервировать значения 11..20 и получить `nextval=11` до того, как в сеансе А сгенерируется `nextval=2`. Таким образом, при значении кеш, равном одному, можно быть уверенными в том, что `nextval` генерирует последовательные значения; но если кеш больше одного, рассчитывать можно только на то, что все значения `nextval` различны; их порядок может быть непоследовательным. Кроме того, `last_value` возвращает последнее зарезервированное значение для всех сеансов, вне зависимости от того, было ли оно уже возвращено функцией `nextval`.

Также следует учитывать, что действие функции `setval`, выполненной для такой последовательности, проявится в других сеансах только после того, как в них будут использованы все предварительно закешированные значения.

42.5.Примеры

Создание возрастающей последовательности с именем `serial`, с начальным значением 101:

```
CREATE SEQUENCE serial START 101;
```

Получение следующего номера этой последовательности:

```
SELECT nextval('serial');
```

```
nextval
-----
101
```

Получение следующего номера этой последовательности:

```
SELECT nextval('serial');
```

```
nextval
-----
      102
```

Использование этой последовательности в команде INSERT:

```
INSERT INTO distributors VALUES (nextval('serial'),
'nothing');
```

Изменение значения последовательности после COPY FROM:

```
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', max(id)) FROM distributors;
END;
```

42.6. Совместимость

Команда CREATE SEQUENCE соответствует стандарту SQL, со следующими исключениями:

- для получения следующего значения применяется функция nextval(), а не выражение NEXT VALUE FOR, как того требует стандарт;
- предложение OWNED BY является расширением Синергия-БД.

43. CREATE SERVER

CREATE SERVER – создать сторонний сервер.

43.1. Синтаксис

```
CREATE SERVER имя_сервера [ TYPE 'тип_сервера' ]
[ VERSION 'версия_сервера' ]
    FOREIGN DATA WRAPPER имя_обертки_сторонних_данных
    [ OPTIONS ( параметр 'значение' [, ... ] ) ]
```

43.2. Описание

CREATE SERVER создает сторонний сервер. Владелец сервера становится создавший его пользователь.

Определение стороннего сервера обычно включает информацию о подключении, которую использует обертка сторонних данных для доступа к внешнему ресурсу. Определяя сопоставления пользователей, можно установить и другие параметры подключения, связанные с пользователями.

Имя сервера должно быть уникальным в базе данных.

Для создания сервера требуется право USAGE для обертки сторонних данных.

43.3.Параметры

IF NOT EXISTS

Не считать ошибкой, если сервер с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующий сервер как-то соотносится с тем, который мог бы быть создан.

имя_сервера

Имя создаваемого стороннего сервера.

тип_сервера

Необязательный тип сервера, может быть полезен для оберток сторонних данных.

версия_сервера

Необязательная версия сервера, может быть полезна для оберток сторонних данных.

имя_обертки_сторонних_данных

Имя обертки сторонних данных, управляющей сервером.

OPTIONS (параметр 'значение' [, ...])

Это предложение определяет параметры сервера. Эти параметры обычно задают свойства подключения к серверу; их конкретные имена и значения зависят от обертки сторонних данных.

43.4.Замечания

При использовании модуля dblink имя стороннего сервера может служить аргументом функции dblink_connect, определяющим параметры подключения. Для такого варианта использования необходимо иметь право USAGE для стороннего

сервера.

43.5.Примеры

Создание сервера myserver, доступного через обертку postgres_fdw:

```
CREATE SERVER myserver FOREIGN DATA WRAPPER postgres_fdw  
OPTIONS (host 'foo', dbname 'foodb', port '5432');
```

43.6.Совместимость

CREATE SERVER соответствует стандарту ISO/IEC 9075-9 (SQL/MED).

44. CREATE STATISTICS

CREATE STATISTICS – создать расширенную статистику.

44.1.Синтаксис

```
CREATE STATISTICS [ IF NOT EXISTS ] имя_статистики  
    [ ( вид_статистики [, ... ] ) ]  
    ON имя_столбца, имя_столбца [, ...]  
    FROM имя_таблицы
```

44.2.Описание

Команда CREATE STATISTICS создаст новый объект расширенной статистики, отслеживающий данные определённой таблицы, сторонней таблицы или материализованного представления. Объект статистики будет создан в текущей базе данных, и его владельцем станет пользователь, выполняющий команду.

Если задано имя схемы (например, CREATE STATISTICS myschema.mystat ...), объект статистики создаётся в указанной схеме, в противном случае – в текущей. Имя объекта статистики должно отличаться от имён других объектов статистики в этой схеме.

44.3.Параметры

IF NOT EXISTS

Не считать ошибкой, если объект статистики с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что при этом проверяется только

имя объекта, а не характеристики его определения.

имя_статистики

Имя создаваемого объекта статистики (возможно, дополненное схемой).

вид_статистики

Вид статистики, которая будет вычисляться в этом объекте. В настоящее время поддерживается статистика `ndistinct`, подсчёт числа различных значений, и `dependencies`, определение функциональных зависимостей. Если это предложение опущено, в объект статистики включаются все поддерживаемые виды статистики.

имя_столбца

Имя столбца таблицы, который будет покрываться вычисляемой статистикой. Необходимо указать имена минимум двух столбцов.

имя_таблицы

Имя (возможно, дополненное схемой) таблицы, содержащей столбцы, по которым создаётся статистика.

44.4.Замечания

Чтобы создать объект статистики, читающий таблицу, необходимо быть владельцем этой таблицы. После создания объекта статистики его владелец может определяться независимо от нижележащей таблицы.

44.5.Примеры

В данном примере создаётся таблица `t1` с двумя функционально зависимыми столбцами; то есть знания значения одного столбца достаточно, чтобы определить значение другого. Затем для этих столбцов строится статистика функциональной зависимости:

```
CREATE TABLE t1 (  
    a    int,  
    b    int  
);
```



```
INSERT INTO t1 SELECT i/100, i/500
      FROM generate_series(1,1000000) s(i);
```

```
ANALYZE t1;
```

```
-- число совпадающих строк будет катастрофически
недооценено:
```

```
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b =
0);
```

```
CREATE STATISTICS s1 (dependencies) ON a, b FROM t1;
```

```
ANALYZE t1;
```

```
-- теперь оценка числа строк стала точнее:
```

```
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b =
0);
```

Без статистики функциональной зависимости планировщик предположил бы, что два условия WHERE независимы друг от друга, и перемножил бы их оценки избирательности, что дало бы слишком заниженную оценку числа строк. Однако с созданной статистикой планировщик понимает, что условия WHERE избыточны и не ошибается с этой оценкой.

44.6. Совместимость

Команда CREATE STATISTICS отсутствует в стандарте SQL.

45. CREATE SUBSCRIPTION

CREATE SUBSCRIPTION – создать подписку.

45.1. Синтаксис

```
CREATE SUBSCRIPTION имя_подписки
      CONNECTION 'строка_подключения'
      PUBLICATION имя_публикации [, ...]
      [ WITH ( параметр_подписки [= значение] [, ...] ) ]
```

45.2.Описание

CREATE SUBSCRIPTION создаёт подписку для текущей базы данных. Имя подписки должно отличаться от имён других существующих подписок в текущей базе.

Подписка представляет собой реплицирующее подключение к публикующему серверу. Поэтому данная команда не только добавляет определения подписки в локальные каталоги, но также создаёт слот репликации на удалённом сервере.

В момент фиксации транзакции, в рамках которой выполняется эта команда, будет запущен рабочий процесс логической репликации.

45.3.Параметры

`имя_подписки`

Имя новой подписки.

`CONNECTION 'строка_подключения'`

Строка подключения к публикующему серверу.

`PUBLICATION имя_публикации`

Имена публикаций на публикующем сервере, на которые оформляется подписка.

`WITH ( параметр_подписки [= значение] [, ... ] )`

В этом предложении могут задаваться следующие необязательные параметры подписки:

`copy_data (boolean)`

Определяет, должны ли копироваться существующие данные в публикациях, на которые оформляется подписка, сразу после начала репликации. Значение по умолчанию – true.

`create_slot (boolean)`

Определяет, должна ли команда создавать слот репликации на публикующем сервере. Значение по умолчанию – true.

`enabled (boolean)`

Определяет, активировать ли репликацию в подписке, или её нужно только настроить, но не запускать сразу. Значение по умолчанию – true.

`slot_name (string)`

Имя слота репликации, которое должно использоваться. По умолчанию в качестве имени слота используется имя подписки.

Когда в качестве `slot_name` задаётся `NONE`, с подпиской не будет связан слот репликации. Это может быть полезно, если слот репликации позднее будет создаваться вручную. У таких подписок также должны быть равны `false` свойства `enabled` и `create_slot`.

`synchronous_commit (enum)`

Значение этого параметра переопределяет свойство `synchronous_commit`. По умолчанию – `off`.

Значение `off` безопасно для логической репликации: если подписчик потеряет транзакции из-за нарушения синхронизации, данные будут повторно переданы с публикующего сервера.

При выполнении синхронной логической репликации может быть уместно другое значение. Рабочие процессы логической репликации передают позиции записанных и сохранённых на диске данных публикующему серверу, так что при синхронной репликации он будет ждать завершения сохранения. Это значит, что значение `off` параметра `synchronous_commit` на подписчике может увеличить задержку при выполнении `COMMIT` на сервере публикации. При таком сценарии может быть выгоднее задать для `synchronous_commit` значение `local` или выше.

`connect (boolean)`

Определяет, нужно ли при выполнении `CREATE SUBSCRIPTION` подключаться к публикующему серверу. Если равняется `false`, значениями по умолчанию параметров `enabled`, `create_slot` и `copy_data` тоже будет `false`.

Значение `false` параметра `connect` несовместимо со значением `true` параметров `enabled`, `create_slot` и `copy_data`.

Так как со значением `false` соединение не устанавливается, подписка на таблицы не оформляется, так что после включения подписки ничего не будет реплицироваться. Чтобы таблицы вошли в подписку, потребуется позже выполнить

```
ALTER SUBSCRIPTION ... REFRESH PUBLICATION.
```

45.4.Замечания

Подробнее о том, как организовать управление доступом подписчиков к публикующему серверу.

При создании слота репликации (поведение по умолчанию) `CREATE SUBSCRIPTION` нельзя выполнять внутри блока транзакции.

Создание подписки с подключением к тому же кластеру баз данных (например, для организации репликации между базами данных в одном кластере или в одной базе данных) будет успешным, только если слот репликации не создаётся той же командой. В противном случае команда `CREATE SUBSCRIPTION` зависнет. Чтобы оформить такую подписку, слот репликации нужно создать отдельно (воспользовавшись функцией `pg_create_logical_replication_slot` и передав ей имя модуля `pgoutput`) и создать подписку с параметром `create_slot = false`. Это ограничение реализации, которое может быть устранено в будущем выпуске.

45.5.Примеры

Создание подписки на репликации `mypublication` и `insert_only` на удалённом сервере с немедленным запуском репликации при фиксировании транзакции:

```
CREATE SUBSCRIPTION mysub
    CONNECTION 'host=192.168.1.50 port=5432
user=foo dbname=foodb'
    PUBLICATION mypublication, insert_only;
```

Создание подписки на публикацию `insert_only` на удалённом сервере с отключением репликации для запуска в будущем.

```
CREATE SUBSCRIPTION mysub
    CONNECTION 'host=192.168.1.50 port=5432
user=foo dbname=foodb'
    PUBLICATION insert_only
    WITH (enabled = false);
```

45.6. Совместимость

CREATE SUBSCRIPTION является расширением Синергия-БД.

46. CREATE TABLE

CREATE TABLE – создать таблицу.

46.1. Синтаксис

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } |  
UNLOGGED ] TABLE [ IF NOT EXISTS ] имя_таблицы ( [  
    { имя_столбца тип_данных [ COLLATE  
правило_сортировки ] [ ограничение_столбца [ ... ] ]  
    | ограничение_таблицы  
    | LIKE исходная_таблица  
[ вариант_копирования ... ] }  
    [, ... ]  
] )  
  
[ INHERITS ( таблица_родитель [, ... ] ) ]  
  
[ PARTITION BY { RANGE | LIST | HASH } ( { имя_столбца |  
( выражение ) } [ COLLATE правило_сортировки ]  
[ класс_операторов ] [, ... ] ) ]  
  
[ WITH ( параметр_хранения [= значение] [, ... ] ) |  
WITH OIDS | WITHOUT OIDS ]  
  
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]  
  
[ TABLESPACE табл_пространство ]  
  
  
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } |  
UNLOGGED ] TABLE [ IF NOT EXISTS ] имя_таблицы  
    OF имя_типа [ (  
        { имя_столбца [ WITH OPTIONS ] [ ограничение_столбца  
[ ... ] ]  
        | ограничение_таблицы }  
        [, ... ]  
    ) ]  
  
[ PARTITION BY { RANGE | LIST | HASH } ( { имя_столбца |  
( выражение ) } [ COLLATE правило_сортировки ]  
[ класс_операторов ] [, ... ] ) ]
```

```

[ WITH ( параметр_хранения [= значение] [, ... ] ) |
WITH OIDS | WITHOUT OIDS ]

[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]

[ TABLESPACE табл_пространство ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } |
UNLOGGED ] TABLE [ IF NOT EXISTS ] имя_таблицы
    PARTITION OF таблица_родитель [ (
    { имя_столбца [ WITH OPTIONS ] [ ограничение_столбца
[ ... ] ]
    | ограничение_таблицы }
    [, ... ]
) ] { FOR VALUES указание_границ_секции | DEFAULT }
[ PARTITION BY { RANGE | LIST | HASH } ( { имя_столбца |
( выражение ) } [ COLLATE правило_сортировки ]
[ класс_операторов ] [, ... ] ) ]
[ WITH ( параметр_хранения [= значение] [, ... ] ) |
WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE табл_пространство ]

```

Здесь ограничение_столбца:

```

[ CONSTRAINT имя_ограничения ]
{ NOT NULL |
NULL |
CHECK ( выражение ) [ NO INHERIT ] |
DEFAULT выражение_по_умолчанию |
GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
[ ( параметры_последовательности ) ] |
UNIQUE параметры_индекса |
PRIMARY KEY параметры_индекса |
REFERENCES целевая_таблица [ ( целевой_столбец ) ]
[ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
[ ON DELETE действие ] [ ON UPDATE действие ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED |
INITIALLY IMMEDIATE ]

```

и ограничение_таблицы:

```
[ CONSTRAINT имя_ограничения ]
{ CHECK ( выражение ) [ NO INHERIT ] |
  UNIQUE ( имя_столбца [, ... ] ) параметры_индекса |
  PRIMARY KEY ( имя_столбца [, ... ] ) параметры_индекса
|
  EXCLUDE [ USING метод_индекса ] ( элемент_исключения
WITH оператор [, ... ] ) параметры_индекса [ WHERE
( предикат ) ] |
  FOREIGN KEY ( имя_столбца [, ... ] ) REFERENCES
целевая_таблица [ ( целевой_столбец [, ... ] ) ]
[ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON
DELETE действие ] [ ON UPDATE действие ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED |
INITIALLY IMMEDIATE ]
```

и вариант_копирования:

```
{ INCLUDING | EXCLUDING } { COMMENTS | CONSTRAINTS |
DEFAULTS | IDENTITY | INDEXES | STATISTICS | STORAGE |
ALL }
```

и указание_границ_секции:

```
IN ( { числовая_константа | строковая_константа | TRUE |
FALSE | NULL } [, ...] ) |
FROM ( { числовая_константа | строковая_константа | TRUE
| FALSE | MINVALUE | MAXVALUE } [, ...] )
TO ( { числовая_константа | строковая_константа | TRUE
| FALSE | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS числовая_константа, REMAINDER
числовая_константа )
```

параметры_индекса в ограничениях UNIQUE, PRIMARY KEY и EXCLUDE:

```
[ INCLUDE ( имя_столбца [, ... ] ) ]
[ WITH ( параметр_хранения [= значение] [, ... ] ) ]
[ USING INDEX TABLESPACE табл_пространство ]
```

элемент_исключения в ограничении EXCLUDE:

```
{ имя_столбца | ( выражение ) } [ класс_операторов ]
[ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

46.2.Описание

CREATE TABLE создает новую, изначально пустую таблицу в текущей базе данных. Владелец таблицы будет пользователь, выполнивший эту команду.

Если задано имя схемы (например, CREATE TABLE myschema.mytable ...), таблица создается в указанной схеме, в противном случае – в текущей. Временные таблицы существуют в специальной схеме, так что при создании таких таблиц имя схемы задать нельзя. Имя таблицы должно отличаться от имен других таблиц, последовательностей, индексов, представлений или сторонних таблиц в этой схеме.

CREATE TABLE также автоматически создает составной тип данных, соответствующий одной строке таблицы. Таким образом, имя таблицы не может совпадать с именем существующего типа в этой же схеме.

Необязательные предложения ограничений задают ограничения (проверки), которым должны удовлетворять добавляемые или изменяемые строки, чтобы операция добавления или изменения была выполнена успешно. Ограничение представляет собой SQL-объект, помогающий некоторым способом определить множество допустимых значений в таблице.

Определить ограничения можно двумя способами: в виде ограничений таблицы и в виде ограничений колонки. Ограничение колонки определяется как часть определения колонки, а ограничение таблицы не привязывается к конкретной колонке и может задействовать несколько колонок. Любые ограничения колонок можно также записать в виде ограничения таблицы, они введены просто для удобства записи в случаях, когда ограничение затрагивает только одну колонку.

Чтобы создать таблицу, необходимо иметь право USAGE для типов всех колонок или типа в предложении OF, соответственно.

46.3.Параметры

TEMPORARY или TEMP

С таким указанием таблица создается как временная. Временные таблицы автоматически удаляются в конце сеанса или могут удаляться в конце текущей

транзакции. Существующая постоянная таблица с тем же именем не будет видна в текущем сеансе, пока существует временная, однако к ней можно обратиться, дополнив имя указанием схемы. Все индексы, создаваемые для временной таблицы, так же автоматически становятся временными.

Демон автоочистки не может прочитать и, как следствие, сжимать и анализировать временные таблицы. По этой причине соответствующие операции очистки и анализа следует выполнять, вызывая SQL-команды в рамках сеанса. Например, если временную таблицу планируется использовать в сложных запросах, будет разумным выполнить для нее `ANALYZE` после того, как она будет наполнена.

По желанию можно добавить указание `GLOBAL` или `LOCAL` перед `TEMPORARY` или `TEMP`. В настоящее время это не имеет значения для Синергия-БД и считается устаревшей возможностью.

`UNLOGGED`

С этим указанием таблица создается как нежурналируемая. Данные, записываемые в нежурналируемые таблицы, не проходят через журнал упреждающей записи, в результате чего такие таблицы работают гораздо быстрее обычных. Однако, они не защищены от сбоя; при сбое или аварийном отключении сервера нежурналируемая таблица автоматически усекается. Кроме того, содержимое нежурналируемой таблицы не реплицируется на резервные серверы. Любые индексы, создаваемые для нежурналируемой таблицы, автоматически становятся нежурналируемыми.

`IF NOT EXISTS`

Не считать ошибкой, если отношение с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее отношение как-то соотносится с тем, которое могло бы быть создано.

`имя_таблицы`

Имя создаваемой таблицы (возможно, дополненное схемой).

`OF имя_типа`

Создаёт типизированную таблицу, структура которой определяется указанным составным типом (его имя может быть дополнено схемой). Типизированная таблица

привязана к породившему её типу; например, при удалении типа (командой `DROP TYPE ... CASCADE`) будет удалена и эта таблица.

Когда создаётся типизированная таблица, типы данных её столбцов определяются нижележащим составным типом, а не задаются командой `CREATE TABLE`. Но `CREATE TABLE` может добавить в таблицу значения по умолчанию и ограничения, а также задать параметры её хранения.

имя_столбца

Имя столбца, создаваемого в новой таблице.

тип_данных

Тип данных столбца (может включать определение массива с этим типом).

`COLLATE` правило_сортировки

Предложение `COLLATE` назначает правило сортировки для столбца (который должен иметь тип, поддерживающий сортировку). Если оно отсутствует, используется правило сортировки по умолчанию, установленное для типа данных столбца.

`INHERITS` (таблица_родитель [, ...])

Необязательное предложение `INHERITS` определяет список таблиц, от которых новая таблица будет автоматически наследовать все столбцы. Родительские таблицы могут быть обычными или сторонними таблицами.

При использовании `INHERITS` создаётся постоянная связь дочерней таблицы с родительскими. Изменения схемы в родительских таблицах обычно также отражаются в дочерних, и по умолчанию при чтении родительских таблиц в результат включаются данные дочерней таблицы.

Когда в нескольких родительских таблицах оказываются столбцы с одним именем, происходит ошибка, за исключением случая, когда типы данных всех этих столбцов в таблицах совпадают. В этом случае одноимённые столбцы объединяются и формируют один столбец в новой таблице. Если имя столбца новой таблицы совпадает с именем одного из унаследованных столбцов, их типы так же должны совпадать, и в этом случае определения столбцов тоже сливаются в одну. Если в новой таблице явно указывается значение по умолчанию для нового столбца, это

значение переопределяет любые значения по умолчанию, унаследованные этим столбцом. В противном случае, если значения по умолчанию определяются в разных родительских таблицах, эти определения должны совпадать, иначе произойдёт ошибка.

Ограничения CHECK объединяются вместе по сути так же, как и столбцы: если несколько родительских таблиц и/или определение новой таблицы содержат одноимённые ограничения CHECK, этим ограничениям должны соответствовать одинаковые выражения проверки, в противном случае произойдёт ошибка. В случае совпадения выражения, эти ограничения с данным выражением объединяются в одно. При этом ограничения со свойством NO INHERIT в родительской таблице исключаются из рассмотрения. Заметьте, что безымянное ограничение CHECK в новой таблице никогда не сливается с другими, так как для него всегда выбирается уникальное имя.

Параметры STORAGE для столбца так же копируются из родительских таблиц.

Если столбец в родительской таблице является столбцом идентификации, это свойство не наследуется. Если требуется, в дочерней таблице этот столбец можно объявить столбцом идентификации.

```
PARTITION BY { RANGE | LIST | HASH } ( { имя_столбца |  
( выражение ) } [ класс_операторов ] [, ...] )
```

Необязательное предложение PARTITION BY задаёт стратегию секционирования таблицы. Таблица, созданная с этим указанием, называется *секционируемой* таблицей. Задаваемый в скобках список столбцов или выражений формирует *ключ разбиения* таблицы. Для разбиения по диапазонам или по хешу ключ разбиения может включать несколько столбцов или выражений (до 32, но этот предел можно изменить при сборке Синергия-БД), но для разбиения по спискам ключ должен состоять из одного столбца или выражения.

Для секционирования по диапазонам и по спискам нужен класс операторов В-дерева, тогда как для секционирования по хешу требуется класс операторов хеширования. Если класс операторов не задан явно, будет применён класс операторов по умолчанию для соответствующего типа; в случае отсутствия такого

класса выдаётся ошибка. Для секционирования по хешу применяемый класс операторов должен реализовывать опорную функцию 2.

Секционируемая таблица разделяется на подтаблицы (называемые секциями), которые создаются отдельными командами `CREATE TABLE`. Сама по себе секционируемая таблица не содержит данных. Строка данных, вставляемая в эту таблицу, перенаправляется в секцию в зависимости от значений столбцов или выражений в ключе разбиения. Если значениям в новой строке не соответствует ни одна из существующих секций, возникает ошибка.

Секционируемые таблицы не поддерживают ограничения `EXCLUDE`; однако вы можете определить такие ограничения в отдельных секциях. Кроме того, в настоящее время не поддерживается создание внешних ключей, ссылающихся на секционированные таблицы, хотя в секционированных таблицах можно создать ограничения `PRIMARY KEY`.

```
PARTITION OF таблица_родитель { FOR VALUES  
указание_границ_секции | DEFAULT }
```

Создаёт таблицу в виде *секции* указанной родительской таблицы. Таблицу можно создать либо как секцию для определённых значений (используя `FOR VALUES`), либо как секцию по умолчанию (используя `DEFAULT`). Это указание неприемлемо для таблиц, секционируемых по хешу.

Здесь *указание_границ_секции* должно соответствовать методу и ключу секционирования родительской таблицы и не должно конфликтовать с другой существующей секцией того же родителя. Вариант указания с `IN` используется для секционирования по спискам, тогда как вариант с `FROM` и `TO` для секционирования по диапазонам, а с `WITH` – для секционирования по хешу.

В качестве значений, задаваемых в *указании_границ_секции*, принимается буквальное значение, `NULL` и `MINVALUE` или `MAXVALUE`. Буквальным значением должна быть либо числовая константа, приводимая к типу соответствующего столбца ключа разбиения, либо строковая константа, представляющая допустимое входное значение для этого типа.

При создании секции с разбиением по спискам возможно указать, что столбец

ключа разбиения может содержать NULL, включив в список секции NULL. Однако у отдельно взятой родительской таблицы может быть не больше одной такой секции. Для диапазонных секций NULL задать нельзя.

При создании диапазонной секции нижняя граница, задаваемая во FROM, включается в диапазон, а верхняя граница, задаваемая в TO – исключается. То есть, значения, задаваемые в списке FROM, являются допустимыми значениями соответствующих столбцов ключа разбиения для этой секции, а значения в списке TO – нет. Заметьте, что это утверждение должно восприниматься с учётом правил сравнения строк таблицы. Например, с разбиением PARTITION BY RANGE (x, y), секция с границами FROM (1, 2) TO (3, 4) примет x=1 с любым значением y>=2, x=2 с любым y, отличным от NULL, и x=3 с любым y<4.

Специальные значения MINVALUE и MAXVALUE могут использоваться при создании диапазонной секции для указания, что нижняя или верхняя граница для значений столбца отсутствует. Например, секция, определённая с указанием FROM (MINVALUE) TO (10), будет принимать любые значения меньше 10, а секция, определённая с указанием FROM (10) TO (MAXVALUE), – любые значения, которые больше или равны 10.

При создании диапазонной секции с более чем одним столбцом может также иметь смысл использовать MAXVALUE в определении нижней границы, а MINVALUE – верхней. Например, секция, определённая с указанием FROM (0, MAXVALUE) TO (10, MAXVALUE), будет принимать любые строки, в которых первый столбец ключа разбиения больше 0 и меньше или равен 10. Подобным образом, секция, определённая с указанием FROM ('a', MINVALUE) TO ('b', MINVALUE), будет принимать строки, в которых первый столбец ключа разбиения начинается с «a».

Заметьте, что если для одного столбца в границе секции задаётся MINVALUE или MAXVALUE, то же значение должно применяться и для всех последующих столбцов. Например, граница (10, MINVALUE, 0) будет некорректной; допустимая граница: (10, MINVALUE, MINVALUE).

Также заметьте, что для некоторых типов элементов, таких как `timestamp`, наряду с другими значениями допускается значение «infinity» (бесконечность). Оно отличается от вариантов `MINVALUE` и `MAXVALUE`, которые на самом деле не обозначают значения, которые можно сохранить, а просто говорят о том, это значение не ограничено. `MAXVALUE` можно воспринимать как значение, которое больше любого другого, включая «бесконечность», а `MINVALUE` меньше любого другого значения, включая «минус бесконечность». Таким образом, диапазон `FROM ('infinity') TO (MAXVALUE)` не будет пустым, а будет принимать ровно одно значение – «infinity».

С указанием `DEFAULT` таблица присоединяется к родительской таблице как секция по умолчанию. Родительской таблицей может быть таблица, разбиваемая по спискам или диапазонам. Ключ разбиения, не попадающий ни в одну другую секцию данного родителя, будет отправлен в секцию по умолчанию. У отдельно взятой родительской таблицы может быть только одна секция по умолчанию.

Когда у таблицы есть секция по умолчанию (`DEFAULT`) и к ней добавляется новая секция, требуется просканировать существующую секцию по умолчанию и убедиться в том, что она не содержит строки, которые должны относиться к новой секции. Если она содержит большое количество строк, это сканирование может быть длительным. Сканирование не будет выполняться, если секция по умолчанию является сторонней таблицей или в ней есть ограничение, гарантирующее отсутствие в этой секции строк, подлежащих перемещению в новую секцию.

При создании секции с разбиением по хешу должен задаваться модуль и остаток. Модулем должно быть положительное число, а остатком неотрицательное число, меньшее модуля. Обычно при начальной настройке таблицы с секционированием по хешу нужно выбрать модуль, равный количеству секций, и назначить каждой секции этот модуль и разные остатки (см. примеры ниже). Однако секциям можно назначить и разные модули, с условием, что модули, назначенные секциям таблицы, разбиваемой по хешу, являются делителями следующих больших модулей. Это позволяет постепенно увеличивать число секций, не производя полное перемещение всех данных. Например, предположим, что у вас есть таблица,

разбиваемая по хешу на 8 секций, для каждой из которых назначен модуль 8, и возникла необходимость увеличить число секций до 16. Вы можете отсоединить одну из секций по модулю 8, создать две новые секции по модулю 16, покрывающих ту же часть пространства ключа (одну с остатком, равным остатку отсоединённой секции, а вторую с остатком, равным тому же остатку плюс 8), и вновь наполнить их данными. Затем вы можете повторять эту операцию (возможно, позже) для следующих секций по модулю 8, пока таковых не останется. Хотя и при таком подходе может потребоваться перемещать большие объёмы данных на каждом этапе, это всё же лучше, чем создавать абсолютно новую таблицу и перемещать все данные сразу.

В секции должны содержаться столбцы с теми же именами и типами, что и в секционированной таблице, к которой она относится. Если родитель имеет характеристику `WITH OIDS`, все секции должны иметь `OID`; родительский столбец `OID` будет наследоваться всеми секциями, как и любой другой столбец. Изменение имён или типов столбцов в секционируемой таблице, или добавление или удаление столбца `OID` будет автоматически распространяться во все секции. Ограничения `CHECK` будут наследоваться автоматически всеми секциями, но для отдельных секций могут быть заданы дополнительные ограничения `CHECK`; дополнительные ограничения с теми же именами и условиями, как в родительской таблицы, будут объединены с родительским ограничением. Также независимо для каждой секции могут быть заданы значения по умолчанию.

Строки, добавляемые в секционированную таблицу, будут автоматически перенаправляться в соответствующую секцию. Если подходящей секции не найдётся, произойдёт ошибка.

Такие операции, как `TRUNCATE`, обычно затрагивают и саму таблицу, и каскадно распространяются на все дочерние секции, но могут также выполняться в отдельных секциях. Заметьте, что для удаления секции с помощью `DROP TABLE` требуется установить блокировку `ACCESS EXCLUSIVE` в родительской таблице.

`LIKE` исходная_таблица [вариант_копирования ...]

Предложение `LIKE` определяет таблицу, из которой в новую таблицу будут

автоматически скопированы все имена столбцов, их типы данных и их ограничения на NULL.

В отличие от `INHERITS`, новая и исходная таблица становятся полностью независимыми после завершения создания. Изменения в исходной таблице не отражаются в новой, а данные новой таблицы не включаются в результат чтения исходной.

Выражения значений по умолчанию в определениях копируемых столбцов переносятся в новую таблицу, только если дополнительно указано `INCLUDING DEFAULTS`. Без этого указания выражения по умолчанию не копируются, вследствие чего в новой таблице скопированные столбцы получают значения по умолчанию `NULL`. Заметьте, что при копировании выражений по умолчанию, которые вызывают функции, модифицирующие БД, как например `nextval`, может образовываться функциональная связь исходной таблицы с новой.

Характеристика идентификации в определении скопированного столбца будет копироваться, только если в команде есть указание `INCLUDING IDENTITY`. Для каждого столбца идентификации в новой таблице создаётся новая последовательность, независимая от последовательностей, связанных со старой таблицей.

Ограничения `NULL` копируются в новую таблицу всегда, а ограничения `CHECK` копируются только при явном указании `INCLUDING CONSTRAINTS`. Это в равной мере касается ограничений как таблиц, так и столбцов.

Расширенная статистика копируется в новую таблицу, если указано `INCLUDING STATISTICS`.

Индексы, свойство `PRIMARY KEY`, а также ограничения `UNIQUE` и `EXCLUDE` копируются из исходной таблицы в новую, только если явно указано `INCLUDING INDEXES`. Имена для новых индексов и ограничений выбираются согласно стандартным правилам, независимо от того, как назывались исходные. Это позволяет избежать ошибок с дублированием имён с новыми индексами.

Параметры `STORAGE` будут перенесены в определение копируемых столбцов,

только если явно указывается `INCLUDING STORAGE`. По умолчанию параметры `STORAGE` исключаются, вследствие чего скопированные столбцы в новой таблице получают параметры по умолчанию, определённые соответствующим типом.

Комментарии для скопированных столбцов, ограничений и индексов будут копироваться, только если явно указано `INCLUDING COMMENTS`. По умолчанию комментарии исключаются, вследствие чего скопированные столбцы и ограничения в новой таблице оказываются без комментариев.

`INCLUDING ALL` представляет собой краткую форму предложения `INCLUDING COMMENTS INCLUDING CONSTRAINTS INCLUDING DEFAULTS INCLUDING IDENTITY INCLUDING INDEXES INCLUDING STATISTICS INCLUDING STORAGE`.

Заметьте, что в отличие от `INHERITS`, столбцы и ограничения, копируемые командой `LIKE`, не объединяются с одноимёнными столбцами и ограничениями. Если дублирующееся имя указывается явно или возникает в другом предложении `LIKE`, происходит ошибка.

Предложение `LIKE` может также применяться для копирования определений столбцов из представлений, сторонних таблиц и составных типов. Неприменимые параметры (например, `INCLUDING INDEXES` для представления) при этом игнорируются.

`CONSTRAINT имя_ограничения`

Необязательное имя столбца или ограничения таблицы. При нарушении ограничения его имя будет выводиться в сообщении об ошибках, так что имена ограничений вида `столбец должен быть положительным` могут сообщить полезную информацию об ограничении клиентскому приложению. Имена ограничений, включающие пробелы, необходимо заключать в двойные кавычки. Если имя ограничения не указано, система генерирует имя автоматически.

`NOT NULL`

Данный столбец не принимает значения `NULL`.

`NULL`

Данный столбец может содержать значения `NULL` (по умолчанию).

Это предложение предназначено только для совместимости с нестандартными базами данных SQL. Использовать его в новых приложениях не рекомендуется.

CHECK (выражение) [NO INHERIT]

В ограничении CHECK задаётся выражение, возвращающее булевский результат, по которому определяется, будет ли успешна операция добавления или изменения для конкретных строк. Операция выполняется успешно, если результат выражения равен TRUE или UNKNOWN. Если же для какой-нибудь строки, задействованной в операции добавления или изменения, будет получен результат FALSE, возникает ошибка, и эта операция не меняет ничего в базе данных. Ограничение-проверка, заданное как ограничение столбца, должно ссылаться только на значение самого столбца, тогда как ограничение на уровне таблицы может ссылаться и на несколько столбцов.

В настоящее время выражения CHECK не могут содержать подзапросы или ссылаться на переменные, кроме как на столбцы текущей строки. Также допустима ссылка на системный столбец tableoid, но не на другие системные столбцы.

Ограничение с пометкой NO INHERIT не будет наследоваться дочерними таблицами.

Когда для таблицы задано несколько ограничений CHECK, они будут проверяться для каждой строки в алфавитном порядке имён после проверки ограничений NOT NULL.

DEFAULT выражение_по_умолчанию

Предложение DEFAULT задаёт значение по умолчанию для столбца, в определении которого оно присутствует. Значение задаётся выражением без переменных (подзапросы и перекрёстные ссылки на другие столбцы текущей таблицы в нём не допускаются). Тип данных выражения, задающего значение по умолчанию, должен соответствовать типу данных столбца.

Это выражение будет использоваться во всех операциях добавления данных, в которых не задаётся значение данного столбца. Если значение по умолчанию не определено, таким значением будет NULL.

GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY

[(*параметры_последовательности*)]

С этим предложением столбец создаётся как *столбец идентификации*. С ним будет связана неявная последовательность, из которой этот столбец будет автоматически получать значения в новых строках.

Предложения ALWAYS и BY DEFAULT определяют, какой приоритет имеет заданное пользователем значение этого столбца в команде INSERT. С предложением ALWAYS пользовательское значение используется, только если в операторе INSERT указано OVERRIDING SYSTEM VALUE. С предложением BY DEFAULT пользовательскому значению отдаётся предпочтение. За подробностями обратитесь к INSERT. В команде COPY всегда используются пользовательские значения, вне зависимости от значения этого параметра.

Используя	необязательное	предложение
параметры_последовательности,	можно	переопределить свойства
последовательности.		

UNIQUE (ограничение столбца)

UNIQUE (*имя_столбца* [, ...]) [INCLUDE (*имя_столбца* [, ...])] (ограничение таблицы)

Ограничение UNIQUE определяет, что группа из одного или нескольких столбцов таблицы может содержать только уникальные значения. Ограничение уникальности для таблицы ведёт себя точно так же, как ограничение для столбца, но может охватывать несколько столбцов.

При проверке ограничения уникальности значения NULL не считаются равными.

В каждом ограничении уникальности для таблицы должен задаваться набор столбцов, отличный от набора любого другого ограничения уникальности или первичного ключа в данной таблице. Иначе это будет одно и то же ограничение, определённое многократно.

При установлении ограничения уникальности в многоуровневой иерархии секционирования в определение ограничения должны включаться все столбцы ключа разбиения целевой секционированной таблицы, а также столбцы всех

подчинённых секционированных таблиц.

При добавлении ограничения уникальности автоматически будет создан уникальный индекс-B-дерево по столбцу или группе столбцов, перечисленных в ограничении. Необязательное предложение INCLUDE добавляет к этому индексу один или несколько столбцов, для которых уникальность не будет требоваться. Заметьте, что хотя ограничение по включаемым столбцам не контролируется, оно всё же зависит от них. Как следствие, некоторые операции с этими столбцами (например, DROP COLUMN) могут повлечь каскадное удаление индекса и ограничения.

```
PRIMARY KEY (ограничение столбца)  
PRIMARY KEY ( имя_столбца [, ... ] ) [ INCLUDE  
( имя_столбца [, ...] ) ] (ограничение таблицы)
```

Ограничение PRIMARY KEY определяет, что столбец или столбцы таблицы могут содержать только уникальные (без повторов) значения, отличные от NULL. Для таблицы может быть задан только один первичный ключ, будь то ограничение столбца или ограничение таблицы.

В определении первичного ключа должен задаваться набор столбцов, отличный от набора любого другого ограничения уникальности, установленного для данной таблицы. В противном случае уникальное ограничение оказывается избыточным и будет отброшено.

PRIMARY KEY устанавливает для данных те же ограничения, что и сочетание UNIQUE и NOT NULL, но образование первичного ключа из набора столбцов также добавляет метаданные о конструкции схемы, так как первичный ключ подразумевает, что другие таблицы могут ссылаться на этот набор столбцов, как на уникальный идентификатор строк.

Ограничения PRIMARY KEY для секционированных таблиц подчиняются тем же требованиям, что и ограничения UNIQUE.

При добавлении ограничения первичного ключа (PRIMARY KEY) автоматически будет создан уникальный индекс-B-дерево по столбцу или группе столбцов, перечисленных в ограничении. Необязательное предложение INCLUDE

позволяет задать список столбцов, которые будут включены в неключевую часть индекса. Хотя уникальность по включаемым столбцам не контролируется, ограничение всё же зависит от них. Как следствие, некоторые операции с этими столбцами (например, DROP COLUMN) могут повлечь каскадное удаление индекса и ограничения.

```
EXCLUDE [ USING метод_индекса ] ( элемент_исключения  
WITH оператор [, ... ] ) параметры_индекса [ WHERE  
( предикат ) ]
```

Предложение EXCLUDE определяет ограничение-исключение, которое гарантирует, что для любых двух строк, сравниваемых по указанным столбцам или выражениям с указанными операторами, результат не будет равен TRUE для всех сравнений. Если все указанные операторы проверяют равенство, это ограничение равносильно ограничению UNIQUE, хотя обычное ограничение уникальности будет работать быстрее. С другой стороны, в ограничениях-исключениях можно задавать более общие условия, чем простое условие на равенство. Например, можно задать ограничение, требующее, чтобы никакие две строки в таблице не содержали пересекающихся кругов, применив оператор &&.

Ограничения-исключения реализуются с помощью индексов, так что каждый указанный в них оператор должен быть связан с соответствующим классом операторов для метода_индекса. Кроме того, операторы должны быть коммутативными. В каждом элементе_исключения можно дополнительно указать класс оператора и/или параметры сортировки, подробно описанные в CREATE INDEX.

Метод индекса должен поддерживать amgettuple; в настоящее время это означает, что метод GIN здесь применять нельзя. Хотя в ограничении-исключении можно использовать B-деревья и хеш-индексы, в этом мало смысла, так как такой подход ничем не лучше обычного ограничения уникальности. Так что на практике методом индекса всегда будет GiST или SP-GiST.

Параметр предикат позволяет указать ограничение-исключение для подмножества таблицы; внутри при этом создаётся частичный индекс. Заметьте, что

предикат необходимо заключить в скобки.

```
REFERENCES внешняя_таблица [ ( внешний_столбец ) ]
[ MATCH тип_совпадения ] [ ON DELETE действие ] [ ON
UPDATE действие ] (ограничение столбца)
FOREIGN KEY ( имя_столбца [, ... ] ) REFERENCES
внешняя_таблица [ ( внешний_столбец [, ... ] ) ] [ MATCH
тип_совпадения ] [ ON DELETE действие ] [ ON UPDATE
действие ] (ограничение таблицы)
```

Эти предложения определяют ограничение внешнего ключа, требующее, чтобы группа из одного или нескольких столбцов новой таблицы содержала только такие значения, которым соответствуют значения в заданных столбцах некоторой строки во внешней таблице. Если список целевых_столбцов опущен, в качестве него используется первичный ключ целевой_таблицы. В качестве целевых столбцов должны указываться столбцы неоткладываемого уникального ограничения или первичного ключа во внешней таблице. При этом пользователь должен иметь право REFERENCES во внешней таблице (либо для всей таблицы, либо только для целевых столбцов). Для добавления ограничения внешнего ключа требуется блокировка SHARE ROW EXCLUSIVE в целевой таблице. Заметьте, что нельзя определить ограничения внешнего ключа, связывающие временные и постоянные таблицы. Также заметьте, что нельзя определить внешний ключ, который будет ссылаться на секционированную таблицу, хотя определить внешний ключ в секционированной таблице можно.

Значения, вставляемые в ссылающиеся столбцы, сверяются со значениями во внешних столбцах внешней таблицы с учётом заданного типа совпадения. Возможны три типа совпадения: MATCH FULL (полное совпадение), MATCH PARTIAL (частичное совпадение) и тип по умолчанию, MATCH SIMPLE (простое совпадение). С MATCH FULL ни один из столбцов составного внешнего ключа не может содержать NULL, кроме случая, когда все внешние столбцы NULL; в этом случае строка может не иметь соответствия во внешней таблице. С MATCH SIMPLE любой из столбцов внешнего ключа может содержать NULL; при этом строка с NULL в одном из таких столбцов может не иметь соответствия во внешней таблице. Тип MATCH PARTIAL ещё не реализован. Чтобы вопросы со сравнением NULL не

возникали, к столбцам, ссылающимся на внешние, можно применить ограничения NOT NULL.

Кроме того, при изменении значений во внешних столбцах с данными в столбцах этой таблицы могут производиться определённые действия. Предложение ON DELETE задаёт действие, производимое при удалении некоторой строки во внешней таблице. Предложение ON UPDATE подобным образом задаёт действие, производимое при изменении значения в целевых столбцах внешней таблицы. Если строка изменена, но это изменение не затронуло целевые столбцы, никакое действие не производится. Ссылочные действия, кроме NO ACTION, нельзя сделать откладываемыми, даже если ограничение объявлено как откладываемое. Для каждого предложения возможные следующие варианты действий:

NO ACTION

Выдать ошибку, показывающую, что при удалении или изменении записи произойдёт нарушение ограничения внешнего ключа. Для отложенных ограничений ошибка произойдёт в момент проверки ограничения, если строки, ссылающиеся на эту запись, по-прежнему будут существовать. Этот вариант действия подразумевается по умолчанию.

RESTRICT

Выдать ошибку, показывающую, что при удалении или изменении записи произойдёт нарушение ограничения внешнего ключа. Этот вариант подобен NO ACTION, но эта проверка будет неоткладываемой.

CASCADE

Удалить все строки, ссылающиеся на удаляемую запись, либо поменять значения в ссылающихся столбцах на новые значения во внешних столбцах, в соответствии с операцией.

SET NULL

Установить ссылающиеся столбцы равными NULL.

SET DEFAULT

Установить в ссылающихся столбцах значения по умолчанию. Если эти значения не равны NULL, во внешней таблице должна быть строка, соответствующая

набору значений по умолчанию; в противном случае операция завершится ошибкой.

Если внешние столбцы меняются часто, будет разумным добавить индекс для ссылающихся столбцов, чтобы действия по обеспечению ссылочной целостности, связанные с ограничением внешнего ключа, выполнялись более эффективно.

```
DEFERRABLE  
NOT DEFERRABLE
```

Это предложение определяет, может ли ограничение быть отложенным. Неоткладываемое ограничение будет проверяться немедленно после каждой команды. Проверка откладываемых ограничений может быть отложена до завершения транзакции (обычно с помощью команды SET CONSTRAINTS). По умолчанию подразумевается вариант NOT DEFERRABLE. В настоящее время это предложение принимают только ограничения UNIQUE, PRIMARY KEY, EXCLUDE и REFERENCES (внешний ключ). Ограничения NOT NULL и CHECK не могут быть отложенными. Заметьте, что откладываемые ограничения не могут применяться в качестве решающих при конфликте в операторе INSERT с предложением ON CONFLICT DO UPDATE.

```
INITIALLY IMMEDIATE  
INITIALLY DEFERRED
```

Для откладываемых ограничений это предложение определяет, когда ограничение должно проверяться по умолчанию. Ограничение с характеристикой INITIALLY IMMEDIATE (подразумеваемой по умолчанию) проверяется после каждого оператора. Ограничение INITIALLY DEFERRED, напротив, проверяется только в конце транзакции. Время проверки ограничения можно изменить явно с помощью команды SET CONSTRAINTS.

```
WITH ( параметр_хранения [= значение] [, ... ] )
```

Это предложение определяет необязательные параметры хранения для таблицы или индекса. Предложение WITH для таблицы может также включать указание OIDS=TRUE (или просто OIDS), устанавливающее, что каждая строка таблицы должна иметь собственный OID (Object Identifier, идентификатор объекта), или указание OIDS=FALSE, устанавливающее, что строки не содержат OID. Если

указание `OIDS` отсутствует, значение этого свойства по умолчанию зависит от конфигурационного параметра `default_with_oids`. (Если новая таблица унаследована от каких-либо таблиц, имеющих `OID`, свойство `OIDS=TRUE` задаётся принудительно, даже если в команде явно написано `OIDS=FALSE`.)

Если явно указано `OIDS=FALSE` или это подразумевается, в новой таблице не будут храниться значения `OID` и новый `OID` не будет генерироваться для каждой добавляемой в неё строки. Для обычных таблиц такое поведение предпочтительнее, так как оно сокращает потребление `OID` и тем самым откладывает заикливание 32-битного счётчика `OID`. Как только происходит заикливание, значения `OID` больше нельзя считать уникальными, что делает их значительно менее полезными. К тому же, исключение столбца `OID` из таблицы сокращает объём, необходимый для хранения таблицы на диске, на 4 байта для каждой строки (на большинстве платформ), что несколько улучшает производительность.

Для удаления данных `OID` из таблицы после её создания воспользуйтесь командой `ALTER TABLE`.

```
WITH OIDS
WITHOUT OIDS
```

Это устаревшее написание указаний `WITH (OIDS)` и `WITH (OIDS=FALSE)`, соответственно. Если требуется определить одновременно свойство `OIDS` и параметры хранения, необходимо использовать синтаксис `WITH ( ... )`.

```
ON COMMIT
```

Поведением временных таблиц в конце блока транзакции позволяет управлять предложение `ON COMMIT`, которое принимает три параметра:

```
PRESERVE ROWS
```

Никакое специальное действие в конце транзакции не выполняется. Это поведение по умолчанию.

```
DELETE ROWS
```

Все строки в этой временной таблице будут удаляться в конце каждого блока транзакции. По сути, при каждой фиксации транзакции будет автоматически выполняться `TRUNCATE`. В случае секционированной таблицы это действие не

распространяется на её секции.

DROP

Временная таблица будет удалена в конце текущего блока транзакции. Если это секционированная таблица, будут удалены и все её секции. Если у таблицы есть потомки в иерархии наследования, они также будут удалены.

TABLESPACE табл_пространство

Здесь табл_пространство – имя табличного пространства, в котором будет создаваться новая таблица. Если оно не указано, выбирается default_tablespace или temp_tablespaces, если таблица временная.

USING INDEX TABLESPACE табл_пространство

Это предложение позволяет выбрать табличное пространство, в котором будут создаваться индексы, связанные с ограничениями UNIQUE, PRIMARY KEY или EXCLUDE. Если оно не указано, выбирается default_tablespace или temp_tablespaces, если таблица временная.

46.4.Параметры хранения

Предложение WITH позволяет установить параметры хранения для таблиц и индексов, связанных с ограничениями UNIQUE, PRIMARY KEY и EXCLUDE. Параметры хранения для индексов документированы в CREATE INDEX. Поддерживаемые в настоящее время параметры хранения для таблиц перечислены ниже. Как показано, для многих параметров существует дополнительный параметр с тем же именем и префиксом toast., который управляет поведением вторичной таблицы TOAST, если она есть. Если значение некоторого параметра задано для таблицы, а значение равнозначного параметра toast. не определено, для таблицы TOAST будет применяться значение параметра основной таблицы. Возможность задания этих параметров для секционированных таблиц не поддерживается, но вы можете задать их для отдельных конечных секций.

Примечание. Параметр хранения MACS=true, позволяющий создавать таблицы со строками, защищенными метками конфиденциальности.

fillfactor (integer)

Фактор заполнения для таблицы, задаваемый в процентах, от 10 до 100.

Значение по умолчанию – 100 (плотное заполнение). При меньшем факторе заполнения операции INSERT упаковывают данные в страницы только до заданного процента; оставшееся место резервируется для изменения строк на этой странице. В результате UPDATE получает шанс поместить измененную копию строки в ту же страницу, что и исходную, что гораздо эффективнее, чем размещать её на другой странице. Для таблиц, записи в которых никогда не меняются, лучшим выбором будет плотное заполнение, но для активно изменяемых таблиц лучше выбрать меньший фактор заполнения. Этот параметр нельзя задать для таблиц TOAST.

```
toast_tuple_target (integer)
```

Параметр `toast_tuple_target` задаёт минимальную длину кортежа, после превышения которой мы будем пытаться переносить значения больших столбцов в таблицы TOAST и до которой мы будем пытаться сократить размер кортежа после перехода к TOAST. Это затрагивает только столбцы с пометкой `External` (внешние) или `Extended` (расширенные) и касается только новых кортежей – на существующие это не влияет. По умолчанию этот параметр имеет значение, позволяющее разместить минимум 4 кортежа в блоке, что при стандартном размере блока составляет 2040 байт. Допустимые значения лежат в интервале от 128 байт до (`размер_блока` - заголовок), по умолчанию 8160 байт. Изменение этого значения может не отражаться на очень коротких и очень длинных кортежах. Заметьте, что выбранное по умолчанию значение часто близко к оптимальному, и весьма вероятно, что изменение этого параметра в некоторых случаях будет иметь отрицательный эффект. Для таблиц TOAST этот параметр задать нельзя.

```
parallel_workers (integer)
```

Данный параметр задаёт число рабочих процессов, которые должны задействоваться при параллельном сканировании таблицы. Если это значение не задано, система будет определять его, исходя из размера отношения. Фактическое число рабочих процессов, выбранное планировщиком или служебными операторами, выполняющими параллельное сканирование, может быть меньше, например, вследствие ограничения `max_worker_processes`.

```
autovacuum_enabled, toast.autovacuum_enabled (boolean)
```

Включает или отключает демон автоочистки для определённой таблицы. Со значением `true` демон автоочистки будет автоматически выполнять операции `VACUUM` и/или `ANALYZE` в этой таблице. Со значением `false` эта таблица не будет подвергаться автоочистке, если только это не потребуется для предотвращения заикливания идентификаторов транзакций. Заметьте, что демон автоочистки не будет запускаться вовсе (если только это не потребуется для предотвращения заикливания), если параметр `autovacuum` имеет значение `false`; это нельзя переопределить, установив параметры хранения для отдельных таблиц. Таким образом, явно устанавливать для этого параметра значение `true` практически не имеет смысла – полезно только значение `false`.

```
autovacuum_vacuum_threshold,  
toast.autovacuum_vacuum_threshold (integer)
```

Значение параметра `autovacuum_vacuum_threshold` для таблицы.

```
autovacuum_vacuum_scale_factor,  
toast.autovacuum_vacuum_scale_factor (float4)
```

Значение параметра `autovacuum_vacuum_scale_factor` для таблицы.

```
autovacuum_analyze_threshold (integer)
```

Значение параметра `autovacuum_analyze_threshold` для таблицы.

```
autovacuum_analyze_scale_factor (float4)
```

Значение параметра `autovacuum_analyze_scale_factor` для таблицы.

```
autovacuum_vacuum_cost_delay,  
toast.autovacuum_vacuum_cost_delay (integer)
```

Значение параметра `autovacuum_vacuum_cost_delay` для таблицы.

```
autovacuum_vacuum_cost_limit,  
toast.autovacuum_vacuum_cost_limit (integer)
```

Значение параметра `autovacuum_vacuum_cost_limit` для таблицы.

```
autovacuum_freeze_min_age,  
toast.autovacuum_freeze_min_age (integer)
```

Значение параметра `vacuum_freeze_min_age` для таблицы. Учтите, что система будет игнорировать установленные для таблиц значения `autovacuum_freeze_min_age`, превышающие половину системного `autovacuum_freeze_max_age`.

```
autovacuum_freeze_max_age,  
toast.autovacuum_freeze_max_age (integer)
```

Значение параметра `autovacuum_freeze_max_age` для таблицы. Учтите, что система будет игнорировать установленные для таблиц значения `autovacuum_freeze_max_age`, превышающие значение системного параметра (они могут быть только меньше).

```
autovacuum_freeze_table_age,  
toast.autovacuum_freeze_table_age (integer)
```

Значение параметра `vacuum_freeze_table_age` для таблицы.

```
autovacuum_multixact_freeze_min_age,  
toast.autovacuum_multixact_freeze_min_age (integer)
```

Значение параметра `vacuum_multixact_freeze_min_age` для таблицы. Учтите, что демон автоочистки будет игнорировать установленные для таблиц значения `autovacuum_multixact_freeze_min_age`, превышающие половину значения системного параметра `autovacuum_multixact_freeze_max_age`.

```
autovacuum_multixact_freeze_max_age,  
toast.autovacuum_multixact_freeze_max_age (integer)
```

Значение параметра `autovacuum_multixact_freeze_max_age` для таблицы. Учтите, что система автоочистки будет игнорировать установленные для таблиц параметры `autovacuum_multixact_freeze_max_age`, превышающие системный параметр (они могут быть только меньше).

```
autovacuum_multixact_freeze_table_age,  
toast.autovacuum_multixact_freeze_table_age (integer)
```

Значения параметра `vacuum_multixact_freeze_table_age` для таблицы.

```
log_autovacuum_min_duration,  
toast.log_autovacuum_min_duration (integer)
```

Значения параметра `log_autovacuum_min_duration` для таблицы.

```
user_catalog_table (boolean)
```

Объявляет таблицу как дополнительную таблицу каталога, например для целей логической репликации. Для таблиц TOAST этот параметр задать нельзя.

46.5.Замечания

Применять OID в новых приложениях не рекомендуется: по возможности лучше использовать в качестве первичного ключа таблицы SERIAL или другой генератор последовательности. Однако, если в вашем приложении для идентификации строк применяется OID, рекомендуется создать уникальное ограничение по колонке `oid` в этой таблице, чтобы значения OID в этой таблице на самом деле однозначно идентифицировали строки даже после заикливания счетчика. Также не стоит полагать, что значения OID уникальны в разных таблицах; если вам требуется идентификатор, уникальный в базе данных, воспользуйтесь для этого комбинацией `tableoid` и OID строки.

Примечание. Применять `OIDS=FALSE` не рекомендуется для таблиц без первичного ключа, так как без OID или уникального ключа данных сложно идентифицировать определенные строки.

Синергия-БД автоматически создает индекс, гарантирующий уникальность, для каждого ограничения уникальности и ограничения первичного ключа. Поэтому явно создавать индекс для колонок первичного ключа не требуется.

Ограничения уникальности и первичные ключи в текущей реализации не наследуются. Вследствие этого ограничения уникальности довольно плохо сочетаются с наследованием.

В таблице не может быть больше 1600 колонок. На практике фактический предел обычно ниже из-за ограничения на длину записи.

46.6.Примеры

Создание таблицы `films` и таблицы `distributors`:

```
CREATE TABLE films (  
    code          char(5) CONSTRAINT firstkey PRIMARY KEY,  
    title         varchar(40) NOT NULL,  
    did           integer NOT NULL,  
    date_prod     date,  
    kind          varchar(10),  
    len           interval hour to minute
```

);

```
CREATE TABLE distributors (  
    did      integer PRIMARY KEY DEFAULT  
nextval('serial'),  
    name     varchar(40) NOT NULL CHECK (name <> '')  
);
```

Создание таблицы с двумерным массивом:

```
CREATE TABLE array_int (  
    vector   int[][]  
);
```

Определение ограничения уникальности для таблицы films. Ограничения уникальности могут быть определены для одной или нескольких колонок таблицы:

```
CREATE TABLE films (  
    code      char(5),  
    title     varchar(40),  
    did       integer,  
    date_prod date,  
    kind      varchar(10),  
    len       interval hour to minute,  
    CONSTRAINT production UNIQUE(date_prod)  
);
```

Определение ограничения-проверки для колонки:

```
CREATE TABLE distributors (  
    did      integer CHECK (did > 100),  
    name     varchar(40)  
);
```

Определение ограничения-проверки для таблицы:

```
CREATE TABLE distributors (  
    did      integer,  
    name     varchar(40)  
    CONSTRAINT con1 CHECK (did > 100 AND name <> '')  
);
```

Определение ограничения первичного ключа для таблицы films:

```
CREATE TABLE films (  
    code          char(5),  
    title         varchar(40),  
    did           integer,  
    date_prod     date,  
    kind          varchar(10),  
    len           interval hour to minute,  
    CONSTRAINT code_title PRIMARY KEY(code,title)  
);
```

Определение ограничения первичного ключа для таблицы distributors.

Следующие два примера равнозначны, но в первом используется синтаксис ограничений для таблицы, а во втором – для колонки:

```
CREATE TABLE distributors (  
    did           integer,  
    name          varchar(40),  
    PRIMARY KEY(did)  
);
```

```
CREATE TABLE distributors (  
    did           integer PRIMARY KEY,  
    name          varchar(40)  
);
```

Определение значений по умолчанию: для колонки name значением по умолчанию будет строка, для колонки did – следующее значение объекта последовательности, а для modtime – время, когда была вставлена запись:

```
CREATE TABLE distributors (  
    name          varchar(40) DEFAULT 'Luso Films',  
    did           integer DEFAULT  
nextval('distributors_serial'),  
    modtime       timestamp DEFAULT current_timestamp  
);
```


Определение двух ограничений NOT NULL для колонок таблицы distributors, одному из которых дается явное имя:

```
CREATE TABLE distributors (  
    did      integer CONSTRAINT no_null NOT NULL,  
    name     varchar(40) NOT NULL  
);
```

Определение ограничения уникальности для колонки name:

```
CREATE TABLE distributors (  
    did      integer,  
    name     varchar(40) UNIQUE  
);
```

То же самое условие, но в виде ограничения таблицы:

```
CREATE TABLE distributors (  
    did      integer,  
    name     varchar(40),  
    UNIQUE (name)  
);
```

Создание такой же таблицы с фактором заполнения 70% для таблицы и её уникального индекса:

```
CREATE TABLE distributors (  
    did      integer,  
    name     varchar(40),  
    UNIQUE (name) WITH (fillfactor=70)  
)  
--  
WITH (fillfactor=70);
```

Создание таблицы circles с ограничением-исключением, не допускающим пересечения двух кругов:

```
CREATE TABLE circles (  
    c circle,  
    EXCLUDE USING gist (c WITH &&)  
);
```

Создание таблицы `cinemas` в табличном пространстве `diskvol1`:

```
CREATE TABLE cinemas (  
    id serial,  
    name text,  
    location text  
) TABLESPACE diskvol1;
```

Создание составного типа и типизированной таблицы:

```
CREATE TYPE employee_type AS (name text, salary  
numeric);
```

```
CREATE TABLE employees OF employee_type (  
    PRIMARY KEY (name),  
    salary WITH OPTIONS DEFAULT 1000  
);
```

Создание таблицы, секционированной по диапазонам:

```
CREATE TABLE measurement (  
    logdate          date not null,  
    peaktemp         int,  
    unitsales        int  
) PARTITION BY RANGE (logdate);
```

Создание таблицы, секционированной по диапазонам, с ключом разбиения, включающим несколько столбцов:

```
CREATE TABLE measurement_year_month (  
    logdate          date not null,  
    peaktemp         int,  
    unitsales        int  
) PARTITION BY RANGE (EXTRACT(YEAR FROM logdate),  
EXTRACT(MONTH FROM logdate));
```

Создание таблицы, секционированной по спискам:

```
CREATE TABLE cities (  
    city_id          bigserial not null,  
    name             text not null,  
    population       bigint
```

```
) PARTITION BY LIST (left(lower(name), 1));
```

Создание таблицы, секционированной по хешу:

```
CREATE TABLE orders (  
    order_id      bigint not null,  
    cust_id       bigint not null,  
    status        text  
)  
PARTITION BY HASH (order_id);
```

Создание секции таблицы, секционированной по диапазонам:

```
CREATE TABLE measurement_y2016m07  
    PARTITION OF measurement (  
        unitsales DEFAULT 0  
)  
FOR VALUES FROM ('2016-07-01') TO ('2016-08-01');
```

Создание нескольких секций для таблицы, секционированной по диапазонам, с ключом разбиения, включающим несколько столбцов:

```
CREATE TABLE measurement_ym_older  
    PARTITION OF measurement_year_month  
    FOR VALUES FROM (MINVALUE, MINVALUE) TO (2016, 11);
```

```
CREATE TABLE measurement_ym_y2016m11  
    PARTITION OF measurement_year_month  
    FOR VALUES FROM (2016, 11) TO (2016, 12);
```

```
CREATE TABLE measurement_ym_y2016m12  
    PARTITION OF measurement_year_month  
    FOR VALUES FROM (2016, 12) TO (2017, 01);
```

```
CREATE TABLE measurement_ym_y2017m01  
    PARTITION OF measurement_year_month  
    FOR VALUES FROM (2017, 01) TO (2017, 02);
```

Создание секции таблицы, секционированной по спискам:

```
CREATE TABLE cities_ab  
    PARTITION OF cities (  
        CONSTRAINT city_id_nonzero CHECK (city_id != 0)
```

```
) FOR VALUES IN ('a', 'b');
```

Создание секции таблицы, секционируемой по спискам (при этом сама секция также создаётся секционируемой), и добавление секции в неё:

```
CREATE TABLE cities_ab  
    PARTITION OF cities (  
        CONSTRAINT city_id_nonzero CHECK (city_id != 0)  
    ) FOR VALUES IN ('a', 'b') PARTITION BY RANGE  
    (population);
```

```
CREATE TABLE cities_ab_10000_to_100000  
    PARTITION OF cities_ab FOR VALUES FROM (10000) TO  
    (100000);
```

Создание секций таблицы, секционируемой по хешу:

```
CREATE TABLE orders_p1 PARTITION OF orders  
    FOR VALUES WITH (MODULUS 4, REMAINDER 0);  
CREATE TABLE orders_p2 PARTITION OF orders  
    FOR VALUES WITH (MODULUS 4, REMAINDER 1);  
CREATE TABLE orders_p3 PARTITION OF orders  
    FOR VALUES WITH (MODULUS 4, REMAINDER 2);  
CREATE TABLE orders_p4 PARTITION OF orders  
    FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

Создание секции по умолчанию:

```
CREATE TABLE cities_partdef  
    PARTITION OF cities DEFAULT;
```

46.7.Совместимость

Команда `CREATE TABLE` соответствует стандарту SQL, с описанными ниже исключениями.

46.7.1. Временные таблицы

Хотя синтаксис `CREATE TEMPORARY TABLE` подобен аналогичному в стандарте SQL, результат получается другим. В стандарте временные таблицы определяются только один раз и существуют (изначально пустые) в каждом сеансе, в

котором они используются. Синергия-БД вместо этого требует, чтобы каждый сеанс выполнял собственную команду `CREATE TEMPORARY TABLE` для каждой временной таблицы, которая будет использоваться. Это позволяет использовать в разных сеансах таблицы с одинаковыми именами для разных целей, тогда как при подходе, регламентированном стандартом, все экземпляры временной таблицы с одним именем должны иметь одинаковую табличную структуру.

Поведение временных таблиц, описанное в стандарте, в большинстве своем игнорируют и другие СУБД, так что в этом отношении Синергия-БД ведет себя так же, как и ряд других СУБД.

В стандарте SQL также разделяются глобальные и локальные временные таблицы – в локальной временной таблице содержится отдельный набор данных для каждого модуля SQL в отдельном сеансе, хотя её определение так же разделяется между ними. Так как в Синергия-БД модули SQL не поддерживаются, это различие в Синергия-БД не существует.

Совместимости ради, Синергия-БД принимает ключевые слова `GLOBAL` и `LOCAL` в объявлении временной таблицы, но в настоящее время они никак не действуют. Использовать их не рекомендуется, так как в будущих версиях Синергия-БД может быть принята их интерпретация, более близкая к стандарту.

Предложение `ON COMMIT` для временных таблиц тоже подобно описанному в стандарте SQL, но есть некоторые отличия. Если предложение `ON COMMIT` опущено, в SQL подразумевается поведение `ON COMMIT DELETE ROWS`. Однако в Синергия-БД по умолчанию действует `ON COMMIT PRESERVE ROWS`. Параметр `ON COMMIT DROP` в стандарте SQL отсутствует.

46.7.2. Неотложенные ограничения уникальности

Когда ограничение `UNIQUE` или `PRIMARY KEY` не является отложенным, Синергия-БД проверяет уникальность непосредственно в момент добавления или изменения строки. Стандарт SQL говорит, что уникальность должна обеспечиваться только в конце оператора; это различие проявляется, например когда одна команда изменяет множество ключевых значений. Чтобы получить поведение, оговоренное

стандартом, объявите ограничение как откладываемое (DEFERRABLE), но не отложенное (т.е., INITIALLY IMMEDIATE). Учтите, что этот вариант может быть значительно медленнее, чем немедленная проверка ограничений.

46.7.3. Ограничения-проверки для колонок

Стандарт SQL говорит, что ограничение CHECK, определяемое для колонки, может ссылаться только на колонку, с которой оно связано; только ограничения CHECK для таблиц могут ссылаться на несколько колонок. В Синергия-БД этого ограничения нет; он воспринимает ограничения-проверки для колонок и таблиц одинаково.

46.7.4. Ограничение EXCLUDE

Ограничения EXCLUDE являются расширением Синергия-БД.

46.7.5. NULL «Ограничение»

«Ограничение» NULL (на самом деле это не ограничение) является расширением Синергия-БД стандарта SQL, которое реализовано для совместимости с некоторыми другими СУБД (и для симметрии с ограничением NOT NULL). Так как это поведение по умолчанию для любой колонки, его присутствие не несет смысловой нагрузки.

46.7.6. Имена ограничений

В стандарте SQL говорится, что имена ограничений таблицы и ограничений домена должны быть уникальными в схеме, содержащей эту таблицу или домен. Однако Синергия-БД менее строга: она требует только, чтобы имена были уникальны среди ограничений, присоединённых к данной конкретной таблице или домену. Но такого послабления нет для ограничений, построенных на индексах (ограничений UNIQUE, PRIMARY KEY и EXCLUDE), так как ограничение и связанный с ним индекс имеют одно имя, а имена индексов должны быть уникальны среди всех отношений в их схеме.

В настоящее время в Синергия-БД ограничения NOT NULL вообще не имеют

имён, так что на них требования уникальности не распространяются. Однако это может поменяться в будущих выпусках.

46.7.7. Наследование

Множественное наследование посредством `INHERITS` является языковым расширением Синергия-БД. `SQL:1999` и более поздние стандарты определяют единичное наследование с другим синтаксисом и смыслом. Наследование в стиле `SQL:1999` пока ещё не поддерживается в Синергия-БД.

46.7.8. Таблицы с нулем колонок

Синергия-БД позволяет создать таблицу без колонок (например, `CREATE TABLE foo () ;`). Это расширение стандарта `SQL`, который не допускает таблицы с нулем колонок. Таблицы с нулем колонок сами по себе не очень полезны, но если их запретить, возникают странные особые ситуации с командой `ALTER TABLE DROP COLUMN`, так что лучшим вариантом кажется игнорировать это требование стандарта.

46.7.9. Множество столбцов идентификации

Синергия-БД позволяет иметь в таблице более одного столбца идентификации. В стандарте же говорится, что в таблице может быть максимум один столбец идентификации. Это ограничение ослаблено в основном для большей гибкости при выполнении изменений в схеме или миграции. Заметьте, что команда `INSERT` поддерживает только одно предложение переопределения значения, применяемое ко всему оператору, так что с несколькими столбцами идентификации разное поведение не поддерживается должным образом.

46.7.10. Предложение LIKE

Хотя предложение `LIKE` описано в стандарте `SQL`, многие варианты его использования, допустимые в Синергия-БД, в стандарте не описаны, а некоторые предусмотренные в стандарте возможности не реализованы в Синергия-БД.

46.7.11. Предложение WITH

Предложение WITH является расширением Синергия-БД; в стандарте ни параметры хранения, ни OID не оговариваются.

46.7.12. Табличные пространства

Концепция табличных пространств в Синергия-БД отсутствует в стандарте. Как следствие, предложения TABLESPACE и USING INDEX TABLESPACE являются расширениями.

46.7.13. Типизированные таблицы

Типизированные таблицы реализуют подмножество стандарта SQL. Согласно стандарту, типизированная таблица содержит колонки, соответствующие нижележащему составному типу, и ещё одну колонку, ссылающуюся на себя. Синергия-БД не поддерживает ссылающиеся на себя колонки явно, но тот же эффект можно получить, воспользовавшись OID.

46.7.14. Предложение PARTITION BY

Предложение PARTITION BY является расширением Синергия-БД.

46.7.15. Предложение PARTITION OF

Предложение PARTITION OF является расширением Синергия-БД.

47. CREATE TABLE AS

CREATE TABLE AS – создать таблицу из результатов запроса.

47.1.Синтаксис

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } |  
UNLOGGED ] TABLE [ IF NOT EXISTS ] имя_таблицы  
    [ (имя_столбца [, ...] ) ]  
    [ WITH ( параметр_хранения [= значение] [, ...] ) |  
WITH OIDS | WITHOUT OIDS ]  
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]  
    [ TABLESPACE табл_пространство ]
```


AS запрос

[WITH [NO] DATA]

47.2.Описание

CREATE TABLE AS создает таблицу и наполняет её данными, полученными в результате выполнения SELECT. Колонки этой таблицы получают имена и типы данных в соответствии с колонками результата SELECT (хотя имена колонок можно переопределить, добавив явно список новых имен колонок).

CREATE TABLE AS напоминает создание представления, но на самом деле есть значительная разница: эта команда создает новую таблицу и выполняет запрос только раз, чтобы наполнить таблицу начальными данными. Последующие изменения в исходных таблицах запроса в новой таблице отражаться не будут. С представлением, напротив, определяющая его команда SELECT выполняется при каждой выборке из него.

47.3.Параметры

GLOBAL или LOCAL

Для совместимости игнорируются. Использование этих ключевых слов считается устаревшим.

TEMPORARY или TEMP

Если указано, создаваемая таблица будет временной.

UNLOGGED

Если указано, создаваемая таблица будет нежурналируемой.

IF NOT EXISTS

Не считать ошибкой, если отношение с таким именем уже существует. В этом случае будет выдано замечание.

имя_таблицы

Имя создаваемой таблицы (возможно, дополненное схемой).

имя_колонки

Имя колонки в создаваемой таблице. Если имена колонок не заданы явно, они определяются по именам колонок результата запроса.

WITH (параметр_хранения [= значение] [, ...])

Это предложение определяет дополнительные параметры хранения для новой таблицы. Предложение WITH может также включать указание OIDS=TRUE (или просто OIDS), с которым строкам в новой таблице будут назначаться идентификаторы объектов (OID), либо указание OIDS=FALSE, с которым строки не будут содержать OID.

WITH OIDS

WITHOUT OIDS

Это устаревшее написание указаний WITH (OIDS) и WITH (OIDS=FALSE), соответственно. Если требуется определить одновременно свойство OIDS и параметры хранения, необходимо использовать синтаксис WITH (...).

ON COMMIT

Поведением временных таблиц в конце блока транзакции позволяет управлять предложение ON COMMIT, которое принимает три параметра:

PRESERVE ROWS

Никакое специальное действие в конце транзакции не выполняется. Это поведение по умолчанию.

DELETE ROWS

Все строки в этой временной таблице будут удаляться в конце каждого блока транзакции. По сути, при каждой фиксации транзакции будет автоматически выполняться TRUNCATE.

DROP

Эта временная таблица будет удаляться в конце текущего блока транзакции.

TABLESPACE табл_пространство

Здесь табл_пространство – имя табличного пространства, в котором будет создаваться новая таблица. Если оно не указано, выбирается default_tablespace или temp_tablespaces, если таблица временная.

запрос

Команда SELECT, TABLE или VALUES, либо команда EXECUTE, выполняющая подготовленный запрос SELECT, TABLE или VALUES.

WITH [NO] DATA

Это предложение определяет, будут ли данные, выданные запросом, копироваться в новую таблицу. Если нет, то копируется только структура. По умолчанию данные копируются.

47.4.Замечания

Функциональность этой команды подобна `SELECT INTO`, но предпочтительнее использовать ее, во избежание путаницы с другими применениями синтаксиса `SELECT INTO`. Кроме того, набор возможностей `CREATE TABLE AS` шире, чем у `SELECT INTO`.

Команда `CREATE TABLE AS` позволяет пользователю явно определить, добавлять ли `OID` в таблицу. Если присутствие `OID` не определено явно, оно определяется конфигурационной переменной `default_with_oids`.

47.5.Примеры

Создание таблицы `films_recent`, содержащей только последние записи из таблицы `films`:

```
CREATE TABLE films_recent AS
    SELECT * FROM films WHERE date_prod >= '2002-01-01';
```

Чтобы скопировать таблицу полностью, можно также использовать короткую форму команды `TABLE`:

```
CREATE TABLE films2 AS
    TABLE films;
```

Создание временной таблицы `films_recent`, содержащей только последние записи таблицы `films`, с применением подготовленного оператора. Новая таблица будет содержать `OID` и прекратит существование при фиксации транзакции:

```
PREPARE recentfilms(date) AS
    SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_recent WITH (oids) ON COMMIT
DROP AS
    EXECUTE recentfilms('2002-01-01');
```

47.6.Совместимость

CREATE TABLE AS соответствует стандарту SQL. Нестандартные расширения перечислены ниже:

1. Стандарт требует заключать предложение подзапроса в скобки, но в Синергия-БД эти скобки необязательны.
2. Стандарт требует присутствия указания WITH [NO] DATA, в Синергия-БД оно необязательно.
3. Синергия-БД работает с временными таблицами не так, как описано в стандарте.
4. Предложение WITH является расширением Синергия-БД; в стандарте ни параметры хранения, ни OID не оговариваются.
5. Концепция табличных пространств в Синергия-БД отсутствует в стандарте. Как следствие, предложение TABLESPACE является расширением.

48. CREATE TABLESPACE

CREATE TABLESPACE – создать табличное пространство.

48.1.Синтаксис

```
CREATE TABLESPACE табл_пространство
    [ OWNER { новый_владелец | CURRENT_USER |
SESSION_USER } ]
    LOCATION 'каталог'
    [ WITH ( параметр_табличного_пространства = значение
[, ... ] ) ]
```

48.2.Описание

CREATE TABLESPACE регистрирует новое табличное пространство на уровне кластера баз данных. Имя табличного пространства должно отличаться от имен уже существующих табличных пространств в кластере.

Табличные пространства позволяют суперпользователям определять альтернативные расположения в файловой системе, где могут находиться файлы, содержащие объекты базы данных (например, таблицы или индексы).

Пользователь, имеющий соответствующие права, может передать параметр `табл_пространство` команде `CREATE DATABASE`, `CREATE TABLE`, `CREATE INDEX` или `ADD CONSTRAINT`, чтобы файлы данных для этих объектов хранились в указанном табличном пространстве

Примечание. Табличное пространство нельзя использовать отдельно от кластера, в котором оно было определено.

48.3.Параметры

`табл_пространство`

Имя создаваемого табличного пространства. Это имя не может начинаться с `pg_`, так как такие имена зарезервированы для системных табличных пространств.

`имя_пользователя`

Имя пользователя, который будет владельцем табличного пространства. Если опущено, владельцем по умолчанию станет пользователь, выполняющий команду. Создавать пространства имен могут только суперпользователи, но их владельцами могут быть назначены и обычные пользователи.

`каталог`

Каталог, который будет использован для этого табличного пространства. Этот каталог должен быть пустым и должен принадлежать системному пользователю Синергия-БД. Задаваться его расположение должно абсолютным путем.

`параметр_табличного_пространства`

Устанавливаемый или сбрасываемый параметр табличного пространства. В настоящее время поддерживаются только параметры `seq_page_cost`, `random_page_cost` и `effective_io_concurrency`. При установке этих значений для заданного табличного пространства будет переопределена обычная оценка стоимости чтения страниц из таблиц в этом пространстве, настраиваемая одноимённым параметром конфигурации. Это может быть полезно, если одно из табличных пространств размещено на диске, который быстрее или медленнее остальной дисковой системы.

48.4.Замечания

Табличные пространства поддерживаются только на платформах, поддерживающих символические ссылки.

CREATE TABLESPACE не может быть выполнена внутри блока транзакции.

48.5.Примеры

Создание табличного пространства dbspace в /data/dbs:

```
CREATE TABLESPACE dbspace LOCATION '/data/dbs';
```

Создание табличного пространства indexspace в /data/indexes и назначение его владельцем genevieve:

```
CREATE TABLESPACE indexspace OWNER genevieve LOCATION  
'/data/indexes';
```

48.6.Совместимость

CREATE TABLESPACE является расширением Синергия-БД.

49. CREATE TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH CONFIGURATION – создать конфигурацию текстового поиска.

49.1.Синтаксис

```
CREATE TEXT SEARCH CONFIGURATION имя (  
    PARSER = имя_анализатора |  
    COPY = исходная_конфигурация  
)
```

49.2.Описание

CREATE TEXT SEARCH CONFIGURATION создает конфигурацию текстового поиска. Конфигурация текстового поиска определяет анализатор текстового поиска, разделяющий строку на фрагменты, и словари, позволяющие установить, какие именно фрагменты представляют интерес при поиске.

Если задается только анализатор, новая конфигурация текстового поиска не будет содержать сопоставления типов фрагментов со словарями и, как следствие,

будет игнорировать все слова. Для создания сопоставлений, которые бы сделали конфигурацию полезной, затем нужно будет воспользоваться командой ALTER TEXT SEARCH CONFIGURATION. Другой вариант команды позволяет скопировать конфигурацию текстового поиска.

Если указывается имя схемы, конфигурация текстового поиска создается в указанной схеме. В противном случае она создается в текущей схеме.

Владельцем конфигурации текстового поиска становится пользователь её создавший.

49.3.Параметры

имя

Имя создаваемой конфигурации текстового поиска, возможно, дополненное схемой.

имя_анализатора

Имя анализатора текстового поиска, который будет использоваться этой конфигурацией.

исходная_конфигурация

Имя существующей конфигурации текстового поиска, которая будет скопирована.

49.4.Замечания

Параметры PARSER и COPY являются взаимоисключающими, так как при копировании существующей конфигурации выбранный в ней анализатор копируется тоже.

49.5.Совместимость

Оператор CREATE TEXT SEARCH CONFIGURATION отсутствует в стандарте SQL.

50. CREATE TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH DICTIONARY – создать словарь текстового поиска.

50.1.Синтаксис

```
CREATE TEXT SEARCH DICTIONARY имя (  
    TEMPLATE = шаблон  
    [, параметр = значение [, ... ]]  
)
```

50.2.Описание

CREATE TEXT SEARCH DICTIONARY создает словарь текстового поиска. Словарь текстового поиска определяет способ выделения слов, представляющих или не представляющих интерес для поиска. Работа словаря зависит от шаблона текстового поиска, в котором определяются функции, собственно выполняющие действия. Обычно словарь задает некоторые параметры, управляющие частным поведением функций шаблона.

Если указывается имя схемы, словарь текстового поиска создается в указанной схеме. В противном случае он создается в текущей схеме.

Владельцем словаря текстового поиска становится пользователь его создавший.

50.3.Параметры

имя

Имя создаваемого словаря текстового поиска, возможно, дополненное схемой.

шаблон

Имя шаблона текстового поиска, который будет определять общее поведение этого словаря.

параметр

Имя параметра шаблона, устанавливаемого для данного словаря.

значение

Значение для параметра, связанного с шаблоном. Если это не простой идентификатор или число, его следует заключить в кавычки (при желании его можно заключать в кавычки всегда).

Параметры могут перечисляться в любом порядке.

50.4.Примеры

Команда в следующем примере создает словарь на базе Snowball с нестандартным списком стоп-слов.

```
CREATE TEXT SEARCH DICTIONARY my_russian (  
    template = snowball,  
    language = russian,  
    stopwords = myrussian  
);
```

50.5.Совместимость

Оператор CREATE TEXT SEARCH DICTIONARY отсутствует в стандарте SQL.

51. CREATE TEXT SEARCH PARSER

CREATE TEXT SEARCH PARSER – создать анализатор текстового поиска

51.1.Синтаксис

```
CREATE TEXT SEARCH PARSER имя (  
    START = функция_начала ,  
    GETTOKEN = функция_выдачи_фрагмента ,  
    END = функция_окончания ,  
    LEXTYPES = функция_лекс_типов  
    [, HEADLINE = функция_выдержек ]  
)
```

51.2.Описание

CREATE TEXT SEARCH PARSER создает анализатор текстового поиска. Анализатор текстового поиска определяет способ разделения текстовой строки на фрагменты и назначения типов (категорий) этим фрагментам. Анализатор не очень полезен сам по себе, для осуществления поиска он должен быть подключен к конфигурации текстового поиска вместе с определенными словарями.

Если указывается имя схемы, словарь текстового поиска создается в указанной схеме. В противном случае он создается в текущей схеме.

Выполнить CREATE TEXT SEARCH PARSER может только суперпользователь. Это ограничение введено потому, что ошибочное определение анализатора текстового поиска может вызвать нарушения или даже сбой в работе сервера.

51.3.Параметры

имя

Имя создаваемого анализатора текстового поиска, возможно, дополненное схемой.

функция_начала

Имя функции, вызываемой в начале обработки.

функция_выдачи_фрагмента

Имя функции, выдающей следующий фрагмент.

функция_окончания

Имя функции, вызываемой по окончании обработки.

функция_лекс_типов

Имя функции перечисления лексических типов (эта функция выдает информацию о множестве типов фрагментов, выделяемых анализатором).

функция_выдержек

Имя функции извлечения выдержек (эта функция выделяет краткое содержание для набора фрагментов).

Имена функций могут быть дополнены именем схемы, если требуется. Типы аргументов не указываются, так как список аргументов для всех типов функций предопределен. Обязательными являются все функции, кроме функции выдержек.

Аргументы могут перечисляться в любом порядке, не только в том, что показан выше.

51.4.Совместимость

Оператор CREATE TEXT SEARCH PARSER отсутствует в стандарте SQL.

52. CREATE TEXT SEARCH TEMPLATE

CREATE TEXT SEARCH TEMPLATE – создать шаблон текстового поиска.

52.1.Синтаксис

```
CREATE TEXT SEARCH TEMPLATE имя (  
    [ INIT = функция_инициализации , ]  
    LEXIZE = функция_выделения_лексем  
)
```

52.2.Описание

CREATE TEXT SEARCH TEMPLATE создает новый шаблон текстового поиска. Шаблоны текстового поиска определяют функции для реализации словарей текстового поиска. Шаблон сам по себе бесполезен, но его экземпляр нужно создать для словаря. Словарь обычно определяет параметры, передаваемые функциям шаблона.

Если указывается имя схемы, шаблон текстового поиска создается в указанной схеме. В противном случае он создается в текущей схеме.

Выполнить CREATE TEXT SEARCH TEMPLATE может только суперпользователь. Это ограничение введено потому, что ошибочное определение шаблона текстового поиска может вызвать нарушения или даже сбой в работе сервера. Смысл отделения шаблонов от словарей в том, что шаблон покрывает «небезопасные» аспекты определения словаря. Параметры, которые можно задать при определении словаря, не могут причинить вред, поэтому создавать словари разрешено и непривилегированным пользователям.

52.3.Параметры

имя

Имя создаваемого шаблона текстового поиска, возможно, дополненное схемой.

функция_инициализации

Имя функции инициализации для шаблона.

функция_выделения_лексем

Имя функции выделения лексем.

Имена функций могут быть дополнены именем схемы, если требуется. Типы аргументов не указываются, так как список аргументов для всех типов функций

предопределен. Функция выделения лексем является обязательной, а функция инициализации может отсутствовать.

Аргументы могут перечисляться в любом порядке, не только в том, что показан выше.

52.4. Совместимость

Оператор `CREATE TEXT SEARCH TEMPLATE` отсутствует в стандарте SQL.

53. CREATE TRANSFORM

`CREATE TRANSFORM` – создать трансформацию.

53.1. Синтаксис

```
CREATE [ OR REPLACE ] TRANSFORM FOR имя_типа LANGUAGE
имя_языка (
    FROM SQL WITH FUNCTION имя_функции_из_sql
    [ (тип_аргумента [, ...]) ],
    TO SQL WITH FUNCTION имя_функции_в_sql
    [ (тип_аргумента [, ...]) ]
);
```

53.2. Описание

`CREATE TRANSFORM` определяет новую трансформацию. `CREATE OR REPLACE TRANSFORM` либо создаёт трансформацию, либо заменяет существующую.

Трансформация определяет, как преобразовать тип данных для процедурного языка. Например, если написать на языке PL/Python функцию, использующую тип `hstore`, PL/Python заведомо не знает, как должны представляться значения `hstore` в среде Python. Обычно реализации языка нисходят к текстовому представлению, но это может быть неудобно, когда более уместен был бы, например, ассоциативный массив или список.

Трансформация определяет две функции:

1. Функция «из SQL» преобразует тип из среды SQL в среду языка. Эта функция будет вызываться для аргументов функции, написанной на этом

языке.

2. Функция «в SQL» преобразует тип из среды языка в среду SQL. Эта функция будет вызываться для значения, возвращаемого из функции на этом языке.

Предоставлять обе эти функции не требуется, можно ограничиться одной. Если одна из них не указана, при необходимости выбирается поведение, принятое для языка по умолчанию. Чтобы полностью перекрыть путь трансформации в одну сторону, можно написать функцию, которая будет всегда выдавать ошибку.

Чтобы создать трансформацию, необходимо быть владельцем и иметь право USAGE для типа, иметь право USAGE для языка, а также быть владельцем и иметь право EXECUTE для функций из-SQL и в-SQL, если они задаются.

53.3.Параметры

имя_типа

Имя типа данных, для которого предназначена трансформация.

имя_языка

Имя языка, для которого предназначена трансформация.

имя_функции_из_sql [(тип_аргумента [, ...])]

Имя функции для преобразования типа из среды SQL в среду языка. Она должна принимать один аргумент типа `internal` и возвращать тип `internal`. Фактический аргумент будет иметь тип, заданный для трансформации, и сама функция должна рассчитывать на это. Но на уровне SQL нельзя объявить функцию, возвращающую тип `internal`, если она не принимает минимум один аргумент типа `internal`. Фактически возвращаемое значение будет определяться реализацией языка. Если список аргументов отсутствует, имя функции должно быть уникальным в её схеме.

имя_функции_в_sql [(тип_аргумента [, ...])]

Имя функции для преобразования типа из среды языка в среду SQL. Она должна принимать один аргумент типа `internal` и возвращать тип, для которого создаётся трансформация. Фактическое значение аргумента будет определяться

реализацией языка. Если список аргументов отсутствует, имя функции должно быть уникальным в её схеме.

53.4.Замечания

Для удаления трансформаций применяется `DROP TRANSFORM`.

53.5.Примеры

Чтобы создать трансформацию для типа `hstore` и языка `plpythonu`, сначала нужно создать тип и язык:

```
CREATE TYPE hstore ...;
```

```
CREATE EXTENSION plpythonu;
```

Затем создайте необходимые функции:

```
CREATE FUNCTION hstore_to_plpython(val internal) RETURNS  
internal
```

```
LANGUAGE C STRICT IMMUTABLE
```

```
AS ...;
```

```
CREATE FUNCTION plpython_to_hstore(val internal) RETURNS  
hstore
```

```
LANGUAGE C STRICT IMMUTABLE
```

```
AS ...;
```

И наконец, создайте трансформацию, соединяющую всё это вместе:

```
CREATE TRANSFORM FOR hstore LANGUAGE plpythonu (  
    FROM SQL WITH FUNCTION hstore_to_plpython(internal),  
    TO SQL WITH FUNCTION plpython_to_hstore(internal)  
);
```

На практике эти команды помещаются в расширение.

В разделе `contrib` представлено несколько расширений, в которых определены трансформации, что может послужить практическим примером реализации.

53.6.Совместимость

Первая форма CREATE TRANSFORM является расширением Синергия-БД. В стандарте SQL есть команда CREATE TRANSFORM, но её предназначение – преобразовывать типы для языков на стороне клиента. Этот вариант использования не поддерживается Синергия-БД.

54. CREATE TRIGGER

CREATE TRIGGER – создать триггер.

54.1.Синтаксис

```
CREATE [ CONSTRAINT ] TRIGGER имя { BEFORE | AFTER |  
INSTEAD OF } { событие [ OR ... ] }  
ON имя_таблицы  
[ FROM ссылающаяся_таблица ]  
[ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY  
IMMEDIATE | INITIALLY DEFERRED ] ]  
[ REFERENCING { { OLD | NEW } TABLE [ AS ]  
имя_переходного_отношения } [ ... ] ]  
[ FOR [ EACH ] { ROW | STATEMENT } ]  
[ WHEN ( условие ) ]  
EXECUTE { FUNCTION | PROCEDURE } имя_функции  
( аргументы )
```

Здесь допускается событие:

```
INSERT  
UPDATE [ OF имя_колонки [, ... ] ]  
DELETE  
TRUNCATE
```

54.2.Описание

CREATE TRIGGER создает новый триггер. Триггер будет связан с указанной таблицей, представлением или сторонней таблицей и будет выполнять заданную функцию имя_функции при определенных событиях.

Триггер можно настроить так, чтобы он срабатывал до операции со строкой

(до проверки ограничений и попытки выполнить INSERT, UPDATE или DELETE) или после её завершения (после проверки ограничений и выполнения INSERT, UPDATE или DELETE), либо вместо операции (при добавлении, изменении и удалении строк в представлении). Если триггер срабатывает до или вместо события, он может пропустить операцию с текущей строкой, либо изменить добавляемую строку (только для операций INSERT и UPDATE). Если триггер срабатывает после события, он «видит» все изменения, включая результат действия других триггеров.

Триггер с пометкой FOR EACH ROW вызывается один раз для каждой строки, изменяемой в процессе операции. Например, операция DELETE, удаляющая 10 строк, приведет к срабатыванию всех триггеров ON DELETE в целевом отношении 10 раз подряд, по одному разу для каждой удаляемой строки. Триггер с пометкой FOR EACH STATEMENT, напротив, вызывается только один раз для конкретной операции, вне зависимости от того, как много строк она изменила (в частности, при выполнении операции, изменяющей ноль строк, все равно будут вызваны все триггеры FOR EACH STATEMENT).

Триггеры, срабатывающие в режиме INSTEAD OF, должны быть помечены FOR EACH ROW и могут быть определены только для представлений. Триггеры BEFORE и AFTER для представлений должны быть помечены FOR EACH STATEMENT.

Кроме того, триггеры можно определить и для команды TRUNCATE, но только типа FOR EACH STATEMENT.

В таблице 54.1 перечисляются типы триггеров, которые могут использоваться для таблиц, представлений и сторонних таблиц.

Таблица 54.1 – Типы триггеров

Когда	Событие	На уровне строк	На уровне оператора
BEFORE	INSERT/UPDATE/DELETE	Таблицы и сторонние таблицы	Таблицы, представления и сторонние таблицы
	TRUNCATE	—	Таблицы
AFTER	INSERT/UPDATE/DELETE	Таблицы и сторонние таблицы	Таблицы, представления и сторонние таблицы
	TRUNCATE	—	Таблицы
INSTEAD OF	INSERT/UPDATE/DELETE	Представления	—
	TRUNCATE	—	—

Кроме того, в определении триггера можно указать булевское условие WHEN,

которое определит, вызывать триггер или нет. В триггерах на уровне строк условия WHEN могут проверять старые и/или новые значения колонок в строке. Триггеры на уровне оператора так же могут содержать условие WHEN, хотя для них это не столь полезно, так как в этом условии нельзя сослаться на какие-либо значения в таблице.

Если для одного события определено несколько триггеров одного типа, они будут срабатывать в алфавитном порядке их имен.

Когда указывается REFERENCING, для триггера собираются *переходные отношения*, представляющие собой множества строк, включающие все строки, которые были добавлены, удалены или изменены текущим оператором SQL. Это позволяет триггеру наблюдать общую картину того, что сделал оператор, а не только одну строку за другой. Это указание допускается только для триггера AFTER, не являющегося триггером ограничения; кроме того, если это триггер для UPDATE, у него должен отсутствовать список имён_столбцов. Указание OLD TABLE может быть задано только один раз и только для триггера, который может срабатывать при UPDATE или DELETE; оно создаёт переходное отношение, содержащее *образы-до-изменения* всех строк, модифицированных или удалённых оператором. Указание NEW TABLE, подобным образом, может быть задано только единожды и только для триггера, который может срабатывать для UPDATE или INSERT; оно создаёт переходное отношение, содержащее *образы-после-изменения* всех строк, модифицированных или добавленных оператором.

SELECT не изменяет никакие строки, поэтому создавать триггеры для SELECT нельзя. Для решения задач, в которых требуются подобные триггеры, могут подойти правила или представления.

54.3.Параметры

ИМЯ

Имя, назначаемое новому триггеру. Это имя должно отличаться от имени любого другого триггера в этой же таблице. Имя не может быть дополнено схемой – триггер наследует схему от своей таблицы. Для триггеров ограничений это имя также используется, когда требуется скорректировать поведение триггера с помощью

команды SET CONSTRAINTS.

BEFORE
AFTER
INSTEAD OF

Определяет, будет ли заданная функция вызываться до, после или вместо события. Для триггера ограничения можно указать только AFTER.

событие

Принимает одно из значений: INSERT, UPDATE, DELETE или TRUNCATE; этот параметр определяет событие, при котором будет срабатывать триггер. Несколько событий можно указать, добавив между ними слово OR.

Для событий UPDATE можно указать список колонок, используя такую запись:

UPDATE OF имя_колонки1 [, имя_колонки2 ...]

Такой триггер сработает, только если в указанном в целевой команде UPDATE списке колонок найдется минимум одна из перечисленных.

Для событий INSTEAD OF UPDATE список колонок задать нельзя.

table_name

Имя (возможно, дополненное схемой) таблицы, представления или сторонней таблицы, для которых предназначен триггер.

ссылающаяся_таблица

Имя (возможно, дополненное схемой) другой таблицы, на которую ссылается ограничение. Оно используется для ограничений внешнего ключа и не рекомендуется для обычного применения. Это указание допускается только для триггеров ограничений.

DEFERRABLE
NOT DEFERRABLE
INITIALLY IMMEDIATE
INITIALLY DEFERRED

Время срабатывания триггера по умолчанию. Это указание допускается только для триггеров ограничений.

REFERENCING

Это ключевое слово непосредственно предшествует объявлению одного или двух имён, по которым можно будет обращаться к переходным отношениями,

образуемым при выполнении целевого оператора.

OLD TABLE
NEW TABLE

Это предложение указывает, будет ли следующее имя относиться к переходному отношению с образом-до-изменения или к переходному отношению с образом-после-изменения.

имя_переходного_отношения

Имя (неполное, без схемы), которое будет использоваться в триггере для обращения к этому переходному отношению.

FOR EACH ROW
FOR EACH STATEMENT

Определяет, будет ли процедура триггера срабатывать один раз для каждой строки, либо для SQL-оператора. Если не указано ничего, подразумевается FOR EACH STATEMENT (для оператора). Для триггеров ограничений можно указать только FOR EACH ROW.

условие

Булевское выражение, определяющее, будет ли выполняться функция триггера. Если для триггера задано указание WHEN, функция будет вызываться, только когда условие возвращает true. В триггерах FOR EACH ROW условие WHEN может ссылаться на значения колонки в старой и/или новой строке, в виде OLD.имя_колонки и NEW.имя_колонки, соответственно. Разумеется, триггеры INSERT не могут ссылаться на OLD, а триггеры DELETE не могут ссылаться на NEW.

Триггеры INSTEAD OF не поддерживают условия WHEN.

В настоящее время выражения WHEN не могут содержать подзапросы.

Учтите, что для триггеров ограничений вычисление условия WHEN не откладывается, а выполняется немедленно после операции, изменяющей строки. Если результат условия – ложь, сам триггер не откладывается для последующего выполнения.

имя_функции

Заданная пользователем функция, объявленная как функция без аргументов и возвращающая тип trigger, которая будет вызываться при срабатывании триггера.

В синтаксисе `CREATE TRIGGER` ключевые слова `FUNCTION` и `PROCEDURE` равнозначны, но указываемая триггерная функция должна в любом случае быть функцией, а не процедурой. Ключевое слово `PROCEDURE` здесь поддерживается по историческим причинам и считается устаревшим.

аргументы

Необязательный список аргументов через запятую, которые будут переданы функции при срабатывании триггера. В качестве аргументов функции передаются строковые константы. И хотя в этом списке можно записать и простые имена или числовые константы, они тоже будут преобразованы в строки. Порядок обращения к таким аргументам в функции триггера может отличаться от обычных аргументов, поэтому его следует уточнить в описании языка реализации этой функции.

54.4.Замечания

Чтобы создать триггер, пользователь должен иметь право `TRIGGER` для этой таблицы. Также пользователь должен иметь право `EXECUTE` для триггерной функции.

Для удаления триггера применяется команда `DROP TRIGGER`.

Триггер для избранных колонок (определенный с помощью `UPDATE OF имя_колонки`) будет срабатывать, когда его колонки перечислены в качестве целевых в списке `SET` команды `UPDATE`. Изменения, вносимые в строки триггерами `BEFORE UPDATE` при этом не учитываются, поэтому значения колонок можно изменить так, что триггер не сработает. И наоборот, при выполнении команды `UPDATE ... SET x = x ...` триггер для колонки `x` сработает, хотя значение колонки не меняется.

В триггере `BEFORE` условие `WHEN` вычисляется непосредственно перед возможным вызовом функции, поэтому проверка `WHEN` существенно не отличается от проверки того же условия в начале функции триггера. В частности, учтите, что строка `NEW`, которую видит ограничение, содержит текущие значения, возможно, измененные предыдущими триггерами. Кроме того, в триггере `BEFORE` условие `WHEN` не может проверять системные колонки в строке `NEW` (например, `oid`), так как

они ещё не установлены.

В триггере AFTER условие WHEN проверяется сразу после изменения строки, и если оно выполняется, событие запоминается, чтобы вызвать триггер в конце оператора. Если же для триггера AFTER условие WHEN не выполняется, нет необходимости запоминать событие для последующей обработки или заново перечитывать строку в конце оператора. Это приводит к значительному ускорению операторов, изменяющих множество строк, когда триггер должен срабатывать только для некоторых из них.

В некоторых случаях одна команда SQL может вызывать сразу нескольких видов триггеров. Например, INSERT с предложением ON CONFLICT DO UPDATE может выполнять операции как добавления, так и изменения, так что она при необходимости будет вызывать триггеры обоих видов. При этом переходные отношения, предоставляемые триггерам, будут разными в зависимости от типа события; то есть триггер INSERT будет видеть только добавленные строки, а триггер UPDATE – только изменённые.

Изменения или удаления строк, вызванные действиями по обеспечению целостности внешнего ключа, например, ON UPDATE CASCADE или ON DELETE SET NULL, считаются частью SQL-команды, вызвавшей эти действия (заметьте, что такие действия не могут быть отложенными). В затрагиваемой таблице будут вызваны соответствующие триггеры, и таким образом появляется возможность вызова триггеров для SQL-команды, не соответствующей непосредственно их типу. В простых ситуациях триггеры, запрашивающие переходные отношения, будут видеть все изменения, произведённые в их таблице одной исходной командой SQL, в виде одного переходного отношения. Однако возможны случаи, в которых присутствие триггера AFTER ROW, запрашивающего переходные отношения, приведёт к тому, что операции для обеспечения целостности внешнего ключа, вызванные одной SQL-командой, будут разделены на несколько этапов, и на каждом будут свои переходные отношения. В таких случаях все существующие триггеры уровня оператора будут срабатывать единожды при создании переходного отношения, что гарантирует, что эти триггеры будут видеть каждую

обрабатываемую строку в переходном отношении один и только один раз.

Триггеры уровня операторов для представления срабатывают, только если операция с представлением обрабатывается триггером уровня строк `INSTEAD OF`. Если операция обрабатывается правилом `INSTEAD`, то вместо исходного оператора, обращающегося к представлению, выполняются те операторы, что генерирует правило, поэтому вызываться будут триггеры, связанные с таблицами, к которым обращаются эти заменяющие операторы. Аналогично, для автоматически изменяемого представления выполнение операции сводится к переписыванию оператора в виде операции с базовой таблицей представления, так что срабатывать будут триггеры уровня операторов для базовой таблицы.

При создании триггера уровня строки для секционированной таблицы такие же триггеры будут созданы во всех существующих секциях этой таблицы; идентичные триггеры будут установлены и в секциях, создаваемых и присоединяемых позже. Для секционированных таблиц могут создаваться только триггеры `AFTER`.

При изменении данных в секционированной таблице или таблице с потомками срабатывают триггеры уровня оператора, связанные с явно задействованной таблицей, но не триггеры уровня оператора для её секций или дочерних таблиц. Триггеры уровня строк, напротив, срабатывают для строк в затрагиваемых секциях или дочерних таблицах, даже если они явно не присутствуют в запросе. Если триггер уровня оператора был определён с переходными отношениями, названными в указании `REFERENCING`, то в них будут видны образы строк из всех затронутых секций или дочерних таблиц. В случае с потомками в иерархии наследования образы строк будут содержать только столбцы, присутствующие в таблице, с которой связан триггер. В настоящее время триггеры уровня строк с переходными отношениями нельзя определить для секций или дочерних таблиц в иерархии наследования.

54.5.Примеры

Выполнение функции `check_account_update` перед любым изменением

строк в таблице accounts:

```
CREATE TRIGGER check_update
  BEFORE UPDATE ON accounts
  FOR EACH ROW
  EXECUTE FUNCTION check_account_update();
```

То же самое, но функция триггера будет выполняться, только если столбец balance присутствует в списке целевых столбцов команды UPDATE:

```
CREATE TRIGGER check_update
  BEFORE UPDATE OF balance ON accounts
  FOR EACH ROW
  EXECUTE FUNCTION check_account_update();
```

В этом примере функция будет выполняться, если значение столбца balance в действительности изменилось:

```
CREATE TRIGGER check_update
  BEFORE UPDATE ON accounts
  FOR EACH ROW
  WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
  EXECUTE FUNCTION check_account_update();
```

Вызов функции, ведущей журнал изменений в accounts, но только если что-то изменилось:

```
CREATE TRIGGER log_update
  AFTER UPDATE ON accounts
  FOR EACH ROW
  WHEN (OLD.* IS DISTINCT FROM NEW.*)
  EXECUTE FUNCTION log_account_update();
```

Выполнение для каждой строки функции view_insert_row, которая будет вставлять строки в нижележащие таблицы представления:

```
CREATE TRIGGER view_insert
  INSTEAD OF INSERT ON my_view
  FOR EACH ROW
  EXECUTE FUNCTION view_insert_row();
```

Выполнение функции check_transfer_balances_to_zero для каждого

оператора, проверяющей, что строки transfer в совокупности дают нулевой баланс:

```
CREATE TRIGGER transfer_insert
  AFTER INSERT ON transfer
  REFERENCING NEW TABLE AS inserted
  FOR EACH STATEMENT
  EXECUTE FUNCTION check_transfer_balances_to_zero();
```

Выполнение функции check_matching_pairs для каждой строки, проверяющей, что соответствующие пары пунктов изменены синхронно (одним оператором):

```
CREATE TRIGGER paired_items_update
  AFTER UPDATE ON paired_items
  REFERENCING NEW TABLE AS newtab OLD TABLE AS oldtab
  FOR EACH ROW
  EXECUTE FUNCTION check_matching_pairs();
```

54.6.Совместимость

Оператор CREATE TRIGGER в Синергия-БД реализует подмножество возможностей, описанных в стандарте SQL. В настоящее время в нем отсутствует следующая функциональность:

1. Тогда как имена переходных таблиц для триггеров AFTER задаются предложением REFERENCING стандартным образом, переменные строк, применяемые в триггерах FOR EACH ROW нельзя объявлять в предложении REFERENCING. Порядок обращения к таким строкам зависит от языка, на котором написана триггерная функция, но для каждого языка он вполне определённый. Некоторые языки по сути действуют так, как будто в команде присутствует предложение REFERENCING с указанием OLD ROW AS OLD NEW ROW AS NEW.
2. Стандарт позволяет использовать переходные таблицы с триггерами UPDATE, ограничивающими набор отслеживаемых столбцов, но тогда и набор строк, видимых в переходных таблицах, должен зависеть от списка целевых столбцов триггера. В настоящее время такое поведение в Синергия-БД не

реализовано.

3. Синергия-БД позволяет задать в качестве действия триггера только функцию, определенную пользователем. Стандарт допускает также выполнение ряда других команд SQL, например, CREATE TABLE. Однако это ограничение несложно преодолеть, создав пользовательскую функцию, выполняющую требуемые команды.

В стандарте SQL определено, что несколько триггеров должны срабатывать по порядку создания. Синергия-БД упорядочивает их по именам, так как это было признано более удобным.

В стандарте SQL определено, что триггеры BEFORE DELETE при каскадном удалении срабатывают после завершения каскадного DELETE. В Синергия-БД триггеры BEFORE DELETE всегда срабатывают перед операцией удаления, даже если она каскадная. Это поведение выбрано как более логичное. Ещё одно отклонение от стандарта проявляется, когда триггеры BEFORE, срабатывающие в результате ссылочной операции, изменяют строки или не дают выполнить изменение. Это может привести к нарушению ограничений или сохранению данных, не соблюдающих ссылочную целостность.

Возможность задать несколько действий для одного триггера с помощью ключевого слова OR – реализованное в Синергия-БД расширение стандарта SQL.

Возможность вызывать триггеры для TRUNCATE – реализованное в Синергия-БД расширение стандарта SQL, как и возможность определять триггеры на уровне оператора для представлений.

CREATE CONSTRAINT TRIGGER – реализованное в Синергия-БД расширение стандарта SQL.

55. CREATE TYPE

CREATE TYPE – создать новый тип данных.

55.1. Синтаксис

CREATE TYPE имя AS

([имя_атрибута тип_данных [COLLATE

```
правило_сортировки ] [, ... ] ] )
```

```
CREATE TYPE имя AS ENUM
```

```
( [ 'метка' [, ... ] ] )
```

```
CREATE TYPE имя AS RANGE (
```

```
    SUBTYPE = подтип
```

```
    [ , SUBTYPE_OPCLASS = класс_оператора_подтипа ]
```

```
    [ , COLLATION = правило_сортировки ]
```

```
    [ , CANONICAL = каноническая_функция ]
```

```
    [ , SUBTYPE_DIFF = функция_разницы_подтипа ]
```

```
)
```

```
CREATE TYPE имя (
```

```
    INPUT = функция_ввода,
```

```
    OUTPUT = функция_вывода
```

```
    [ , RECEIVE = функция_получения ]
```

```
    [ , SEND = функция_отправки ]
```

```
    [ , TYPMOD_IN = функция_ввода_модификатора_типа ]
```

```
    [ , TYPMOD_OUT = функция_вывода_модификатора_типа ]
```

```
    [ , ANALYZE = функция_анализа ]
```

```
    [ , INTERNALLENGTH = { внутр_длина | VARIABLE } ]
```

```
    [ , PASSEDBYVALUE ]
```

```
    [ , ALIGNMENT = выравнивание ]
```

```
    [ , STORAGE = хранение ]
```

```
    [ , LIKE = тип_образец ]
```

```
    [ , CATEGORY = категория ]
```

```
    [ , PREFERRED = предпочитаемый ]
```

```
    [ , DEFAULT = по_умолчанию ]
```

```
    [ , ELEMENT = элемент ]
```

```
    [ , DELIMITER = разделитель ]
```

```
    [ , COLLATABLE = сортируемый ]
```

```
)
```

CREATE TYPE имя

55.2. Описание

CREATE TYPE регистрирует новый тип данных для использования в текущей базе данных. Владельцем типа становится создавший его пользователь.

Если указано имя схемы, тип создается в указанной схеме. В противном случае, он создается в текущей схеме. Имя типа должно отличаться от имен любых других существующих типов или доменов в той же схеме. А так как с таблицами связываются типы данных, имя типа должно также отличаться и от имен существующих таблиц в этой схеме.

Команда CREATE TYPE имеет пять форм, показанных выше в сводке синтаксиса. Они создают соответственно составной тип, перечисление, диапазон, базовый тип или тип-пустышку. Первые четыре эти типа рассматриваются по порядку ниже. Тип-пустышка представляет собой просто заготовку для типа, который будет определен позже; он создается командой CREATE TYPE с одним именем, без параметров. Типы-пустышки необходимы для определения прямых ссылок при создании базовых типов и типов-диапазонов, как описывается в соответствующих разделах.

55.2.1. Составные типы

Первая форма CREATE TYPE создает составной тип. Составной тип задается списком имен и типами данных атрибутов. Если тип данных является сортируемым, то для атрибута можно также задать правило сортировки. Составной тип по сути не отличается от типа строки таблицы, но CREATE TYPE избавляет от необходимости создавать таблицу, когда все, что нужно, это создать тип. Отдельный составной тип может быть полезен, например, для передачи аргументов или результатов функции.

Чтобы создать составной тип, необходимо иметь право USAGE для типов всех его атрибутов.

55.2.2. Типы перечислений

Вторая форма CREATE TYPE создаёт тип-перечисление. Перечисления

принимают список меток в кавычках. Максимальная длина каждой метки – `NAMEDATALEN` байт (64 байта в стандартной сборке Синергия-БД). Также возможно создать перечисляемый тип без меток, но этот тип нельзя будет использовать для хранения значений, пока командой `ALTER TYPE` не будет добавлена хотя бы одна метка.

55.2.3. Диапазонные типы

Третья форма `CREATE TYPE` создает тип-диапазон.

Задаваемый для диапазона подтип может быть любым типом со связанным классом операторов В-дерева (что позволяет определить порядок значений в диапазоне). Обычно порядок элементов определяет класс операторов В-дерева по умолчанию, но его можно изменить, задав имя другого класса в параметре `класс_операторов_подтипа`. Если подтип поддерживает сортировку и требуется, чтобы значения упорядочивались с нестандартным правилом сортировки, его имя можно задать в параметре `правило_сортировки`.

Необязательная каноническая_функция должна принимать один аргумент определяемого типа диапазона и возвращать значение того же типа. Это используется для преобразования значений диапазона в каноническую форму, когда это уместно. За дополнительными сведениями обратитесь к . Создается каноническая_функция несколько нетривиально, так как она должна быть уже определена, прежде чем можно будет объявить тип-диапазон. Для этого нужно сначала создать тип-пустышку, который будет заготовкой типа, не имеющей никаких свойств, кроме имени и владельца. Это можно сделать, выполнив команду `CREATE TYPE имя` без дополнительных параметров. Затем можно объявить функцию, для которой тип-пустышка будет типом аргумента и результата, и, наконец, объявить тип-диапазон с тем же именем. При этом тип-пустышка автоматически заменится полноценным типом-диапазоном.

Необязательная функция_разницы_подтипа должна принимать в аргументах два значения типа подтип и возвращать значение `double precision`, представляющее разницу между двумя данными значениями. Хотя эту

функцию можно не использовать, она позволяет кардинально увеличить эффективность индексов GiST для колонок с типом-диапазоном.

55.2.4. Базовые типы

Четвертая форма `CREATE TYPE` создает новый базовый тип (скалярный тип). Чтобы создать новый базовый тип, нужно быть суперпользователем. Это ограничение введено потому, что ошибочное определение типа может вызвать нарушения или даже сбой в работе сервера.

Эти параметры могут перечисляться в любом порядке, не только в показанном выше, и большинство из них необязательные.

Прежде чем создавать тип, необходимо зарегистрировать две или более функций (с помощью `CREATE FUNCTION`). Обязательными являются функции `функция_ввода` и `функция_вывода`, тогда как `функция_получения`, `функция_отправки`, `функция_модификатора_типа`, `функция_вывода_модификатора_типа` и `функция_анализа` могут отсутствовать. Обычно эти функции разрабатываются на C или другом низкоуровневом языке.

`Функция_ввода` преобразует внешнее текстовое представление типа во внутреннее, с которым работают операторы и функции, определенные для этого типа. `Функция_вывода` выполняет обратное преобразование. Функцию ввода можно объявить как принимающую один аргумент типа `cstring`, либо как принимающую три аргумента типов `cstring`, `oid` и `integer`. В первом аргументе передается вводимый текст в виде строки в стиле C, во втором аргументе – собственный OID типа (кроме типов массивов, для которых передается OID типа элемента), а в третьем – `модификатор_типа` для целевой колонки, если он определен (или -1 в противном случае). Функция ввода должна возвращать значение нового типа данных. Обычно функция ввода должна быть строгой (`STRICT`); если это не так, при получении на вход значения `NULL` она будет вызываться с первым параметром `NULL`. Функция может в этом случае сама вернуть `NULL` или вызвать ошибку. Это полезно в основном для поддержки функций ввода доменных типов,

которые не должны принимать данные `NULL`. Функция вывода должна принимать один аргумент нового типа данных, а возвращать она должна `cstring`. Для значений `NULL` функции вывода не вызываются.

Необязательная функция `_получения` преобразует двоичное внешнее представление типа во внутреннее представление. Если эта функция отсутствует, новый тип не сможет участвовать в двоичном вводе. Двоичное представление следует выбирать таким, чтобы оно легко переводилось во внутреннюю форму и при этом было переносимым до разумной степени. Например, для стандартных целочисленных типов данных во внешнем двоичном представлении выбран сетевой порядок байтов, тогда как внутреннее представление определяется порядком байтов в процессоре. Функция получения должна выполнить проверку вводимого значения на допустимость. Функция получения может быть объявлена как принимающая один аргумент типа `internal`, либо как принимающая три аргумента типов `internal`, `oid` и `integer`. В первом аргументе передается указатель на буфер `StringInfo`, содержащий полученную байтовую строку, а дополнительные аргументы такие же, как и для функции ввода текста. Функция получения должна возвращать значение нового типа данных. Обычно функция получения должна быть строгой (`STRICT`); если это не так, при получении на вход значения `NULL`, она будет вызываться с первым параметром `NULL`. Функция может в этом случае сама вернуть `NULL` или вызывать ошибку. Это полезно в основном для поддержки функций получения доменных типов, которые не должны принимать значения `NULL`. Подобным образом, необязательная функция `_отправки` преобразует данные из внутреннего во внешнее двоичное представление. Если эта функция не определена, новый тип не может участвовать в двоичном выводе. Функция отправки должна принимать один аргумент нового типа данных, а возвращать она должна `bytea`. Для значений `NULL` функции отправки не вызываются.

Здесь у вас может возникнуть вопрос, как функции ввода и вывода могут быть объявлены принимающими или возвращающими значения нового типа, если они должны быть созданы до объявления нового типа. Ответ довольно прост: сначала

нужно создать тип-пустышку, который будет заготовкой типа, не имеющей никаких свойств, кроме имени и владельца. Это можно сделать, выполнив команду `CREATE TYPE имя` без дополнительных параметров. Затем можно будет определить функции ввода/вывода, ссылающиеся на этот тип. И наконец, команда `CREATE TYPE` с полным определением заменит тип-пустышку окончательным и полноценным определением, после чего новый тип можно будет использовать как обычно.

Необязательные функция_ввода_модификатора_типа и функция_вывода_модификатора_типа требуются, только если типы поддерживают модификаторы, или, другими словами, дополнительные ограничения, связываемые с объявлением типа, например `char(5)` или `numeric(30,2)`. В Синергия-БД типы могут принимать в качестве модификаторов одну или несколько простых констант или идентификаторов. Однако эти данные должны упаковываться в единственное неотрицательное целочисленное значение, которое и будет храниться в системных каталогах. Функция_ввода_модификатора_типа получает объявленные модификаторы в виде строки `cstring`. Она должна проверить значения на допустимость (и вызвать ошибку, если они неверны), а затем выдать неотрицательное значение `integer`, которое будет сохранено в колонке «`typmod`». Если для типа не определена функция_ввода_модификатора_типа, модификаторы типа приниматься не будут. Функция_вывода_модификатора_типа преобразует внутреннее целочисленное значение `typmod` обратно, в форму, понятную пользователю. Она должна вернуть значение `cstring`, которое именно в этом виде будет добавлено к имени типа; например, функция для `numeric` должна вернуть `(30,2)`. Функция_вывода_модификатора_типа может быть опущена, в этом случае сохраненное целочисленное значение `typmod` по умолчанию будет выводиться просто в виде числа, заключенного в скобки.

Необязательная функция_анализа выполняет сбор специфической для этого типа статистики в колонках с таким типом данных. По умолчанию `ANALYZE`

пытается собрать статистику, используя операторы «равно» и «меньше», если для этого типа определен класс операторов В-дерева по умолчанию. Для нескаларных типов это поведение скорее всего не подойдет, поэтому его можно переопределить, задав собственную функцию анализа. Эта функция должна принимать единственный аргумент типа `internal` и возвращать результат `boolean`. Более глубоко API функций анализа описан в `src/include/commands/vacuum.h`.

Если особенности внутреннего представления нового типа известны функциям ввода/вывода и другим функциям, созданным специально для работы с этим типом, необходимо определить ряд характеристик внутреннего представления, о которых должна знать Синергия-БД. В первую очередь это `internallength` (внутренняя длина). Если базовый тип данных имеет фиксированную длину, в `internallength` указывается эта длина в виде положительного числа, а если длина переменная, в `internallength` задается значение `VARIABLE`. (Внутри при этом `typlen` принимает значение `-1`.) Внутреннее представление всех типов переменной длины должно начинаться с 4-байтового целого, задающего общую длину значения этого типа.

Необязательный флаг `PASSEDBYVALUE` указывает, что значения этого типа данных передаются по значению, а не по ссылке. Передавать типы по значению можно, только если их внутреннее представление не превышает размера типа `Datum` (на большинстве платформ 4 байта, на некоторых – 8).

Параметр `выравнивание` определяет, как требуется выравнивать данные этого типа. Допускается выравнивание по границам 1, 2, 4 или 8 байт. Заметьте, что типы переменной длины должны быть выравнены как минимум по границе 4 байт, так как их первым компонентом обязательно должен быть `int4`.

Параметр `хранение` позволяет выбрать стратегию хранения для типов данных переменной длины. Для типов с фиксированной длиной поддерживается только вариант `plain`. Если выбрана стратегия `plain`, данные этого типа всегда хранятся внутри, без сжатия. Со стратегией `extended` система сначала попытается сжать большое значение, а затем выносит его из строки основной таблицы, если оно

все же окажется слишком большим. С `external` значение может быть вынесено из основной таблицы, но система не будет пытаться сжать его. Стратегия `main` позволяет сжать данные, но не стремится вынести их из основной таблицы. Элементы данных с этой стратегией хранения, тем не менее, могут быть вынесены из основной таблицы, если другого способа уместить их в строке нет, но все же она отдает большее предпочтение основной таблице, по сравнению со стратегиями `extended` и `external`.

Значения `storage`, отличные от `plain`, подразумевают, что функции типа данных могут принимать значения в формате `toast`. Эти значения просто определяют стратегию хранения TOAST по умолчанию для столбцов отделяемого в TOAST типа данных; пользователи могут выбирать другие стратегии для отдельных столбцов, применяя команду `ALTER TABLE SET STORAGE`.

Параметр `тип_образец` позволяет задать основные свойства представления типа другим способом: скопировать их из существующего типа. В частности, из указанного типа будут скопированы свойства `internallength`, `passedbyvalue`, `alignment` и `storage`. Также возможно, хотя обычно это не требуется, переопределить некоторые из этих значений, указав их вместе с предложением `LIKE`. Определять представление типа таким образом особенно удобно, когда низкоуровневая реализация нового типа некоторым образом опирается на существующий тип.

Параметры `категория` и `предпочитаемый` позволяют определять, какое неявное приведение будет применяться в неоднозначных ситуациях. Каждый тип данных принадлежит к некоторой категории, обозначаемой одним символом ASCII, при этом он может быть, либо не быть «предпочитаемым» в этой категории. Анализатор запроса по возможности выберет приведение к предпочитаемому типу (но только среди других типов той же категории), когда это может помочь разрешить имя перегруженной функции или оператора. Если для типа не определено неявное приведение к какому-либо другому типу или обратное, для этих параметров достаточно оставить значения по умолчанию. Однако если есть группа связанных типов, для которых определены неявные приведения, часто бывает полезно пометить

их все как принадлежащие некоторой категории и назначить один или два «наиболее общих» предпочитаемыми в этой категории. Параметр категория особенно полезен при добавлении типа, определенного пользователем, в существующую встроенную категорию, например, в категорию числовых или строковых типов. Однако так же возможно создать категории типов, полностью определенные пользователем. В качестве имени такой категории можно выбрать любой ASCII-символ, кроме латинской заглавной буквы.

Если пользователь хочет назначить колонкам с этим типом данных значение по умолчанию, отличное от NULL, он может задать его в этой команде, указав его после ключевого слова DEFAULT. Такое значение по умолчанию можно переопределить явным предложением DEFAULT, добавленным при создании колонки.

Чтобы обозначить, что тип является массивом, укажите тип элементов массива, добавив ключевое слово ELEMENT. Например, чтобы определить массив из четырехбайтовых целых (int4), укажите ELEMENT = int4. Дополнительные сведения о типах массивов приведены ниже.

Параметр delimiter позволяет задать разделитель, который будет вставляться между значениями во внешнем представлении массива с элементами этого типа. По умолчанию разделителем является запятая (,). Заметьте, что разделитель связывается с типом элементов массива, а не с типом самого массива.

Если необязательный логический параметр сортируемый равен true, определения колонок и выражения с этим типом могут включать указания о порядке сортировки, в предложении COLLATE. Как именно будут использоваться эти указания, зависит от реализации функций, работающих с этим типом; эти указания не действуют автоматически просто от того, что тип помечен как сортируемый.

55.2.5. Типы массивов

При создании любого нового типа Синергия-БД автоматически создает соответствующий тип массива, имя которого он получает, добавляя подчеркивание перед именем типа элементов. Если полученное имя оказывается не короче NAMEDATALEN байт, оно усекается. Если полученное таким образом имя

конфликтует с именем уже существующего типа, процесс повторяется, пока не будет получено уникальное имя. Этот неявно создаваемый тип массива имеет переменную длину и использует встроенные функции ввода и вывода `array_in` и `array_out`. Тип массива отражает любые изменения владельца или схемы связанного типа элемента и удаляется сам при удалении типа элемента.

Вы можете вполне резонно спросить, зачем нужен параметр `ELEMENT`, если система создает правильный тип массива автоматически. Единственный случай, когда параметр `ELEMENT` может быть полезен, это когда вы создаете тип фиксированной длины, который внутри оказывается массивом одинаковых элементов, и вы хотите, чтобы к этим элементам можно было обращаться по индексу, помимо того, что вы можете реализовать какие угодно операции с типом в целом. Например, тип `point` представлен просто как два числа с плавающей точкой, к которым можно обратиться так: `point[0]` и `point[1]`. Заметьте, что это работает только с типами фиксированной длины, которые представляют собой в точности последовательность одинаковых полей фиксированной длины. Тип массива переменной длины должен иметь обобщенное внутреннее представление, с которым умеют работать `array_in` и `array_out`. По историческим причинам (т.е. это определенно некорректно, но менять уже слишком поздно), индексы в массивах фиксированной длины начинаются с нуля, а не с 1, как в массивах переменной длины.

55.3.Параметры

имя

Имя (возможно, дополненное схемой) создаваемого типа.

имя_атрибута

Имя атрибута (колонки) составного типа.

тип_данных

Имя существующего типа данных, который станет типом колонки составного типа.

правило_сортировки

Имя существующего правила сортировки, связываемого с колонкой составного типа или с типом-диапазоном.

метка

Строковая константа, представляющая текстовую метку, связанную с отдельным значением типа-перечисления.

подтип

Имя типа элемента, множество значений которого будет представлять тип-диапазон.

класс_оператора_подтипа

Имя класса операторов В-дерева для подтипа.

каноническая_функция

Имя функции канонизации для типа-диапазона.

функция_разницы_подтипа

Имя функции разницы для значений подтипа.

функция_ввода

Имя функции, преобразующей данные из внешнего текстового представления типа во внутреннюю форму.

функция_вывода

Имя функции, преобразующей данные из внутренней формы во внешнее текстовое представление типа.

функция_получения

Имя функции, преобразующей данные из внешнего двоичного представления типа во внутреннюю форму.

функция_отправки

Имя функции, преобразующей данные из внутренней формы во внешнее двоичное представление типа.

функция_ввода_модификатора_типа

Имя функции, преобразующей массив модификаторов типа во внутреннюю форму.

функция_вывода_модификатора_типа

Имя функции, преобразующей внутреннюю форму модификаторов типа во

внешнее текстовое представление.

функция_анализа

Имя функции, производящей статистический анализ типа данных.

внутр_длина

Числовая константа, задающая размер внутреннего представления нового типа в байтах. По умолчанию предполагается, что тип имеет переменную длину.

выравнивание

Требуемое выравнивание для типа данных. Допустимые значения этого параметра, если он указывается: `char`, `int2`, `int4` или `double`; по умолчанию подразумевается `int4`.

хранение

Стратегия хранения для типа данных. Допустимые значения этого параметра, если он указывается: `plain`, `external`, `extended` или `main`; по умолчанию подразумевается `plain`.

тип_образец

Имя существующего типа данных, от которого новый тип получит свойства представления. Из этого типа будут скопированы значения параметров `internallength`, `passedbyvalue`, `alignment` и `storage`, если их не переопределят явные указания, заданные дополнительно в этой команде `CREATE TYPE`.

категория

Код категории (один символ ASCII) для этого типа. По умолчанию подразумевается `'U'` (что означает пользовательский тип, «User-defined»). Для нестандартных категорий можно выбрать другие ASCII-символы.

предпочитаемый

Если значение этого параметра равно `true`, создаваемый тип будет предпочитаемым в своей категории. По умолчанию подразумевается `false`. Будьте очень осторожны, создавая новый предпочитаемый тип в существующей категории, так как это может поменять поведение выражений неожиданным образом.

по_умолчанию

Значение по умолчанию для создаваемого типа данных. Если не указано, значением по умолчанию будет NULL.

элемент

Создаваемый тип будет массивом; этот параметр определяет тип элементов массива.

разделитель

Символ, разделяющий значения в массивах, образованных из значений создаваемого типа.

сортируемый

Если значение этого параметра равно true, в операциях с создаваемым типом может учитываться информация о правилах сортировки. По умолчанию подразумевается false.

55.4.Замечания

Так как на использование типа данных после создания не накладываются ограничения, объявление базового типа или типа-диапазона по сути дает всем право на выполнение функций, упомянутых в определении типа. Обычно это не проблема для таких функций, какие бывают полезны в определении типов. Но прежде чем создать тип, преобразование которого во внешнюю форму и обратно будет использовать «секретную» информацию, стоит подумать дважды.

В Синергия-БД имя генерируемого типа-массива всегда образовалось из имени типа элемента и добавленного спереди символа подчеркивания (). Таким образом, допустимая максимальная длина имени типа была на символ меньше, чем длины других имен. Хотя и сейчас имя типа массива чаще всего образуется таким образом, оно может быть и другим в случае достижения максимальной длины или конфликтов с именами пользовательских типов, начинающихся с подчеркивания. Поэтому полагаться на это соглашение в коде не рекомендуется. Вместо этого, имя типа массива, связанного с данным типом, следует определять по значению `pg_type.typarray`.

Вообще же можно посоветовать не использовать имена типов и таблиц, начинающиеся с подчеркивания. Хотя сервер сможет сгенерировать другое имя, не

конфликтующее с пользовательским, некоторая путаница все же возможна, особенно со старыми клиентскими приложениями, которые могут полагать, что имя типа, начинающееся с подчеркивания, всегда относится к типу массива.

55.5.Примеры

В этом примере создается составной тип, а затем он используется в определении функции:

```
CREATE TYPE compfoo AS (f1 int, f2 text);
```

```
CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$  
    SELECT fooid, fooname FROM foo  
$$ LANGUAGE SQL;
```

В этом примере создается тип-перечисление, а затем он используется в определении таблицы:

```
CREATE TYPE bug_status AS ENUM ('new', 'open',  
    'closed');
```

```
CREATE TABLE bug (  
    id serial,  
    description text,  
    status bug_status  
);
```

В этом примере создается тип-диапазон:

```
CREATE TYPE float8_range AS RANGE (subtype = float8,  
    subtype_diff = float8mi);
```

В следующем примере создается базовый тип данных box, а затем он используется в определении таблицы:

```
CREATE TYPE box;
```

```
CREATE FUNCTION my_box_in_function(cstring) RETURNS box  
AS ... ;  
CREATE FUNCTION my_box_out_function(box) RETURNS cstring  
AS ... ;
```

```
CREATE TYPE box (  
    INTERNALLENGTH = 16,  
    INPUT = my_box_in_function,  
    OUTPUT = my_box_out_function  
);
```

```
CREATE TABLE myboxes (  
    id integer,  
    description box  
);
```

Если бы внутренней структурой box был массив из четырех элементов float4, вместо этого можно было бы использовать определение:

```
CREATE TYPE box (  
    INTERNALLENGTH = 16,  
    INPUT = my_box_in_function,  
    OUTPUT = my_box_out_function,  
    ELEMENT = float4  
);
```

В таком случае к числам, составляющим значение этого типа, можно было бы обращаться по индексу. В остальном поведение этого типа будет таким же.

В этом примере создается тип большого объекта, а затем он используется в определении таблицы:

```
CREATE TYPE bigobj (  
    INPUT = lo_filein, OUTPUT = lo_fileout,  
    INTERNALLENGTH = VARIABLE  
);  
CREATE TABLE big_objs (  
    id integer,  
    obj bigobj  
);
```


55.6.Совместимость

Первая форма команды CREATE TYPE, создающая составной тип, соответствует стандарту SQL. Другие формы являются расширениями Синергия-БД. Для оператора CREATE TYPE в стандарте SQL также определены другие формы, не реализованные в Синергия-БД.

Возможность создавать составной тип без атрибутов – специфическое отклонение Синергия-БД от стандарта (как и аналогичная особенность команды CREATE TABLE).

56. CREATE USER

CREATE USER – создать роль в базе данных.

56.1.Синтаксис

CREATE USER имя [[WITH] параметр [...]]

Здесь параметр:

```
SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| BYPASSRLS | NOBYPASSRLS
| CONNECTION LIMIT предел_подключений
| [ ENCRYPTED ] PASSWORD 'пароль' | PASSWORD NULL
| VALID UNTIL 'дата_время'
| IN ROLE имя_роли [, ...]
| IN GROUP имя_роли [, ...]
| ROLE имя_роли [, ...]
| ADMIN имя_роли [, ...]
| USER имя_роли [, ...]
| SYSID uid
```

56.2.Описание

Команда `CREATE USER` теперь является просто синонимом `CREATE ROLE`. Единственное отличие в том, что для команды, записанной в виде `CREATE USER`, по умолчанию подразумевается `LOGIN`, а в виде `CREATE ROLE` подразумевается `NOLOGIN`.

56.3.Совместимость

Оператор `CREATE USER` является расширением Синергия-БД. В стандарте SQL определение пользователей считается зависимым от реализации.

57. CREATE USER MAPPING

`CREATE USER MAPPING` – создать сопоставление пользователя для стороннего сервера.

57.1.Синтаксис

```
CREATE USER MAPPING [ IF NOT EXISTS ] FOR
{ имя_пользователя | USER | CURRENT_USER | PUBLIC }
    SERVER имя_сервера
    [ OPTIONS ( параметр 'значение' [ , ... ] ) ]
```

57.2.Описание

`CREATE USER MAPPING` создает сопоставление пользователя на внешнем сервере. Сопоставление пользователя обычно содержит информацию о подключении, которую будет использовать обертка сторонних данных вместе с информацией о стороннем сервере для получения доступа к внешнему ресурсу.

Владелец стороннего сервера может создать сопоставление для любых пользователей на этом сервере. Кроме того, пользователь может создать сопоставление для своего собственного имени пользователя, если он наделен правом `USAGE` на данном сервере.

57.3.Параметры

`IF NOT EXISTS`

Не считать ошибкой, если сопоставление данного пользователя для данного стороннего сервера уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее сопоставление как-то соотносится с тем, которое могло бы быть создано. `имя_пользователя`

Имя существующего пользователя, для которого создается сопоставление на стороннем сервере. Ключевые слова `CURRENT_USER` и `USER` обозначают имя текущего пользователя. Если указывается `PUBLIC`, создается так называемое общее сопоставление, которое будет использоваться при отсутствии сопоставления для конкретного пользователя.

`имя_сервера`

Имя существующего сервера, для которого создается сопоставление пользователя.

`OPTIONS ( параметр 'значение' [, ... ] )`

В этом предложении задаются параметры сопоставления. Эти параметры обычно определяют фактическое имя и пароль пользователя на целевом сервере. Имена параметров должны быть уникальными. Набор допустимых имен и значений параметров определяется оберткой сторонних данных внешнего сервера.

57.4.Примеры

Создание сопоставления для пользователя bob на сервере foo:

```
CREATE USER MAPPING FOR bob SERVER foo OPTIONS (user
'bob', password 'secret');
```

57.5.Совместимость

`CREATE USER MAPPING` соответствует стандарту ISO/IEC 9075-9 (SQL/MED).

58. CREATE VIEW

`CREATE VIEW` – создать представление.

58.1.Синтаксис

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ]
```

```
VIEW имя [ ( имя_колонки [, ...] ) ]  
    [ WITH ( имя_параметра_представления [= значение_параметра_представления] [, ...] ) ]  
    AS запрос  
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

58.2.Описание

CREATE VIEW создает представление запроса. Создаваемое представление лишено физической материализации, поэтому указанный запрос будет выполняться при каждом обращении к представлению.

Команда CREATE OR REPLACE VIEW действует подобным образом, но если представление с этим именем уже существует, оно заменяется. Новый запрос должен выдавать те же колонки, что выдавал запрос, ранее определенный для этого представления (то есть, колонки с такими же именами должны иметь те же типы данных и следовать в том же порядке), но может добавить несколько новых колонок в конце списка. Вычисления, в результате которых формируются колонки представления, могут быть совершенно другими.

Если задано имя схемы (например, CREATE VIEW myschema.myview ...), представление создается в указанной схеме, в противном случае – в текущей. Временные представления существуют в специальной схеме, так что при создании таких представлений имя схемы задать нельзя. Имя представления должно отличаться от имен других представлений, таблиц, последовательностей, индексов или сторонних таблиц в этой схеме.

58.3.Параметры

TEMPORARY или TEMP

С таким указанием представление создается как временное. Временные представления автоматически удаляются в конце сеанса. Существующее постоянное представление с тем же именем не будет видно в текущем сеансе, пока существует временное, однако к нему можно обратиться, дополнив имя указанием схемы.

Если в определении представления задействованы временные таблицы, представление так же создается как временное (вне зависимости от присутствия

явного указания TEMPORARY).

RECURSIVE

Создает рекурсивное представление. Синтаксис:

```
CREATE RECURSIVE VIEW имя (колонки) AS SELECT ...;
```

Равнозначен:

```
CREATE VIEW имя AS WITH RECURSIVE имя (колонки) AS  
(SELECT ...) SELECT колонки FROM имя;
```

Для рекурсивного представления обязательно должен задаваться список колонок.

имя

Имя (возможно, дополненное схемой) создаваемого представления.

имя_колонки

Необязательный список имен, назначаемых колонкам представления. Если отсутствует, имена колонок формируются из результатов запроса.

```
WITH ( имя_параметра_представления [=  
значение_параметра_представления] [, ... ] )
```

В этом предложении могут задаваться следующие необязательные параметры представления:

check_option (string)

Этот параметр может принимать значение local (локально) или cascaded (каскадно) и равнозначен указанию WITH [CASCADED | LOCAL] CHECK OPTION. Изменить этот параметр у существующего представления с помощью ALTER VIEW нельзя.

security_barrier (string)

Этот параметр следует использовать, если представление должно обеспечивать безопасность на уровне строк.

запрос

Команда SELECT или VALUES, которая выдает колонки и строки представления.

```
WITH [ CASCADED | LOCAL ] CHECK OPTION
```

Это указание управляет поведением автоматически изменяемых

представлений. Если оно присутствует, при выполнении операций INSERT и UPDATE с этим представлением будет проверяться, удовлетворяют ли новые строки условию, определяющему представление (то есть, проверяется, будут ли новые строки видны через это представление). Если они не удовлетворяют условию, операция не будет выполнена. Если указание CHECK OPTION отсутствует, команды INSERT и UPDATE смогут создавать в этом представлении строки, которые не будут видны в нем. Поддерживаются следующие варианты проверки:

LOCAL

Новые строки проверяются только по условиям, определенным непосредственно в самом представлении. Любые условия, определенные в нижележащих базовых представлениях, не проверяются (если только в них нет указания CHECK OPTION).

CASCADED

Новые строки проверяются по условиям данного представления и всех нижележащих базовых. Если указано CHECK OPTION, а LOCAL и CASCADED опущено, подразумевается указание CASCADED.

Указание CHECK OPTION нельзя использовать с рекурсивными представлениями.

Заметьте, что CHECK OPTION поддерживается только для автоматически изменяемых представлений, не имеющих триггеров INSTEAD OF и правил INSTEAD. Если автоматически изменяемое представление определено поверх базового представления с триггерами INSTEAD OF, то для проверки ограничений автоматически изменяемого представления можно применить указание LOCAL CHECK OPTION, хотя условия базового представления с триггерами INSTEAD OF при этом проверяться не будут (каскадная проверка не будет спускаться к представлению, модифицируемому триггером, и любые параметры проверки, определенные для такого представления, будут просто игнорироваться). Если для представления или любого из его базовых отношений определено правило INSTEAD, приводящее к перезаписи команды INSERT или UPDATE, в

перезаписанном запросе все параметры проверки будут игнорироваться, в том числе проверки автоматически изменяемых представлений, определенных поверх отношений с правилом `INSTEAD`.

58.4.Замечания

Для удаления представлений применяется оператор `DROP VIEW`.

Будьте аккуратны в определении представления, чтобы получить желаемые имена и типы колонок. Например, такая команда:

```
CREATE VIEW vista AS SELECT 'Hello World';
```

создаст представление с двумя недостатками: именем колонки по умолчанию будет `?column?`, а типом данных – `unknown` (неизвестный). Если вы хотите получить в представлении строковую константу, лучше сделать так:

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

Доступ к таблицам, задействованным в представлении, определяется правами владельца представления. В некоторых случаях это позволяет организовать безопасный, но ограниченный доступ к нижележащим таблицам. Однако учтите, что не все представления могут быть защищенными. Функции, вызываемые в представлении, выполняются так, как будто они вызываются непосредственно из запроса, обращающегося к представлению. Поэтому пользователь представления должен иметь все права, необходимые для вызова всех функций, задействованных в представлении.

При выполнении `CREATE OR REPLACE VIEW` для существующего представления меняется только правило `SELECT`, определяющее представление. Другие свойства представления, включая владельца, права и правила, кроме `SELECT`, остаются неизменными. Чтобы изменить определение представления, необходимо быть его владельцем (или членом роли-владельца).

58.5.Изменяемые представления

Простые представления становятся изменяемыми автоматически: система позволит выполнять команды `INSERT`, `UPDATE` и `DELETE` с таким представлением так же, как и с обычной таблицей. Представление будет автоматически изменяемым,

если оно удовлетворяют одновременно всем следующим условиям:

- список FROM в запросе, определяющем представлении, должен содержать ровно один элемент, и это должна быть таблица или другое изменяемое представление;
- определение представления не должно содержать предложения WITH, DISTINCT, GROUP BY, HAVING, LIMIT и OFFSET на верхнем уровне запроса;
- определение представления не должно содержать операции с множествами (UNION, INTERSECT, EXCEPT) на верхнем уровне запроса;
- список выборки в запросе не должен содержать агрегатные и оконные функции, а также функции, возвращающие множества.

Автоматически обновляемое представление может содержать как изменяемые, так и не изменяемые колонки. Колонка будет изменяемой, если это простая ссылка на изменяемую колонку нижележащего базового отношения; в противном случае, эта колонка будет доступна только для чтения, и если команда INSERT или UPDATE попытается записать значение в нее, возникнет ошибка.

Если представление автоматически изменяемое, система будет преобразовывать обращающиеся к нему операторы INSERT, UPDATE и DELETE в соответствующие операторы, обращающиеся к нижележащему базовому отношению. При этом в полной мере поддерживаются операторы INSERT с предложением ON CONFLICT UPDATE.

Если автоматически изменяемое представление содержит условие WHERE, это условие ограничивает набор строк, которые могут быть изменены командой UPDATE и удалены командой DELETE в этом представлении. Однако UPDATE может изменить строку так, что она больше не будет соответствовать условию WHERE и, как следствие, больше не будет видна через представление. Команда INSERT подобным образом может вставить в базовое отношение строки, которые не удовлетворяют условию WHERE и поэтому не будут видны через представление. Чтобы запретить командам INSERT и UPDATE создавать такие строки, которые не

видны через представление, можно воспользоваться указанием CHECK OPTION.

Если автоматически изменяемое представление имеет свойство `security_barrier` (барьер безопасности), то все условия WHERE этого представления (и все условия с герметичными операторами (LEAKPROOF)) будут всегда вычисляться перед условиями, добавленными пользователем представления. Заметьте, что по этой причине строки, которые в конце концов не были выданы (потому что не прошли проверку в пользовательском условии WHERE), могут все же остаться заблокированными. Чтобы определить, какие условия применяются на уровне отношения (и, как следствие, избавляют часть строк от блокировки), можно воспользоваться командой EXPLAIN.

Более сложные представления, не удовлетворяющие этим условиям, по умолчанию доступны только для чтения: система не позволит выполнить операции добавления, изменения или удаления строк в таком представлении. Создать эффект изменяемого представления для них можно, определив триггеры INSTEAD OF, которые будут преобразовывать запросы на изменение данных в соответствующие действия с другими таблицами. Так же есть возможность создавать правила (CREATE RULE), но на практике триггеры проще для понимания и применения.

Учтите, что пользователь, выполняющий операции добавления, изменения или удаления данных в представлении, должен иметь соответствующие права для этого представления. Кроме того, владелец представления должен иметь сопутствующие права в нижележащих базовых отношениях, хотя пользователь, собственно выполняющий эти операции, может этих прав не иметь.

58.6.Примеры

Создание представления, содержащего все комедийные фильмы:

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';
```

Эта команда создаст представление с колонками, которые содержались в

таблице `film` в момент выполнения команды. Хотя при создании представления было указано `*`, колонки, добавляемые в таблицу позже, частью представления не будут.

Создание представления с указанием `LOCAL CHECK OPTION`:

```
CREATE VIEW universal_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'U'
    WITH LOCAL CHECK OPTION;
```

Эта команда создаст представление на базе представления `comedies`, выдающее только комедии (`kind = 'Comedy'`) универсальной возрастной категории `classification = 'U'`. Любая попытка выполнить в представлении `INSERT` или `UPDATE` со строкой, не удовлетворяющей условию `classification = 'U'`, будет отвергнута, но ограничение по полю `kind` (тип фильма) проверяться не будет.

Создание представления с указанием `CASCADED CHECK OPTION`:

```
CREATE VIEW pg_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'PG'
    WITH CASCADED CHECK OPTION;
```

Это представление будет проверять, удовлетворяют ли новые строки обоим условиям: по колонке `kind` и по колонке `classification`.

Создание представления с изменяемыми и неизменяемыми колонками:

```
CREATE VIEW comedies AS
    SELECT f.*,
           country_code_to_name(f.country_code) AS
country,
           (SELECT avg(r.rating)
            FROM user_ratings r
            WHERE r.film_id = f.id) AS avg_rating
```

```
FROM films f
WHERE f.kind = 'Comedy';
```

Это представление будет поддерживать операции INSERT, UPDATE и DELETE. Изменяемыми будут все колонки из таблицы films, тогда как вычисляемые колонки country и avg_rating будут доступны только для чтения.

Создание рекурсивного представления, содержащего числа от 1 до 100:

```
CREATE RECURSIVE VIEW nums_1_100 (n) AS
VALUES (1)
UNION ALL
SELECT n+1 FROM nums_1_100 WHERE n < 100;
```

Заметьте, что несмотря на то, что имя рекурсивного представления дополнено схемой в этой команде CREATE, внутренняя ссылка представления на себя же схемой не дополняется. Это связано с тем, что имя неявно создаваемого СТЕ не может дополняться схемой.

58.7. Совместимость

Команда CREATE OR REPLACE VIEW – языковое расширение Синергия-БД. Так же расширением является предложение WITH (...) и концепция временного представления.

59. DEALLOCATE

DEALLOCATE – освободить подготовленный оператор.

59.1. Синтаксис

```
DEALLOCATE [ PREPARE ] { имя | ALL }
```

59.2. Описание

DEALLOCATE применяется для освобождения ранее подготовленного оператора SQL. Если не освободить подготовленный оператор явно, он будет освобожден при завершении сеанса.

За дополнительными сведениями о подготовленных операторах обратитесь к PREPARE.

59.3.Параметры

PREPARE

Это ключевое слово игнорируется.

ИМЯ

Имя подготовленного оператора, подлежащего освобождению.

ALL

Освобождает все подготовленные операторы.

59.4.Совместимость

В стандарте SQL есть оператор DEALLOCATE, но он предназначен только для применения во встраиваемом SQL.

60. DECLARE

DECLARE – определить курсор.

60.1.Синтаксис

```
DECLARE имя [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]  
        CURSOR [ { WITH | WITHOUT } HOLD ] FOR запрос
```

60.2.Описание

Оператор DECLARE позволяет пользователю создавать курсоры, с помощью которых можно выбирать по очереди некоторое количество строк из результата большого запроса. Когда курсор создан, через него можно получать строки, применяя команду FETCH.

Примечание. На этой странице описывается применение курсоров на уровне команд SQL. Если вы попытаетесь использовать курсоры внутри функции PL/pgSQL, правила будут другими.

60.3.Параметры

ИМЯ

Имя создаваемого курсора.

BINARY

Курсор с таким свойством возвращает данные в двоичном, а не текстовом

формате.

INSENSITIVE

Указывает, что данные, считываемые из курсора, не должны зависеть от изменений, которые могут происходить в нижележащих таблицах после создания курсора. В Синергия-БД это поведение подразумевается по умолчанию, так что это ключевое слово ни на что не влияет и принимается только для совместимости со стандартом SQL.

SCROLL

NO SCROLL

Указание SCROLL определяет, что курсор может прокручивать набор данных и получать строки непоследовательно (например, в обратном порядке). В зависимости от сложности плана запроса указание SCROLL может отрицательно отразиться на скорости выполнения запроса. Указание NO SCROLL, напротив, определяет, что через курсор нельзя будет получать строки в произвольном порядке. По умолчанию прокрутка в некоторых случаях разрешается; но это не равнозначно эффекту указания SCROLL.

WITH HOLD

WITHOUT HOLD

Указание WITH HOLD определяет, что курсор можно продолжать использовать после успешной фиксации создавшей его транзакции. WITHOUT HOLD определяет, что курсор нельзя будет использовать за рамками транзакции, создавшей его. Если не указано ни WITHOUT HOLD, ни WITH HOLD, по умолчанию подразумевается WITHOUT HOLD.

запрос

Команда SELECT или VALUES, поставляющая строки для курсора.

Ключевые слова BINARY, INSENSITIVE и SCROLL могут указываться в любом порядке.

60.4.Замечания

Обычный курсор выдает данные в текстовом виде, в каком их выдает SELECT. Однако с указанием BINARY курсор может выдавать их и в двоичном формате. Это

упрощает операции преобразования данных для сервера и клиента, за счет дополнительных усилий, требующихся от программиста для работы с платформозависимыми двоичными форматами. Например, если запрос получает значение 1 из целочисленной колонки, обычный курсор выдаст строку, содержащую 1, тогда как через двоичный курсор будет получено четырехбайтовое поле, содержащее внутреннее представление значения (с сетевым порядком байтов).

Двоичные курсоры должны применяться с осмотрительностью. Многие приложения, в том числе `psql`, не приспособлены к работе с двоичными курсорами и ожидают, что данные будут поступать в текстовом формате.

Примечание. Когда клиентское приложение выполняет команду `FETCH`, используя протокол «расширенных запросов», в сообщении `Bind` этого протокола указывается, в каком формате, текстовом или двоичном, должны быть получены данные. Это указание переопределяет свойство курсора, заданное в его объявлении. Таким образом, концепция курсора, объявляемого двоичным, становится устаревшей при использовании протокола расширенных запросов – любой курсор может быть прочитан как текстовый или двоичный.

Если в команде объявления курсора не указано `WITH HOLD`, созданный ей курсор может использоваться только в текущей транзакции. Таким образом, оператор `DECLARE` без `WITH HOLD` бесполезен вне блока транзакции: курсор будет существовать только до завершения этого оператора. Поэтому Синергия-БД сообщает об ошибке, если такая команда выполняется вне блока транзакции. Чтобы определить блок транзакции, примените команды `BEGIN` и `COMMIT` (или `ROLLBACK`).

Если в объявлении курсора указано `WITH HOLD` и транзакция, создавшая курсор, успешно фиксируется, к этому курсору могут продолжать обращаться последующие транзакции в этом сеансе. Но если создавшая курсор транзакция прерывается, курсор уничтожается. Курсор со свойством `WITH HOLD` (удерживаемый) может быть закрыт явно, командой `CLOSE`, либо неявно, по завершении сеанса. В текущей реализации строки, представляемые удерживаемым курсором, копируются во временный файл или в область памяти, так что они остаются доступными для следующих транзакций.

Объявить курсор со свойством `WITH HOLD` можно, только если запрос не

содержит указаний `FOR UPDATE` и `FOR SHARE`.

Параметр `SCROLL` указывается при определении курсора, который будет выбирать данные в обратном порядке. Это поведение требуется стандартом SQL. Однако для совместимости с предыдущими версиями, Синергия-БД допускает выборку в обратном направлении и без указания `SCROLL`, если план запроса курсора достаточно прост, чтобы реализовать прокрутку назад без дополнительных операций. Тем не менее, разработчикам приложений не следует рассчитывать на то, что курсор, созданный без указания `SCROLL`, можно будет прокручивать назад. С указанием `NO SCROLL` прокрутка назад запрещается в любом случае.

Выборка в обратном направлении также запрещается, если запрос содержит указания `FOR UPDATE` и `FOR SHARE`; в этом случае параметр `SCROLL` не принимается.

Примечание. Прокручиваемые и удерживаемые (`WITH HOLD`) курсоры могут выдавать неожиданные результаты, если они вызывают изменчивые функции. Когда повторно выбирается ранее прочитанная строка, функции могут вызываться снова и выдавать результаты, отличные от полученных в первый раз. Один из способов обойти эту проблему – объявить курсор с указанием `WITH HOLD` и зафиксировать транзакцию, прежде чем читать из него какие-либо строки. В этом случае весь набор данных курсора будет материализован во временном хранилище, так что изменчивые функции будут выполнены для каждой строки лишь единожды.

Если запрос в определении курсора включает указания `FOR UPDATE` или `FOR SHARE`, возвращаемые курсором строки блокируются в момент первой выборки, так же, как это происходит при выполнении `SELECT` с этими указаниями. Кроме того, при чтении строк будут возвращаться их наиболее актуальные версии; таким образом, с этими указаниями курсор будет вести себя как «чувствительный курсор», определенный в стандарте SQL (указать `INSENSITIVE` для курсора с запросом `FOR UPDATE` или `FOR SHARE` нельзя).

Примечания:

1. Обычно рекомендуется использовать `FOR UPDATE`, если курсор предназначен для применения в командах `UPDATE ... WHERE CURRENT OF` и `DELETE ... WHERE CURRENT OF`. Указание `FOR UPDATE` предотвращает изменение строк другими сеансами после того, как они были считаны, и до того, как выполнится команда. Без `FOR UPDATE` последующая команда с `WHERE CURRENT OF` не сработает, если строка будет изменена после создания курсора.

2. Ещё одна причина использовать указание `FOR UPDATE` в том, что без него последующие команды с `WHERE CURRENT OF` могут выдать ошибку, если запрос курсора не удовлетворяет оговоренному в стандарте SQL критерию «простой изменяемости» (в частности, курсор должен ссылаться только на одну таблицу и не должен использовать группировку и сортировку (`ORDER BY`)). Курсоры, не удовлетворяющие этому критерию, могут работать, либо не работать, в зависимости от конкретного выбранного плана; так что в худшем случае приложение может работать в тестовой, но сломается в производственной среде.

3. Не использовать же `FOR UPDATE` для команд с `WHERE CURRENT OF` в основном имеет смысл, только если требуется получить прокручиваемый курсор или курсор, не отражающий последующие изменения (то есть, продолжающий показывать прежние данные). Если это действительно необходимо, обязательно учтите при реализации приведенные выше замечания.

В стандарте SQL механизм курсоров предусмотрен только для встраиваемого SQL. Синергия-БД не реализует для курсоров оператор `OPEN`; курсор считается открытым при объявлении. Однако ECPG, встраиваемый препроцессор SQL для Синергия-БД, следует соглашениям стандарта, в том числе поддерживая для курсоров операторы `DECLARE` и `OPEN`.

Получить список всех доступных курсоров можно, обратившись к системному представлению `pg_cursors`.

60.5.Примеры

Объявление курсора:

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

Другие примеры использования курсора можно найти в `FETCH`.

60.6.Совместимость

В стандарте SQL говорится, что чувствительность курсоров к параллельному обновлению нижележащих данных по умолчанию определяется реализацией. В Синергия-БД курсоры по умолчанию нечувствительные, а чувствительными их можно сделать с помощью указания `FOR UPDATE`. Другие СУБД могут работать иначе.

Стандарт SQL допускает курсоры только во встраиваемом SQL и в модулях. Синергия-БД позволяет использовать курсоры интерактивно.

Двоичные курсоры являются расширением Синергия-БД.

61. DELETE

DELETE – удалить записи таблицы.

61.1. Синтаксис

```
[ WITH [ RECURSIVE ] запрос_WITH [ , ... ] ]  
DELETE FROM [ ONLY ] имя_таблицы [ * ] [ [ AS ]  
псевдоним ]  
    [ USING список_USING ]  
    [ WHERE условие | WHERE CURRENT OF имя_курсора ]  
    [ RETURNING * | выражение_результата [ [ AS ]  
имя_результата ] [ , ... ] ]
```

61.2. Описание

Команда DELETE удаляет из указанной таблицы строки, удовлетворяющие условию WHERE. Если предложение WHERE отсутствует, она удаляет из таблицы все строки, в результате будет получена рабочая, но пустая таблица.

Примечание. TRUNCATE реализует более быстрый механизм удаления всех строк из таблицы.

Удалить строки в таблице, используя информацию из других таблиц в базе данных, можно двумя способами: применяя вложенные запросы или указав дополнительные таблицы в предложении USING. Выбор предпочитаемого варианта зависит от конкретных обстоятельств.

Предложение RETURNING указывает, что команда DELETE должна вычислить и вернуть значения для каждой фактически удаленной строки. Вычислить в нем можно любое выражение с колонками целевой таблицы и/или колонками других таблиц, упомянутых в USING. Список RETURNING имеет тот же синтаксис, что и список результатов SELECT.

Чтобы удалять данные из таблицы, необходимо иметь право DELETE для нее, а также право SELECT для всех таблиц, перечисленных в предложении USING, и таблиц, данные которых считываются в условии.

61.3.Параметры

запрос_WITH

Предложение WITH позволяет задать один или несколько подзапросов, на которые затем можно ссылаться по имени в запросе DELETE. Подробнее об этом см. SELECT.

имя_таблицы

Имя (возможно, дополненное схемой) таблицы, из которой будут удалены строки. Если перед именем таблицы добавлено ONLY, соответствующие строки удаляются только из указанной таблицы. Без ONLY строки будут также удалены из всех таблиц, унаследованных от указанной. При желании, после имени таблицы можно указать *, чтобы явно обозначить, что операция затрагивает все дочерние таблицы.

псевдоним

Альтернативное имя целевой таблицы. Когда указывается это имя, оно полностью скрывает фактическое имя таблицы. Например, в запросе DELETE FROM foo AS f дополнительные компоненты оператора DELETE должны обращаться к целевой таблице по имени f, а не foo.

список_USING

Список табличных выражений, позволяющий добавлять в условие WHERE колонки из других таблиц. Он подобен списку таблиц, который можно задать в предложении Предложение FROM оператора SELECT; например, в нем можно определить псевдоним для таблицы. Целевую таблицу повторять в списке_USING нужно, только если требуется определить замкнутое соединение.

условие

Выражение, возвращающее значение типа boolean. Удалены будут только те строки, для которых это выражение возвращает true.

имя_курсора

Имя курсора, который будет использоваться в условии WHERE CURRENT OF. С таким условием будет удалена строка, выбранная из этого курсора последней. Курсор должен образовываться запросом, не применяющим группировку, к целевой

таблице команды DELETE. Заметьте, что WHERE CURRENT OF нельзя задать вместе с булевым условием. За дополнительными сведениями об использовании курсоров с WHERE CURRENT OF обратитесь к DECLARE.

выражение_результата

Выражение, которое будет вычисляться и возвращаться командой DELETE после удаления каждой строки. В этом выражении можно использовать имена любых колонок таблицы имя_таблицы или таблиц, перечисленных в списке USING. Чтобы получить все колонки, достаточно написать *.

имя_результата

Имя, назначаемое возвращаемой колонке.

61.4.Выводимая информация

В случае успешного завершения, DELETE возвращает метку команды в виде:

DELETE число

Здесь число – количество удаленных строк. Заметьте, что это число может быть меньше числа строк, соответствующих условию, если удаления были подавлены триггером BEFORE DELETE. Если число равно 0, это означает, что запрос не удалил ни одной строки (это не считается ошибкой).

Если команда DELETE содержит предложение RETURNING, её результат будет похож на результат оператора SELECT (с теми же колонками и значениями, что содержатся в списке RETURNING), полученный для строк, удаленных этой командой.

61.5.Замечания

Синергия-БД позволяет ссылаться на колонки других таблиц в условии WHERE, когда эти таблицы перечисляются в предложении USING. Например, удалить все фильмы определенного продюсера можно так:

```
DELETE FROM films USING producers
```

```
WHERE producer_id = producers.id AND producers.name =  
'foo';
```

По сути в этом запросе выполняется соединение таблиц films и producers,

и все успешно включенные в соединение строки в `films` помечаются для удаления. Этот синтаксис не соответствует стандарту. Следуя стандарту, эту задачу можно решить так:

```
DELETE FROM films
  WHERE producer_id IN (SELECT id FROM producers WHERE
name = 'foo');
```

В ряде случаев запрос в стиле соединения легче написать и он может работать быстрее, чем в стиле вложенного запроса.

61.6.Примеры

Удаление всех фильмов, кроме мюзиклов:

```
DELETE FROM films WHERE kind <> 'Musical';
```

Очистка таблицы `films`:

```
DELETE FROM films;
```

Удаление завершенных задач с получением всех данных удаленных строк:

```
DELETE FROM tasks WHERE status = 'DONE' RETURNING *;
```

Удаление из `tasks` строки, на которой в текущий момент располагается курсор `c_tasks`:

```
DELETE FROM tasks WHERE CURRENT OF c_tasks;
```

61.7.Совместимость

Эта команда соответствует стандарту SQL, но предложения `USING` и `RETURNING` являются расширениями Синергия-БД, как и возможность использовать `WITH` с `DELETE`.

62. DISCARD

`DISCARD` – очистить состояние сеанса.

62.1.Синтаксис

```
DISCARD { ALL | PLANS | SEQUENCES | TEMPORARY | TEMP }
```

62.2.Описание

`DISCARD` высвобождает внутренние ресурсы, связанные с сеансом

использования базы данных. Эта команда полезна для частичного или полного сброса состояния сеанса. Для освобождения различных типов ресурсов она поддерживает несколько разных подкоманд; вариант DISCARD ALL включает в себя все остальные и также сбрасывает дополнительное состояние.

62.3.Параметры

PLANS

Высвобождает все кешированные планы запросов, вынуждая сервер провести планирование заново при следующем использовании связанного подготовленного оператора.

SEQUENCES

Сбрасывает кешированное состояние, связанное с последовательностями, включая внутреннюю информацию `currval()/lastval()` и любые предварительно выделенные значения последовательностей, которые ещё не выдала функция `nextval()`.

TEMPORARY или TEMP

Удаляет все временные таблицы, созданные в текущем сеансе.

ALL

Высвобождает все временные ресурсы, связанные с текущим сеансом, и сбрасывает сеанс к начальному состоянию. В настоящее время действует так же, как и следующая последовательность операторов:

```
SET SESSION AUTHORIZATION DEFAULT;  
RESET ALL;  
DEALLOCATE ALL;  
CLOSE ALL;  
UNLISTEN *;  
SELECT pg_advisory_unlock_all();  
DISCARD PLANS;  
DISCARD SEQUENCES;  
DISCARD TEMP;
```

62.4.Замечания

DISCARD ALL нельзя выполнять внутри блока транзакции.

62.5.Совместимость

DISCARD является расширением Синергия-БД.

63. DO

DO – выполнить анонимный блок кода.

63.1.Синтаксис

DO [LANGUAGE имя_языка] код

63.2.Описание

DO выполняет анонимный блок кода или, другими словами, разовую анонимную функцию на процедурном языке.

Блок кода воспринимается, как если бы это было тело функции, которая не имеет параметров и возвращает `void`. Этот код разбирается и выполняется один раз.

Необязательное предложение LANGUAGE можно записать до или после блока кода.

63.3.Параметры

код

Выполняемый код на процедурном языке. Он должен задаваться в виде текстовой строки (ее рекомендуется заключать в доллары), как и код в CREATE FUNCTION.

имя_языка

Имя процедурного языка, на котором написан код. По умолчанию подразумевается `plpgsql`.

63.4.Замечания

Применяемый процедурный язык должен быть уже зарегистрирован в текущей базе с помощью команды CREATE EXTENSION. По умолчанию зарегистрирован

только `plpgsql`, но не другие языки.

Пользователь должен иметь право `USAGE` для данного процедурного языка, либо быть суперпользователем, если этот язык недоверенный. Такие же требования действуют и при создании функции на этом языке.

Если команда `DO` выполняется в блоке транзакции, код процедуры не может вызывать операторы управления транзакциями. Такие операторы допускаются, только если `DO` выполняется в отдельной транзакции.

63.5.Примеры

Следующий код даст все права для всех представлений в схеме `public` роли `webuser`:

```
DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT table_schema, table_name FROM
information_schema.tables
            WHERE table_type = 'VIEW' AND table_schema
= 'public'
        LOOP
            EXECUTE 'GRANT ALL ON ' ||
quote_ident(r.table_schema) || '.' ||
quote_ident(r.table_name) || ' TO webuser';
        END LOOP;
END$$;
```

63.6.Совместимость

Оператор `DO` отсутствует в стандарте SQL.

64. DROP ACCESS METHOD

`DROP ACCESS METHOD` – удалить метод доступа.

64.1.Синтаксис

```
DROP ACCESS METHOD [ IF EXISTS ] имя [ CASCADE |
RESTRICT ]
```

64.2.Описание

DROP ACCESS METHOD удаляет существующий метод доступа. Удалять методы доступа разрешено только суперпользователям.

64.3.Параметры

IF EXISTS

Не считать ошибкой, если метод доступа не существует. В этом случае будет выдано замечание.

ИМЯ

Имя существующего метода доступа.

CASCADE

Автоматически удалять объекты, зависящие от данного метода доступа (например, классы и семейства операторов, а также индексы), и, в свою очередь, все зависящие от них объекты.

RESTRICT

Отказать в удалении метода доступа, если от него зависят какие-либо объекты. Это поведение по умолчанию.

64.4.Примеры

Удаление метода доступа heptree:

```
DROP ACCESS METHOD heptree;
```

64.5.Совместимость

DROP ACCESS METHOD является расширением Синергия-БД.

65. DROP AGGREGATE

DROP AGGREGATE – удалить агрегатную функцию.

65.1.Синтаксис

```
DROP AGGREGATE [ IF EXISTS ] имя  
( сигнатура_агр_функции ) [ CASCADE | RESTRICT ]
```

Здесь сигнатура_агр_функции:

* |


```
[ режим_аргумента ] [ имя_аргумента ] тип_аргумента
[ , ... ] |
[ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
[ , ... ] ] ORDER BY [ режим_аргумента ]
[ имя_аргумента ] тип_аргумента [ , ... ]
```

65.2.Описание

DROP AGGREGATE удаляет существующую агрегатную функцию.

Пользователь, выполняющий эту команду, должен быть владельцем агрегатной функции.

65.3.Параметры

IF EXISTS

Не считать ошибкой, если агрегатная функция не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) существующей агрегатной функции.

режим_аргумента

Режим аргумента: IN или VARIADIC. По умолчанию подразумевается IN.

имя_аргумента

Имя аргумента. Заметьте, что на самом деле DROP AGGREGATE не обращает внимание на имена аргументов, так как для однозначной идентификации агрегатной функции достаточно только типов аргументов.

тип_аргумента

Тип входных данных, с которыми работает агрегатная функция. Чтобы сослаться на агрегатную функцию без аргументов, укажите вместо списка аргументов *, а чтобы сослаться на сортирующую агрегатную функцию, добавьте ORDER BY между указаниями непосредственных и агрегируемых аргументов.

CASCADE

Автоматически удалять объекты, зависящие от данной агрегатной функции.

RESTRICT

Отказать в удалении агрегатной функции, если от нее зависят какие-либо объекты. Это поведение по умолчанию.

65.4.Примеры

Удаление агрегатной функции myavg для типа integer:

```
DROP AGGREGATE myavg(integer);
```

Удаление гипотезирующей агрегатной функции myrank, которая принимает произвольный список сортируемых колонок и соответствующий список прямых аргументов:

```
DROP AGGREGATE myrank(VARIADIC «any» ORDER BY VARIADIC  
«any»);
```

Удаление нескольких агрегатных функций одной командой:

```
DROP AGGREGATE myavg(integer), myavg(bigint);
```

65.5.Совместимость

Оператор DROP AGGREGATE отсутствует в стандарте SQL.

66. DROP CAST

DROP CAST – удалить приведение типа.

66.1.Синтаксис

```
DROP CAST [ IF EXISTS ] (исходный_тип AS целевой_тип)  
[ CASCADE | RESTRICT ]
```

66.2.Описание

DROP CAST удаляет ранее определенное приведение типа.

Чтобы удалить приведение, необходимо быть владельцем исходного или целевого типа данных. Такие же требования действуют и при создании приведения.

66.3.Параметры

IF EXISTS

Не считать ошибкой, если приведение не существует. В этом случае будет выдано замечание.

исходный_тип

Имя исходного типа данных для приведения.

целевой_тип

Имя целевого типа данных для приведения.

CASCADE
RESTRICT

Эти ключевые слова игнорируются, так как от приведений не зависят никакие объекты.

66.4.Примеры

Удаление приведения типа `text` к типу `int`:

```
DROP CAST (text AS int);
```

66.5.Совместимость

Команда `DROP CAST` соответствует стандарту SQL.

67. DROP COLLATION

`DROP COLLATION` – удалить правило сортировки.

67.1.Синтаксис

```
DROP COLLATION [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

67.2.Описание

`DROP COLLATION` удаляет ранее определенное правило сортировки. Чтобы удалить правило сортировки, необходимо быть его владельцем.

67.3.Параметры

`IF EXISTS`

Не считать ошибкой, если правило сортировки не существует. В этом случае будет выдано замечание.

имя

Имя правила сортировки, возможно, дополненное схемой.

CASCADE

Автоматически удалять объекты, зависящие от данного правила сортировки.

RESTRICT

Отказать в удалении правила сортировки, если от него зависят какие-либо объекты. Это поведение по умолчанию.

67.4.Примеры

Удаление правила сортировки с именем german:

```
DROP COLLATION german;
```

67.5.Совместимость

Команда DROP COLLATION соответствует стандарту SQL, за исключением параметра IF EXISTS, являющегося расширением Синергия-БД.

68. DROP CONVERSION

DROP CONVERSION – удалить преобразование.

68.1.Синтаксис

```
DROP CONVERSION [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

68.2.Описание

DROP CONVERSION удаляет ранее определенное преобразование. Чтобы удалить преобразование, необходимо быть его владельцем.

68.3.Параметры

IF EXISTS

Не считать ошибкой, если преобразование не существует. В этом случае будет выдано замечание.

имя

Имя преобразования, возможно, дополненное схемой.

CASCADE

RESTRICT

Эти ключевые слова игнорируются, так как от преобразований не зависят никакие объекты.

68.4.Примеры

Удаление преобразования с именем myname:

```
DROP CONVERSION myname;
```

68.5.Совместимость

Оператор DROP CONVERSION отсутствует в стандарте SQL, хотя в нем есть оператор DROP TRANSLATION, сопутствующий оператору CREATE TRANSLATION, который, в свою очередь, подобен CREATE CONVERSION в Синергия-БД.

69. DROP DATABASE

DROP DATABASE – удалить базу данных.

69.1.Синтаксис

DROP DATABASE [IF EXISTS] имя

69.2.Описание

Оператор DROP DATABASE удаляет базу данных. Он удаляет из каталогов записи, относящиеся к базе данных, а также удаляет каталог, содержащий данные. Выполнить эту команду может только владелец базы данных. Кроме того, удалить базу нельзя, пока к ней подключены вы или кто-то еще. Чтобы выполнить эту команду, подключитесь к postgres или любой другой базе данных.

Действие команды DROP DATABASE нельзя отменить.

69.3.Параметры

IF EXISTS

Не считать ошибкой, если база данных не существует. В этом случае будет выдано замечание.

имя

Имя базы данных, подлежащей удалению.

69.4.Замечания

DROP DATABASE нельзя выполнять внутри блока транзакции.

Эту команду нельзя выполнить, если установлено подключение к удаляемой базе данных. Поэтому может быть удобнее вместо нее использовать программу dropdb, которая сама вызывает эту команду внутри.

69.5.Совместимость

Оператор DROP DATABASE отсутствует в стандарте SQL.

70. DROP DOMAIN

DROP DOMAIN – удалить домен.

70.1.Синтаксис

```
DROP DOMAIN [ IF EXISTS ] имя [, ...] [ CASCADE |  
RESTRICT ]
```

70.2.Описание

DROP DOMAIN удаляет домен. Удалить домен может только его владелец.

70.3.Параметры

IF EXISTS

Не считать ошибкой, если домен не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) существующего домена.

CASCADE

Автоматически удалять объекты, зависящие от данного домена (например, колонки таблиц).

RESTRICT

Отказать в удалении домена, если от него зависят какие-либо объекты. Это поведение по умолчанию.

70.4.Примеры

Удаление домена box:

```
DROP DOMAIN box;
```

70.5.Совместимость

Эта команда соответствует стандарту SQL, за исключением параметра IF EXISTS, являющегося расширением Синергия-БД.

71. DROP EVENT TRIGGER

DROP EVENT TRIGGER – удалить событийный триггер.

71.1.Синтаксис

```
DROP EVENT TRIGGER [ IF EXISTS ] имя [ CASCADE |  
RESTRICT ]
```

71.2.Описание

DROP EVENT TRIGGER удаляет существующий событийный триггер. Пользователь, выполняющий эту команду, должен быть владельцем событийного триггера.

71.3.Параметры

IF EXISTS

Не считать ошибкой, если событийный триггер не существует. В этом случае будет выдано замечание.

имя

Имя событийного триггера, подлежащего удалению.

CASCADE

Автоматически удалять объекты, зависящие от данного триггера.

RESTRICT

Отказать в удалении триггера, если от него зависят какие-либо объекты. Это поведение по умолчанию.

71.4.Примеры

Удаление триггера snitch:

```
DROP EVENT TRIGGER snitch;
```

71.5.Совместимость

Оператор DROP EVENT TRIGGER отсутствует в стандарте SQL.

72. DROP EXTENSION

DROP EXTENSION – удалить расширение.

72.1.Синтаксис

```
DROP EXTENSION [ IF EXISTS ] имя [, ...] [ CASCADE |  
RESTRICT ]
```

72.2.Описание

DROP EXTENSION удаляет расширения из базы данных. При удалении расширения также удаляются все составляющие его объекты.

Чтобы выполнить DROP EXTENSION, необходимо быть владельцем расширения.

72.3.Параметры

IF EXISTS

Не считать ошибкой, если расширение не существует. В этом случае будет выдано замечание.

имя

Имя установленного расширения.

CASCADE

Автоматически удалять объекты, зависящие от расширения.

RESTRICT

Отказать в удалении расширения, если от него зависят какие-либо объекты (кроме объектов, составляющих его, и других расширений, перечисленных в той же команде DROP). Это поведение по умолчанию.

72.4.Примеры

Удаление расширения hstore из текущей базы данных.

```
DROP EXTENSION hstore;
```

Эта команда не будет выполнена, если какие-либо объекты из hstore будут задействованы, например, если в какой-либо таблице окажется колонка типа hstore. Чтобы принудительно удалить и эти зависимые объекты, необходимо добавить параметр CASCADE.

72.5.Совместимость

DROP EXTENSION является расширением Синергия-БД.

73. DROP FOREIGN DATA WRAPPER

DROP FOREIGN DATA WRAPPER – удалить обертку сторонних данных.

73.1.Синтаксис

```
DROP FOREIGN DATA WRAPPER [ IF EXISTS ] имя [ CASCADE |  
RESTRICT ]
```

73.2.Описание

DROP FOREIGN DATA WRAPPER удаляет существующую обертку сторонних данных. Пользователь, выполняющий эту команду, должен быть владельцем обертки.

73.3.Параметры

IF EXISTS

Не считать ошибкой, если обертка сторонних данных не существует. В этом случае будет выдано замечание.

имя

Имя существующей обертки сторонних данных.

CASCADE

Автоматически удалять объекты, зависящие от этой обертки сторонних данных (например, серверы).

RESTRICT

Отказаться в удалении обертки сторонних данных, если от нее зависят какие-либо объекты. Это поведение по умолчанию.

73.4.Примеры

Удаление обертки сторонних данных dbi:

```
DROP FOREIGN DATA WRAPPER dbi;
```

73.5.Совместимость

DROP FOREIGN DATA WRAPPER соответствует стандарту ISO/IEC 9075-9 (SQL/MED). Предложение IF EXISTS является расширением Синергия-БД.

74. DROP FOREIGN TABLE

DROP FOREIGN TABLE – удалить стороннюю таблицу.

74.1.Синтаксис

```
DROP FOREIGN TABLE [ IF EXISTS ] имя [, ...] [ CASCADE |  
RESTRICT ]
```

74.2.Описание

DROP FOREIGN TABLE удаляет стороннюю таблицу. Удалить стороннюю таблицу может только её владелец.

74.3.Параметры

IF EXISTS

Не считать ошибкой, если сторонняя таблица не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) сторонней таблицы, подлежащей удалению.

CASCADE

Автоматически удалять объекты, зависящие от данной сторонней таблицы (например, представления).

RESTRICT

Отказаться в удалении сторонней таблицы, если от нее зависят какие-либо объекты. Это поведение по умолчанию.

74.4.Примеры

Удаление двух сторонних таблиц, films и distributors:

```
DROP FOREIGN TABLE films, distributors;
```

74.5. Совместимость

Эта команда соответствует ISO/IEC 9075-9 (SQL/MED), но возможность удалять в одной команде несколько таблиц и указание IF EXISTS являются расширениями Синергия-БД.

75. DROP FUNCTION

DROP FUNCTION – удалить функцию.

75.1. Синтаксис

```
DROP FUNCTION [ IF EXISTS ] имя ( [ [ режим_аргумента ]  
[ имя_аргумента ] тип_аргумента [, ...] ] )  
[ CASCADE | RESTRICT ]
```

75.2. Описание

DROP FUNCTION удаляет определение существующей функции. Пользователь, выполняющий эту команду, должен быть владельцем функции. Помимо имени функции требуется указать типы её аргументов, так как в базе данных могут существовать несколько функций с одним именем, но с разными списками аргументов.

75.3. Параметры

IF EXISTS

Не считать ошибкой, если функция не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) существующей функции.

режим_аргумента

Режим аргумента: IN, OUT, INOUT или VARIADIC. По умолчанию подразумевается IN. Заметьте, что DROP FUNCTION на самом деле не учитывает аргументы OUT, так как для идентификации функции нужны только типы входных аргументов. Поэтому достаточно перечислить только аргументы IN, INOUT и VARIADIC.

имя_аргумента

Имя аргумента. Заметьте, что на самом деле DROP FUNCTION не обращает внимание на имена аргументов, так как для однозначной идентификации функции достаточно только типов аргументов.

тип_аргумента

Тип данных аргументов функции (возможно, дополненный именем схемы), если таковые имеются.

CASCADE

Автоматически удалять объекты, зависящие от данной функции (например, операторы или триггеры).

RESTRICT

Отказать в удалении функции, если от нее зависят какие-либо объекты. Это поведение по умолчанию.

75.4.Примеры

Эта команда удаляет функцию, вычисляющую квадратный корень:

```
DROP FUNCTION sqrt(integer);
```

Удаление нескольких функций одной командой:

```
DROP FUNCTION sqrt(integer), sqrt(bigint);
```

Если имя функции уникально в её схеме, на неё можно сослаться без списка аргументов:

```
DROP FUNCTION update_employee_salaries;
```

Заметьте, что это отличается от

```
DROP FUNCTION update_employee_salaries();
```

Данная форма ссылается на функцию с нулём аргументов, тогда как первый вариант подходит для функции с любым числом аргументов, в том числе и с нулём, если имя функции уникально.

75.5.Совместимость

Эта команда соответствует стандарту SQL, но дополнена следующими расширениями Синергия-БД:

- стандарт допускает удаление в одной команде только одной функции;
- возможность указывать режимы и имена аргументов.

76. DROP GROUP

DROP GROUP – удалить роль в базе данных.

76.1.Синтаксис

```
DROP GROUP [ IF EXISTS ] имя [, ...]
```

76.2.Описание

DROP GROUP теперь является синонимом оператора DROP ROLE.

76.3.Совместимость

Оператор DROP GROUP отсутствует в стандарте SQL.

77. DROP INDEX

DROP INDEX – удалить индекс.

77.1.Синтаксис

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] имя [, ...]  
[ CASCADE | RESTRICT ]
```

77.2.Описание

DROP INDEX удаляет существующий индекс из базы данных. Выполнить эту команду может только владелец индекса.

77.3.Параметры

CONCURRENTLY

С этим указанием индекс удаляется, не блокируя параллельные операции выборки, добавления, изменения и удаления данных в таблице индекса. Обычный оператор DROP INDEX запрашивает исключительную блокировку для таблицы, запрещая другим обращаться к ней до завершения удаления. Если же добавлено это указание, команда, напротив, будет ждать завершения конфликтующих транзакций.

Применяя это указание, надо учитывать несколько особенностей. В частности,

при этом можно задать имя только одного индекса, а параметр CASCADE не поддерживается. Таким образом, индекс, поддерживающий ограничение UNIQUE или PRIMARY KEY, так удалить нельзя. Кроме того, обычную команду DROP INDEX можно выполнить в блоке транзакции, а DROP INDEX CONCURRENTLY – нет.

IF EXISTS

Не считать ошибкой, если индекс не существует. В этом случае будет выдано замечание.

ИМЯ

Имя (возможно, дополненное схемой) индекса, подлежащего удалению.

CASCADE

Автоматически удалять объекты, зависящие от данного индекса.

RESTRICT

Отказать в удалении индекса, если от него зависят какие-либо объекты. Это поведение по умолчанию.

77.4.Примеры

Эта команда удалит индекс title_idx:

```
DROP INDEX title_idx;
```

77.5.Совместимость

DROP INDEX является языковым расширением Синергия-БД. Средства обеспечения индексов в стандарте SQL не описаны.

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

SQL	-	Structured Query Language (язык структурированных запросов)
WAL	-	Write-Ahead Log (журнал упреждающей записи)
БД	-	база данных
КСЗ	-	комплекс средств защиты
ОЗУ	-	оперативное запоминающее устройство
ОС	-	операционная система
ПРД	-	правила разграничения доступа
СУБД	-	система управления базами данных

Лист регистрации изменений

[illegible]

Федеральное государственное унитарное предприятие
Российский федеральный ядерный центр
Всероссийский научно-исследовательский институт экспериментальной физики

УТВЕРЖДЕН
07623615.00082-02 34 01-ЛУ

Име. № подл.	Подп. и дата	Взам. инв. №	Име. № дубл.	Подп. и дата

КОМПЛЕКС ПРОГРАММ В ЗАЩИЩЕННОМ ИСПОЛНЕНИИ
«СИСТЕМА ПОЛНОГО ЖИЗНЕННОГО ЦИКЛА ИЗДЕЛИЙ
«ЦИФРОВОЕ ПРЕДПРИЯТИЕ»

**Программный модуль
«Система управления базами данных «Синергия-БД»**

Руководство оператора

Часть 4

07623615.00082-02 34 01-4

Листов 190

СОДЕРЖАНИЕ

1. DROP INDEX	5
2. DROP LANGUAGE.....	6
3. DROP MATERIALIZED VIEW	7
4. DROP OPERATOR	8
5. DROP OPERATOR CLASS.....	9
6. DROP OPERATOR FAMILY	11
7. DROP OWNED	12
8. DROP POLICY	13
9. DROP PROCEDURE.....	14
10. DROP PUBLICATION.....	16
11. DROP ROLE	16
12. DROP ROUTINE.....	18
13. DROP RULE	18
14. DROP SCHEMA.....	19
15. DROP SEQUENCE.....	21
16. DROP SERVER.....	21
17. DROP STATISTICS	22
18. DROP SUBSCRIPTION	23
19. DROP TABLE.....	25
20. DROP TABLESPACE.....	26
21. DROP TEXT SEARCH CONFIGURATION	27
22. DROP TEXT SEARCH DICTIONARY	28
23. DROP TEXT SEARCH PARSER	29
24. DROP TEXT SEARCH TEMPLATE	30

25. DROP TRANSFORM.....	31
26. DROP TRIGGER.....	32
27. DROP TYPE.....	33
28. DROP USER.....	34
29. DROP USER MAPPING	35
30. DROP VIEW	36
31. END	37
32. EXECUTE	38
33. EXPLAIN.....	39
34. FETCH.....	46
35. GRANT.....	51
36. IMPORT FOREIGN SCHEMA	63
37. INSERT.....	65
38. LISTEN.....	76
39. LOAD	78
40. LOCK	79
41. MOVE	83
42. NOTIFY.....	85
43. PREPARE.....	88
44. PREPARE TRANSACTION.....	92
45. REASSIGN OWNED	95
46. REFRESH MATERIALIZED VIEW	96
47. REINDEX.....	98
48. RELEASE SAVEPOINT.....	102
49. RESET	103

50. REVOKE.....	104
51. ROLLBACK	110
52. ROLLBACK PREPARED	111
53. ROLLBACK TO SAVEPOINT.....	112
54. SAVEPOINT	114
55. SELECT.....	115
56. SELECT INTO	151
57. SET	153
58. SET CONSTRAINTS	156
59. SET ROLE	158
60. SET SESSION AUTHORIZATION.....	160
61. SET TRANSACTION	162
62. SHOW	166
63. START TRANSACTION.....	168
64. TRUNCATE	169
65. UNLISTEN.....	172
66. UPDATE.....	173
67. VACUUM.....	180
68. VALUES	184
Перечень сокращений	189

Документ содержит справочную информацию о командах SQL, поддерживаемых СУБД «Синергия-БД» (далее Синергия-БД). Под «SQL» понимается язык в общем; информация о соответствии стандарту SQL и совместимости с ним приведена на страницах, соответствующих каждой команде.

1. DROP INDEX

DROP INDEX – удалить индекс.

1.1. Синтаксис

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] имя [, ...]  
[ CASCADE | RESTRICT ]
```

1.2. Описание

DROP INDEX удаляет существующий индекс из базы данных. Выполнить эту команду может только владелец индекса.

1.3. Параметры

CONCURRENTLY

С этим указанием индекс удаляется, не блокируя параллельные операции выборки, добавления, изменения и удаления данных в таблице индекса. Обычный оператор DROP INDEX запрашивает исключительную блокировку для таблицы, запрещая другим обращаться к ней до завершения удаления. Если же добавлено это указание, команда, напротив, будет ждать завершения конфликтующих транзакций.

Применяя это указание, надо учитывать несколько особенностей. В частности, при этом можно задать имя только одного индекса, а параметр CASCADE не поддерживается. Таким образом, индекс, поддерживающий ограничение UNIQUE или PRIMARY KEY, так удалить нельзя. Кроме того, обычную команду DROP INDEX можно выполнить в блоке транзакции, а DROP INDEX CONCURRENTLY – нет.

IF EXISTS

Не считать ошибкой, если индекс не существует. В этом случае будет выдано

замечание.

ИМЯ

Имя (возможно, дополненное схемой) индекса, подлежащего удалению.

CASCADE

Автоматически удалять объекты, зависящие от данного индекса.

RESTRICT

Отказаться в удалении индекса, если от него зависят какие-либо объекты. Это поведение по умолчанию.

1.4. Примеры

Эта команда удалит индекс `title_idx`:

```
DROP INDEX title_idx;
```

1.5. Совместимость

`DROP INDEX` является языковым расширением Синергия-БД. Средства обеспечения индексов в стандарте SQL не описаны.

2. DROP LANGUAGE

`DROP LANGUAGE` – удалить процедурный язык.

2.1. Синтаксис

```
DROP [ PROCEDURAL ] LANGUAGE [ IF EXISTS ] имя [ CASCADE  
| RESTRICT ]
```

2.2. Описание

`DROP LANGUAGE` удаляет определение ранее зарегистрированного процедурного языка. Выполнить `DROP LANGUAGE` может только владелец языка или суперпользователь.

2.3. Параметры

IF EXISTS

Не считать ошибкой, если язык не существует. В этом случае будет выдано замечание.

ИМЯ

Имя существующего процедурного языка. В целях обратной совместимости имя можно заключить в апострофы.

CASCADE

Автоматически удалять объекты, зависящие от языка (например, функции на этом языке).

RESTRICT

Отказаться в удалении языка, если от него зависят какие-либо объекты. Это поведение по умолчанию.

2.4. Примеры

Эта команда удаляет процедурный язык `plsample`:

```
DROP LANGUAGE plsample;
```

2.5. Совместимость

Оператор `DROP LANGUAGE` отсутствует в стандарте SQL.

3. DROP MATERIALIZED VIEW

`DROP MATERIALIZED VIEW` – удалить материализованное представление.

3.1. Синтаксис

```
DROP MATERIALIZED VIEW [ IF EXISTS ] имя [, ...]  
[ CASCADE | RESTRICT ]
```

3.2. Описание

`DROP MATERIALIZED VIEW` удаляет существующее материализованное представление. Выполнить эту команду может только владелец материализованного представления.

3.3. Параметры

`IF EXISTS`

Не считать ошибкой, если материализованное представление не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) материализованного представления,

подлежащего удалению.

CASCADE

Автоматически удалять объекты, зависящие от данного материализованного представления (например, другие материализованные или обычные представления), и, в свою очередь, все зависящие от них объекты.

RESTRICT

Отказаться в удалении материализованного представления, если от него зависят какие-либо объекты. Это поведение по умолчанию.

3.4. Примеры

Эта команда удаляет материализованное представление с именем `order_summary`:

```
DROP MATERIALIZED VIEW order_summary;
```

3.5. Совместимость

DROP MATERIALIZED VIEW – расширение Синергия-БД.

4. DROP OPERATOR

DROP OPERATOR – удалить оператор.

4.1. Синтаксис

```
DROP OPERATOR [ IF EXISTS ] имя ( { тип_слева | NONE } ,  
{ тип_справа | NONE } ) [ CASCADE | RESTRICT ]
```

4.2. Описание

DROP OPERATOR удаляет существующий оператор из базы данных. Выполнить эту команду может только владелец оператора.

4.3. Параметры

IF EXISTS

Не считать ошибкой, если оператор не существует. В этом случае будет выдано замечание.

ИМЯ

Имя (возможно, дополненное схемой) существующего оператора.

тип_слева

Тип данных левого операнда оператора; если у оператора нет левого операнда, укажите NONE.

тип_справа

Тип данных правого операнда оператора; если у оператора нет правого операнда, укажите NONE.

CASCADE

Автоматически удалять объекты, зависящие от данного оператора.

RESTRICT

Отказать в удалении оператора, если от него зависят какие-либо объекты. Это поведение по умолчанию.

4.4. Примеры

Удаление оператора возведения в степень a^b для типа integer:

```
DROP OPERATOR ^ (integer, integer);
```

Удаление левого унарного оператора двоичного дополнения $\sim b$ для типа bit:

```
DROP OPERATOR ~ (none, bit);
```

Удаление правого унарного оператора вычисления факториала $x!$ для типа bigint:

```
DROP OPERATOR ! (bigint, none);
```

Удаление нескольких операторов одной командой:

```
DROP OPERATOR ~ (none, bit), ! (bigint, none);
```

4.5. Совместимость

Команда DROP OPERATOR отсутствует в стандарте SQL.

5. DROP OPERATOR CLASS

DROP OPERATOR CLASS – удалить класс операторов.

5.1. Синтаксис

```
DROP OPERATOR CLASS [ IF EXISTS ] имя USING  
метод_индекса [ CASCADE | RESTRICT ]
```

5.2. Описание

`DROP OPERATOR CLASS` удаляет существующий класс операторов. Выполнить эту команду может только владелец класса операторов.

`DROP OPERATOR CLASS` не удаляет операторы или функции, связанные с этим классом. Если же существуют индексы, зависящие от этого класса, класс будет удален успешно (вместе с индексами), только если добавить указание `CASCADE`.

5.3. Параметры

`IF EXISTS`

Не считать ошибкой, если класс операторов не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) существующего класса операторов.

метод_индекса

Имя индексного метода, для которого предназначен этот класс операторов.

`CASCADE`

Автоматически удалять объекты, зависящие от данного класса операторов.

`RESTRICT`

Отказать в удалении класса операторов, если от него зависят какие-либо объекты. Это поведение по умолчанию.

5.4. Замечания

`DROP OPERATOR CLASS` не удалит семейство операторов, содержавшее этот класс, даже если в этом семействе больше ничего не останется (в том числе, если семейство было неявно создано командой `CREATE OPERATOR CLASS`). Пустое семейство операторов безвредно, но порядка ради затем следует удалить и его, командой `DROP OPERATOR FAMILY`; или, возможно, выполнить `DROP OPERATOR FAMILY` в первую очередь.

5.5. Примеры

Удаление класса операторов B-дерева с именем `widget_ops`:

```
DROP OPERATOR CLASS widget_ops USING btree;
```

Эта команда не будет выполнена, если в базе существуют индексы, использующие этот класс. Чтобы удалить такие индексы вместе с классом операторов, нужно добавить указание CASCADE.

5.6. Совместимость

Команда DROP OPERATOR CLASS отсутствует в стандарте SQL.

6. DROP OPERATOR FAMILY

DROP OPERATOR FAMILY – удалить семейство операторов.

6.1. Синтаксис

```
DROP OPERATOR FAMILY [ IF EXISTS ] имя USING  
метод_индекса [ CASCADE | RESTRICT ]
```

6.2. Описание

DROP OPERATOR FAMILY удаляет существующее семейство операторов. Выполнить эту команду может только владелец семейства операторов.

DROP OPERATOR FAMILY удаляет также все классы операторов, содержащиеся в семействе, но не удаляет связанные с ним операторы или функции. Если от классов операторов, содержащихся в семействе, зависят какие-либо индексы, семейство будет удалено успешно (вместе с классами и индексами), только если добавить указание CASCADE.

6.3. Параметры

IF EXISTS

Не считать ошибкой, если семейство операторов не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) существующего семейства операторов.

метод_индекса

Имя индексного метода, для которого предназначено это семейство операторов.

CASCADE

Автоматически удалять объекты, зависящие от данного семейства операторов.

RESTRICT

Отказать в удалении семейства операторов, если от него зависят какие-либо объекты. Это поведение по умолчанию.

6.4. Примеры

Удаление семейства операторов B-дерева с именем `float_ops`:

```
DROP OPERATOR FAMILY float_ops USING btree;
```

Эта команда не будет выполнена, если в базе существуют индексы, использующие классы операторов из этого семейства. Чтобы удалить такие индексы вместе с семейством операторов, нужно добавить указание `CASCADE`.

6.5. Совместимость

Команда `DROP OPERATOR FAMILY` отсутствует в стандарте SQL.

7. DROP OWNED

`DROP OWNED` – удалить объекты базы данных, принадлежащие роли.

7.1. Синтаксис

```
DROP OWNED BY { имя | CURRENT_USER | SESSION_USER }  
[, ...] [ CASCADE | RESTRICT ]
```

7.2. Описание

`DROP OWNED` удаляет в текущей базе данных все объекты, принадлежащие любой из указанных ролей. Кроме того, эти роли лишаются всех прав, которые они имели для объектов текущей базы данных и общих объектов (баз данных, табличных пространств).

7.3. Параметры

ИМЯ

Имя роли, все объекты которой будут уничтожены, а права отозваны.

CASCADE

Автоматически удалять объекты, зависящие от затрагиваемых объектов.

RESTRICT

Отказать в удалении объектов, принадлежащих роли, если от каких-либо из них зависят другие объекты в базе данных. Это поведение по умолчанию.

7.4. Замечания

DROP OWNED часто применяется при подготовке к удалению одной или нескольких ролей. Так как команда DROP OWNED затрагивает объекты только в текущей базе данных, обычно её нужно выполнять в каждой базе данных, которая содержит объекты, принадлежащие удаляемой роли.

С указанием CASCADE эта команда может рекурсивно удалить объекты, принадлежащие и другим пользователям.

Команда REASSIGN OWNED предлагает альтернативное решение, меняя владельца всех объектов в базе данных, принадлежащих одной или нескольким заданным ролям.

Базы данных и табличные пространства, принадлежащие указанным ролям, эта команда не удаляет.

7.5. Совместимость

Команда DROP OWNED является расширением Синергия-БД.

8. DROP POLICY

DROP POLICY – удалить политику защиты на уровне строк из таблицы.

8.1. Синтаксис

```
DROP POLICY [ IF EXISTS ] имя ON имя_таблицы [ CASCADE |  
RESTRICT ]
```

8.2. Описание

DROP POLICY удаляет указанную политику из таблицы. Заметьте, что если из таблицы удаляется последняя политика, а в таблице продолжает действовать защита на уровне строк (включенная командой ALTER TABLE), будет применена политика запрета по умолчанию. Отключить защиту на уровне строк для таблицы можно с помощью команды ALTER TABLE с указанием DISABLE ROW SECURITY, независимо от того, определены ли политики для этой таблицы или нет.

8.3. Параметры

IF EXISTS

Не считать ошибкой, если политика не существует. В этом случае будет выдано замечание.

имя

Имя политики, подлежащей удалению.

имя_таблицы

Имя (возможно, дополненное схемой) таблицы, к которой применяется эта политика.

CASCADE

RESTRICT

Эти ключевые слова игнорируются, так как от политик не зависят никакие объекты.

8.4. Примеры

Удаление политики p1 из таблицы my_table:

```
DROP POLICY p1 ON my_table;
```

9. DROP PROCEDURE

DROP PROCEDURE – удалить процедуру.

9.1. Синтаксис

```
DROP PROCEDURE [ IF EXISTS ] имя  
[ ( [ [ режим_аргумента ] [ имя_аргумента ]  
тип_аргумента [, ...] ] ) ] [, ...]  
[ CASCADE | RESTRICT ]
```

9.2. Описание

DROP PROCEDURE удаляет определение существующей процедуры. Пользователь, выполняющий эту команду, должен быть владельцем процедуры. Помимо имени процедуры требуется указать типы её аргументов, так как в базе данных могут существовать несколько процедур с одним именем, но с разными списками аргументов.

9.3. Параметры

IF EXISTS

Не считать ошибкой, если процедура не существует. В этом случае будет выдано замечание.

имя

Имя существующей процедуры (возможно, дополненное схемой). Если список аргументов не указан, имя процедуры должно быть уникальным в её схеме.

режим_аргумента

Режим аргумента: IN или VARIADIC. По умолчанию подразумевается IN.

имя_аргумента

Имя аргумента. Заметьте, что на самом деле DROP PROCEDURE не обращает внимание на имена аргументов, так как для однозначной идентификации процедуры достаточно только типов аргументов.

тип_аргумента

Тип данных аргументов процедуры (возможно, дополненный именем схемы), если таковые имеются.

CASCADE

Автоматически удалять объекты, зависящие от данной процедуры, и, в свою очередь, все зависящие от них объекты.

RESTRICT

Отказать в удалении процедуры, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

9.4. Примеры

```
DROP PROCEDURE do_db_maintenance();
```

9.5. Совместимость

Эта команда соответствует стандарту SQL, но дополнена расширениями Синергия-БД:

- стандарт позволяет удалять с помощью этой команды только одну процедуру;
- указание IF EXISTS – возможность указывать режимы и имена аргументов.

10. DROP PUBLICATION

DROP PUBLICATION – удалить публикацию.

10.1.Синтаксис

```
DROP PUBLICATION [ IF EXISTS ] имя [, ...] [ CASCADE |  
RESTRICT ]
```

10.2.Описание

DROP PUBLICATION удаляет существующую публикацию из базы данных.

Удалить публикацию может только её владелец или суперпользователь.

10.3.Параметры

IF EXISTS

Не считать ошибкой, если публикация не существует. В этом случае будет выдано замечание.

ИМЯ

Имя существующей публикации.

CASCADE

RESTRICT

Эти ключевые слова игнорируются, так как от публикаций не зависят никакие объекты.

10.4.Примеры

Удаление публикации:

```
DROP PUBLICATION mypublication;
```

10.5.Совместимость

DROP PUBLICATION является расширением Синергия-БД.

11. DROP ROLE

DROP ROLE – удалить роль в базе данных.

11.1.Синтаксис

```
DROP ROLE [ IF EXISTS ] имя [, ...]
```


11.2.Описание

DROP ROLE удаляет указанные роли. Удалить роль суперпользователя может только суперпользователь, а чтобы удалить роль обычного пользователя, достаточно иметь право CREATEROLE.

Если на эту роль есть ссылки в какой-либо базе данных в кластере, возникнет ошибка и роль не будет удалена. Прежде чем удалять роль, необходимо удалить все принадлежащие ей объекты (или сменить их владельца), а также лишить её данных ей прав. Для этой цели можно применить команды REASSIGN OWNED и DROP OWNED.

Однако ликвидировать членство в ролях, связанное с этой ролью, не требуется; DROP ROLE автоматически исключит данную роль из других ролей, и третьи роли из данной. Сами роли при этом не удаляются и другим образом никак не затрагиваются.

11.3.Параметры

IF EXISTS

Не считать ошибкой, если роль не существует. В этом случае будет выдано замечание.

ИМЯ

Имя роли, подлежащей удалению.

11.4.Замечания

Синергия-БД включает программу dropuser, которая предоставляет ту же функциональность (на самом деле она вызывает эту команду), но может запускаться в командной оболочке.

11.5.Примеры

Удаление роли:

```
DROP ROLE jonathan;
```

11.6.Совместимость

В стандарте SQL определена команда DROP ROLE, но она может удалять

только по одной роли, а для её выполнения требуются другие права, не такие как в Синергия-БД.

12. DROP ROUTINE

DROP ROUTINE – удалить подпрограмму.

12.1. Синтаксис

```
DROP ROUTINE [ IF EXISTS ] имя [ ( [ [ режим_аргумента ]  
[ имя_аргумента ] тип_аргумента [, ...] ] ) ] [, ...]  
[ CASCADE | RESTRICT ]
```

12.2. Описание

DROP ROUTINE удаляет определение существующей подпрограммы, которой может быть обычная функция, агрегат или процедура.

12.3. Примеры

Удаление подпрограммы `foo` для типа `integer`:

```
DROP ROUTINE foo(integer);
```

Эта команда будет работать независимо от того, является ли `foo` агрегатом, функцией или процедурой.

12.4. Совместимость

Эта команда соответствует стандарту SQL, но дополнена следующими расширениями Синергия-БД:

- стандарт позволяет удалять с помощью этой команды только одну подпрограмму;
- указание `IF EXISTS` – возможность указывать режимы и имена аргументов;
- поддержка агрегатных функций.

13. DROP RULE

DROP RULE – удалить правило перезаписи.

13.1. Синтаксис

```
DROP RULE [ IF EXISTS ] имя ON имя_таблицы [ CASCADE |
```

RESTRICT]

13.2.Описание

DROP RULE удаляет правило перезаписи.

13.3.Параметры

IF EXISTS

Не считать ошибкой, если правило не существует. В этом случае будет выдано замечание.

имя

Имя правила, подлежащего удалению.

table_name

Имя (возможно, дополненное схемой) существующей таблицы (или представления), к которой применяется это правило.

CASCADE

Автоматически удалять объекты, зависящие от данного правила.

RESTRICT

Отказать в удалении правила, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примечание. С указанием CASCADE эта команда может удалить объекты не только в данной схеме, но и в других.

13.4.Примеры

Удаление правила перезаписи newrule:

```
DROP RULE newrule ON mytable;
```

13.5.Совместимость

DROP RULE является языковым расширением Синергия-БД, как и вся система перезаписи запросов.

14. DROP SCHEMA

DROP SCHEMA – удалить схему.

14.1. Синтаксис

```
DROP SCHEMA [ IF EXISTS ] имя [, ...] [ CASCADE |  
RESTRICT ]
```

14.2. Описание

DROP SCHEMA удаляет схемы из базы данных.

Схему может удалить только её владелец или суперпользователь. Заметьте, что владелец может удалить схему (вместе со всеми содержащимися в ней объектами), даже если он не владеет некоторыми объектами в своей схеме.

14.3. Параметры

IF EXISTS

Не считать ошибкой, если схема не существует. В этом случае будет выдано замечание.

имя

Имя схемы.

CASCADE

Автоматически удалять объекты (таблицы, функции и т.п.), содержащиеся в схеме.

RESTRICT

Отказать в удалении схемы, если она содержит какие-либо объекты. Это поведение по умолчанию.

14.4. Примеры

Удаление схемы mystuff из базы данных вместе со всем, что в ней содержится:

```
DROP SCHEMA mystuff CASCADE;
```

14.5. Совместимость

Команда DROP SCHEMA полностью соответствует стандарту SQL, но возможность удалять в одной команде несколько схем и указание IF EXISTS являются расширениями Синергия-БД.

15. DROP SEQUENCE

DROP SEQUENCE – удалить последовательность.

15.1.Синтаксис

```
DROP SEQUENCE [ IF EXISTS ] имя [, ...] [ CASCADE |  
RESTRICT ]
```

15.2.Описание

DROP SEQUENCE удаляет генераторы числовых последовательностей.

Удалить последовательность может только её владелец или суперпользователь.

15.3.Параметры

IF EXISTS

Не считать ошибкой, если последовательность не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) последовательности.

CASCADE

Автоматически удалять объекты, зависящие от данной последовательности.

RESTRICT

Отказать в удалении последовательности, если от нее зависят какие-либо объекты. Это поведение по умолчанию.

15.4.Примеры

Удаление последовательности serial:

```
DROP SEQUENCE serial;
```

15.5.Совместимость

DROP SEQUENCE соответствует стандарту SQL, но возможность удалять в одной команде несколько последовательностей и указание IF EXISTS являются расширениями Синергия-БД.

16. DROP SERVER

DROP SERVER – удалить описание стороннего сервера.

16.1.Синтаксис

`DROP SERVER [ IF EXISTS ] имя [ CASCADE | RESTRICT ]`

16.2.Описание

`DROP SERVER` удаляет существующее описание стороннего сервера. Пользователь, выполняющий эту команду, должен быть владельцем сервера.

16.3.Параметры

`IF EXISTS`

Не считать ошибкой, если сервер не существует. В этом случае будет выдано замечание.

`имя`

Имя существующего сервера.

`CASCADE`

Автоматически удалять объекты, зависящие от данного сервера (например, сопоставления пользователей).

`RESTRICT`

Отказаться в удалении сервера, если от него зависят какие-либо объекты. Это поведение по умолчанию.

16.4.Примеры

Удаление определения сервера `foo`, если оно существует:

`DROP SERVER IF EXISTS foo;`

16.5.Совместимость

`DROP SERVER` соответствует стандарту ISO/IEC 9075-9 (SQL/MED). Предложение `IF EXISTS` является расширением Синергия-БД.

17. DROP STATISTICS

`DROP STATISTICS` – удалить расширенную статистику.

17.1.Синтаксис

`DROP STATISTICS [ IF EXISTS ] имя [, ...]`

17.2.Описание

DROP STATISTICS удаляет объект(ы) статистики из базы данных. Удалить объект статистики может только владелец объекта, владелец схемы или суперпользователь.

17.3.Параметры

IF EXISTS

Не считать ошибкой, если объект статистики не существует. В этом случае будет выдано замечание.

ИМЯ

Имя (возможно, дополненное схемой) объекта статистики, подлежащего удалению.

17.4.Примеры

Удаление двух объектов статистики в разных схемах (если эти объекты отсутствуют, ошибки не будет):

```
DROP STATISTICS IF EXISTS
    accounting.users_uid_creation,
    public.grants_user_role;
```

17.5.Совместимость

Оператор DROP STATISTICS отсутствует в стандарте SQL.

18. DROP SUBSCRIPTION

DROP SUBSCRIPTION – удалить подписку.

18.1.Синтаксис

```
DROP SUBSCRIPTION [ IF EXISTS ] имя [ CASCADE |
RESTRICT ]
```

18.2.Описание

DROP SUBSCRIPTION удаляет подписку из кластера баз данных.

Удалить подписку может только суперпользователь.

Команду DROP SUBSCRIPTION нельзя выполнять в блоке транзакции, если

подписка связана со слотом репликации. Для освобождения слота можно использовать ALTER SUBSCRIPTION.

18.3.Параметры

ИМЯ

Имя подписки, подлежащей удалению.

CASCADE

RESTRICT

Эти ключевые слова игнорируются, так как от подписок не зависят никакие объекты.

18.4.Замечания

При удалении подписки, связанной со слотом репликации на удалённом узле (это типичная ситуация), команда DROP SUBSCRIPTION подключится к удалённому узлу и попытается удалить слот репликации в ходе этой операции. Это необходимо для освобождения ресурсов, выделенных для подписки на удалённом узле. Если при этом происходит сбой, либо из-за недоступности удалённого узла, либо из-за ошибки при удалении слота репликации, либо вообще из-за его отсутствия, команда DROP SUBSCRIPTION прерывается. Для разрешения этой ситуации разорвите связь подписки с этим слотом репликации, выполнив команду ALTER SUBSCRIPTION ... SET (slot_name = NONE). После этого команда DROP SUBSCRIPTION не будет пытаться выполнять какие-либо действия на удалённом узле. Заметьте, что если удалённый слот репликации фактически продолжает существовать, его нужно будет удалить вручную; в противном случае для него будет по-прежнему сохраняться WAL, что в конце концов может привести к переполнению диска.

Если подписка связана со слотом репликации, команду DROP SUBSCRIPTION нельзя выполнять внутри блока транзакции.

18.5.Примеры

Удаление подписки:

```
DROP SUBSCRIPTION mysub;
```


18.6.Совместимость

DROP SUBSCRIPTION является расширением Синергия-БД.

19. DROP TABLE

DROP TABLE – удалить таблицу.

19.1.Синтаксис

```
DROP TABLE [ IF EXISTS ] имя [, ...] [ CASCADE |  
RESTRICT ]
```

19.2.Описание

DROP TABLE удаляет таблицы из базы данных. Удалить таблицу может только её владелец, владелец схемы или суперпользователь. Чтобы опустошить таблицу, не удаляя её саму, вместо этой команды следует использовать DELETE или TRUNCATE.

DROP TABLE всегда удаляет все индексы, правила, триггеры и ограничения, существующие в целевой таблице. Однако, чтобы удалить таблицу, на которую ссылается представление или ограничение внешнего ключа в другой таблице, необходимо дополнительно указать CASCADE. С указанием CASCADE зависимое представление будет удалено полностью, но в случае с первичным ключом удалено будет только само ограничение, а не таблица, к которой оно относится.

19.3.Параметры

IF EXISTS

Не считать ошибкой, если таблица не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) таблицы, подлежащей удалению.

CASCADE

Автоматически удалять объекты, зависящие от данной таблицы (например, представления).

RESTRICT

Отказать в удалении таблицы, если от нее зависят какие-либо объекты. Это поведение по умолчанию.

19.4.Примеры

Удаление таблиц `films` и `distributors`:

```
DROP TABLE films, distributors;
```

19.5.Совместимость

Эта команда соответствует стандарту SQL, но возможность удалять в одной команде несколько таблиц и указание `IF EXISTS` являются расширениями Синергия-БД.

20. DROP TABLESPACE

`DROP TABLESPACE` – удалить табличное пространство.

20.1.Синтаксис

```
DROP TABLESPACE [ IF EXISTS ] имя
```

20.2.Описание

`DROP TABLESPACE` удаляет табличное пространство из системы.

Удалить табличное пространство может только его владелец или суперпользователь. Перед удалением его необходимо очистить от всех объектов базы данных. Даже если в текущей базе данных не будет ни одного объекта, находящегося в этом пространстве, в нем вполне могут оставаться объекты других баз данных. Кроме того, если табличное пространство указано в списке `temp_tablespaces` любого активного сеанса, команда `DROP` может завершиться ошибкой, если в этом пространстве окажутся временные файлы.

20.3.Параметры

`IF EXISTS`

Не считать ошибкой, если табличное пространство не существует. В этом случае будет выдано замечание.

имя

Имя табличного пространства.

20.4.Замечания

DROP TABLESPACE не может быть выполнена в блоке транзакции.

20.5.Примеры

Удаление табличного пространства mystuff из системы:

```
DROP TABLESPACE mystuff;
```

20.6.Совместимость

DROP TABLESPACE является расширением Синергия-БД.

21. DROP TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH CONFIGURATION – удалить конфигурацию текстового поиска.

21.1.Синтаксис

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] имя  
[ CASCADE | RESTRICT ]
```

21.2.Описание

DROP TEXT SEARCH CONFIGURATION удаляет существующую конфигурацию текстового поиска. Выполнить эту команду может только владелец конфигурации.

21.3.Параметры

IF EXISTS

Не считать ошибкой, если конфигурация текстового поиска не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) существующей конфигурации текстового поиска.

CASCADE

Автоматически удалять объекты, зависящие от данной конфигурации текстового поиска.

RESTRICT

Отказать в удалении конфигурации текстового поиска, если от нее зависят какие-либо объекты. Это поведение по умолчанию.

21.4.Примеры

Удаление конфигурации текстового поиска my_english:

```
DROP TEXT SEARCH CONFIGURATION my_english;
```

Эта команда не будет выполнена, если какие-либо индексы ссылаются на эту конфигурацию в вызовах to_tsvector. Чтобы удалить такие индексы вместе с конфигурацией, следует добавить указание CASCADE.

21.5.Совместимость

Оператор DROP TEXT SEARCH CONFIGURATION отсутствует в стандарте SQL.

22. DROP TEXT SEARCH DICTIONARY

DROP TEXT SEARCH DICTIONARY – удалить словарь текстового поиска.

22.1.Синтаксис

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] имя [ CASCADE  
| RESTRICT ]
```

22.2.Описание

DROP TEXT SEARCH DICTIONARY удаляет существующий словарь текстового поиска. Выполнить эту команду может только владелец словаря.

22.3.Параметры

IF EXISTS

Не считать ошибкой, если словарь текстового поиска не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) существующего словаря текстового поиска.

CASCADE

Автоматически удалять объекты, зависящие от данного словаря текстового поиска.

RESTRICT

Отказать в удалении словаря текстового поиска, если от него зависят какие-либо объекты. Это поведение по умолчанию.

22.4.Примеры

Удалить словарь текстового поиска english:

```
DROP TEXT SEARCH DICTIONARY english;
```

Эта команда не будет выполнена, если существуют конфигурации текстового поиска, использующие этот словарь. Чтобы удалить такие конфигурации вместе со словарем, следует добавить указание CASCADE.

22.5.Совместимость

Оператор DROP TEXT SEARCH DICTIONARY отсутствует в стандарте SQL.

23. DROP TEXT SEARCH PARSER

DROP TEXT SEARCH PARSER – удалить анализатор текстового поиска.

23.1.Синтаксис

```
DROP TEXT SEARCH PARSER [ IF EXISTS ] имя [ CASCADE |  
RESTRICT ]
```

23.2.Описание

DROP TEXT SEARCH PARSER удаляет существующий анализатор текстового поиска. Выполнить эту команду может только суперпользователь.

23.3.Параметры

IF EXISTS

Не считать ошибкой, если анализатор текстового поиска не существует. В этом случае будет выдано замечание.

ИМЯ

Имя (возможно, дополненное схемой) существующего анализатора текстового

поиска.

CASCADE

Автоматически удалять объекты, зависящие от данного анализатора текстового поиска.

RESTRICT

Отказаться в удалении анализатора текстового поиска, если от него зависят какие-либо объекты. Это поведение по умолчанию.

23.4.Примеры

Удаление анализатора текстового поиска `my_parser`:

```
DROP TEXT SEARCH PARSEr my_parser;
```

Эта команда не будет выполнена, если существуют конфигурации текстового поиска, использующие этот анализатор. Чтобы удалить такие конфигурации вместе с анализатором, следует добавить указание `CASCADE`.

23.5.Совместимость

Оператор `DROP TEXT SEARCH PARSEr` отсутствует в стандарте SQL.

24. DROP TEXT SEARCH TEMPLATE

`DROP TEXT SEARCH TEMPLATE` – удалить шаблон текстового поиска

24.1.Синтаксис

```
DROP TEXT SEARCH TEMPLATE [ IF EXISTS ] имя [ CASCADE |  
RESTRICT ]
```

24.2.Описание

`DROP TEXT SEARCH TEMPLATE` удаляет существующий шаблон текстового поиска. Выполнить эту команду может только суперпользователь.

24.3.Параметры

`IF EXISTS`

Не считать ошибкой, если шаблон текстового поиска не существует. В этом случае будет выдано замечание.

ИМЯ

Имя (возможно, дополненное схемой) существующего шаблона текстового поиска.

CASCADE

Автоматически удалять объекты, зависящие от данного шаблона текстового поиска.

RESTRICT

Отказаться в удалении шаблона текстового поиска, если от него зависят какие-либо объекты. Это поведение по умолчанию.

24.4.Примеры

Удаление шаблона текстового поиска thesaurus:

```
DROP TEXT SEARCH TEMPLATE thesaurus;
```

Эта команда не будет выполнена, если существуют словари текстового поиска, использующие этот шаблон. Чтобы удалить такие словари вместе с шаблоном, следует добавить указание CASCADE.

24.5.Совместимость

Оператор DROP TEXT SEARCH TEMPLATE отсутствует в стандарте SQL.

25. DROP TRANSFORM

DROP TRANSFORM – удалить трансформацию.

25.1.Синтаксис

```
DROP TRANSFORM [ IF EXISTS ] FOR имя_типа LANGUAGE  
имя_языка [ CASCADE | RESTRICT ]
```

25.2.Описание

DROP TRANSFORM удаляет ранее определённую трансформацию.

Чтобы удалить трансформацию, необходимо быть владельцем типа и языка. Такие же требования действуют и при создании трансформации.

25.3.Параметры

IF EXISTS

Не считать ошибкой, если трансформация не существует. В этом случае будет выдано замечание.

имя_типа

Имя типа данных, для которого предназначена трансформация.

имя_языка

Имя языка, для которого предназначена трансформация.

CASCADE

Автоматически удалять объекты, зависящие от данной трансформации, и, в свою очередь, все зависящие от них объекты.

RESTRICT

Отказать в удалении трансформации, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

25.4.Примеры

Удаление трансформации для типа `hstore` и языка `plpythonu`:

```
DROP TRANSFORM FOR hstore LANGUAGE plpythonu;
```

25.5.Совместимость

Эта форма `DROP TRANSFORM` является расширением Синергия-БД.

26. DROP TRIGGER

`DROP TRIGGER` – удалить триггер.

26.1.Синтаксис

```
DROP TRIGGER [ IF EXISTS ] имя ON имя_таблицы [ CASCADE  
| RESTRICT ]
```

26.2.Описание

`DROP TRIGGER` удаляет существующее определение триггера. Пользователь, выполняющий эту команду, должен быть владельцем таблицы, для которой определен данный триггер.

26.3.Параметры

`IF EXISTS`

Не считать ошибкой, если триггер не существует. В этом случае будет выдано замечание.

имя

Имя триггера, подлежащего удалению.

имя_таблицы

Имя (возможно, дополненное схемой) таблицы, для которой определен триггер.

CASCADE

Автоматически удалять объекты, зависящие от данного триггера.

RESTRICT

Отказать в удалении триггера, если от него зависят какие-либо объекты. Это поведение по умолчанию.

26.4.Примеры

Удаление триггера `if_dist_exists` в таблице `films`:

```
DROP TRIGGER if_dist_exists ON films;
```

26.5.Совместимость

Оператор `DROP TRIGGER` в Синергия-БД несовместим со стандартом SQL. В стандарте имена триггеров не считаются локальными по отношению к таблицам, так что синтаксис команды проще: `DROP TRIGGER имя`.

27. DROP TYPE

`DROP TYPE` – удалить тип данных.

27.1.Синтаксис

```
DROP TYPE [ IF EXISTS ] имя [, ...] [ CASCADE |  
RESTRICT ]
```

27.2.Описание

`DROP TYPE` удаляет определенный пользователем тип данных. Удалить тип может только его владелец.

27.3.Параметры

IF EXISTS

Не считать ошибкой, если тип не существует. В этом случае будет выдано замечание.

ИМЯ

Имя (возможно, дополненное схемой) типа данных, подлежащего удалению.

CASCADE

Автоматически удалять объекты, зависящие от данного типа (например, колонки таблиц, функции, операторы).

RESTRICT

Отказать в удалении типа, если от него зависят какие-либо объекты. Это поведение по умолчанию.

27.4.Примеры

Удаление типа данных box:

```
DROP TYPE box;
```

27.5.Совместимость

Эта команда подобна соответствующей команде в стандарте SQL, но указание IF EXISTS является расширением Синергия-БД. Однако учтите, что команда CREATE TYPE и механизм расширения типов в Синергия-БД значительно отличаются от стандарта SQL.

28. DROP USER

DROP USER – просто альтернативное написание команды DROP ROLE.

28.1.Синтаксис

```
DROP USER [ IF EXISTS ] имя [, ...]
```

28.2.Описание

DROP USER теперь является синонимом DROP PROCEDURE.

28.3.Совместимость

DROP USER является расширением Синергия-БД. В стандарте SQL определение пользователей считается зависимым от реализации.

29. DROP USER MAPPING

DROP USER MAPPING – удалить сопоставление пользователя для стороннего сервера.

29.1.Синтаксис

```
DROP USER MAPPING [ IF EXISTS ] FOR { имя_пользователя |  
USER | CURRENT_USER | PUBLIC } SERVER имя_сервера
```

29.2.Описание

DROP USER MAPPING удаляет существующее сопоставление пользователя для стороннего сервера.

Владелец стороннего сервера может удалить заданные для этого сервера сопоставления любых пользователей. Кроме того, обычный пользователь может удалить сопоставление для собственного имени, если он имеет право USAGE для сервера.

29.3.Параметры

IF EXISTS

Не считать ошибкой, если сопоставление не существует. В этом случае будет выдано замечание.

имя_пользователя

Имя пользователя для сопоставления. Значения CURRENT_USER и USER соответствуют имени текущего пользователя. Значение PUBLIC соответствует именам всех текущих и будущих пользователей системы.

имя_сервера

Имя сервера, для которого меняется сопоставление пользователей.

29.4.Примеры

Удаление сопоставления пользователя bob для сервера foo, если оно

существует:

```
DROP USER MAPPING IF EXISTS FOR bob SERVER foo;
```

29.5.Совместимость

DROP USER MAPPING соответствует стандарту ISO/IEC 9075-9 (SQL/MED).
Предложение IF EXISTS является расширением Синергия-БД.

30. DROP VIEW

DROP VIEW – удалить представление.

30.1.Синтаксис

```
DROP VIEW [ IF EXISTS ] имя [, ...] [ CASCADE |  
RESTRICT ]
```

30.2.Описание

DROP VIEW удаляет существующее представление. Выполнить эту команду может только владелец представления.

30.3.Параметры

IF EXISTS

Не считать ошибкой, если представление не существует. В этом случае будет выдано замечание.

ИМЯ

Имя (возможно, дополненное схемой) представления, подлежащего удалению.

CASCADE

Автоматически удалять объекты, зависящие от данного представления (например, другие представления).

RESTRICT

Отказаться в удалении представления, если от него зависят какие-либо объекты. Это поведение по умолчанию.

30.4.Примеры

Эта команда удаляет представление с именем kinds:

```
DROP VIEW kinds;
```

30.5.Совместимость

Эта команда соответствует стандарту SQL, но возможность удалять в одной команде несколько представлений и указание IF EXISTS являются расширениями Синергия-БД.

31. END

END – зафиксировать текущую транзакцию.

31.1.Синтаксис

END [WORK | TRANSACTION]

31.2.Описание

END фиксирует текущую транзакцию. Все изменения, произведенные этой транзакцией, становятся видимыми для других и гарантированно сохраняются в случае сбоя. Эта команда является расширением Синергия-БД и равнозначна команде COMMIT.

31.3.Параметры

WORK
TRANSACTION

Необязательные ключевые слова, не оказывают никакого влияния.

31.4.Замечания

Для прерывания транзакции используйте ROLLBACK.

При попытке выполнить END вне транзакции ничего не произойдет, но будет выдано предупреждение.

31.5.Примеры

Следующая команда фиксирует текущую транзакцию и сохраняет все изменения:

END;

31.6.Совместимость

END является расширением Синергия-БД и выполняет ту же функцию, что и

оператор COMMIT, описанный в стандарте SQL.

32. EXECUTE

EXECUTE – выполнить подготовленный оператор.

32.1.Синтаксис

EXECUTE имя [(параметр [, ...])]

32.2.Описание

EXECUTE выполняет подготовленный ранее оператор. Так как подготовленные операторы существуют только в рамках сеанса, они должны создаваться командой PREPARE, выполненной в текущем сеансе ранее.

Если команда PREPARE, создающая оператор, определяет некоторый набор параметров, команде EXECUTE должны быть переданы подходящие значения этих параметров; в противном случае возникнет ошибка. Заметьте, что подготовленные операторы (в отличие от функций) не перегружаются в зависимости от типа или числа параметров; имя подготовленного оператора должно быть уникальным в рамках текущего сеанса.

Чтобы узнать больше о создании и использовании подготовленных операторов, обратитесь к PREPARE.

32.3.Параметры

имя

Имя подготовленного оператора, который будет выполнен.

параметр

Фактическое значение параметра подготовленного оператора. Это может быть выражение, выдающее значение, совместимое с типом данных этого параметра, который был определен при создании подготовленного оператора.

32.4.Выводимая информация

Метка команды, возвращаемая EXECUTE, соответствует подготовленному оператору, а не оператору EXECUTE.

32.5.Совместимость

В стандарте SQL есть оператор EXECUTE, но он предназначен только для применения во встраиваемом SQL. Эта версия оператора EXECUTE имеет также несколько другой синтаксис.

33. EXPLAIN

EXPLAIN – показать план выполнения оператора.

33.1.Синтаксис

EXPLAIN [(параметр [, ...])] оператор
EXPLAIN [ANALYZE] [VERBOSE] оператор

Здесь допускается параметр:

```
ANALYZE [ boolean ]  
VERBOSE [ boolean ]  
COSTS [ boolean ]  
BUFFERS [ boolean ]  
TIMING [ boolean ]  
FORMAT { TEXT | XML | JSON | YAML }
```

33.2.Описание

Эта команда выводит план выполнения, генерируемый планировщиком Синергия-БД для заданного оператора. План выполнения показывает, как будут сканироваться таблицы, затрагиваемые оператором – просто последовательно, по индексу и т.д. – а если запрос связывает несколько таблиц, какой алгоритм соединения будет выбран для объединения считанных из них строк.

Наибольший интерес в выводимой информации представляет ожидаемая стоимость выполнения оператора, которая показывает, сколько, по мнению планировщика, будет выполняться этот оператор (это значение измеряется в единицах стоимости, которые не имеют точного определения, но обычно это обращение к странице на диске). Фактически выводятся два числа: стоимость запуска до выдачи первой строки и общая стоимость выдачи всех строк. Для большинства запросов важна общая стоимость, но в таких контекстах, как подзапрос

в EXISTS, планировщик будет минимизировать стоимость запуска, а не общую стоимость (так как исполнение запроса все равно завершится сразу после получения одной строки). Кроме того, если количество возвращаемых строк ограничивается предложением LIMIT, планировщик интерполирует стоимость между двумя этими числами, выбирая наиболее выгодный план.

С параметром ANALYZE оператор будет выполнен на самом деле, а не только запланирован. При этом в вывод добавляются фактические сведения о времени выполнения, включая общее время, затраченное на каждый узел плана (в миллисекундах) и общее число строк, выданных в результате. Это помогает понять, насколько близки к реальности предварительные оценки планировщика.

Примечание. Имейте в виду, что с указанием ANALYZE оператор действительно выполняется. Хотя EXPLAIN отбрасывает результат, который вернул бы SELECT, в остальном все действия выполняются как обычно. Если вы хотите выполнить EXPLAIN ANALYZE с командой INSERT, UPDATE, DELETE, CREATE TABLE AS или EXECUTE, не допуская изменения данных этой командой, воспользуйтесь таким приемом:

```
BEGIN;  
EXPLAIN ANALYZE ...;  
ROLLBACK;
```

Без скобок для этого оператора можно указать только параметры ANALYZE и VERBOSE и только в таком порядке.

33.3.Параметры

ANALYZE

Выполнить команду и вывести фактическое время выполнения и другую статистику. По умолчанию этот параметр равен FALSE.

VERBOSE

Вывести дополнительную информацию о плане запроса. В частности, включить список колонок результата для каждого узла в дереве плана, дополнить схемой имена таблиц и функций, всегда указывать для переменных в выражениях псевдоним их таблицы, а также выводить имена всех триггеров, для которых выдается статистика. По умолчанию этот параметр равен FALSE.

COSTS

Вывести рассчитанную стоимость запуска и общую стоимость каждого узла плана, а также рассчитанное число строк и ширину каждой строки. Этот параметр по умолчанию равен TRUE.

BUFFERS

Включить информацию об использовании буфера. В частности, вывести число попаданий, блоков прочитанных, загрязненных и записанных в разделяемом и локальном буфере, а также число прочитанных и записанных временных блоков. Попаданием считается ситуация, когда требуемый блок уже находится в кеше и чтения с диска удастся избежать. Блоки в общем буфере содержат данные обычных таблиц и индексов, в локальном – данные временных таблиц и индексов, а временные блоки предназначены для краткосрочного использования при выполнении сортировки, хеширования, материализации и подобных узлов плана. Число загрязненных блоков показывает, сколько ранее не модифицированных блоков изменила данная операция; тогда как число записанных блоков показывает, сколько ранее загрязненных блоков данный серверный процесс вынес из кеша при обработке запроса. Значения, указываемые для узла верхнего уровня, включают значения всех его дочерних узлов. В текстовом формате выводятся только ненулевые значения. Этот параметр действует только в режиме ANALYZE. По умолчанию его значение равно FALSE.

TIMING

Включить в вывод фактическое время запуска и время, затраченное на каждый узел. Постоянное чтение системных часов может значительно замедлить запрос, так что если достаточно знать фактическое число строк, имеет смысл сделать этот параметр равным FALSE. Время выполнения всего оператора замеряется всегда, даже когда этот параметр выключен и на уровне узлов время не подсчитывается. Этот параметр действует только в режиме ANALYZE. По умолчанию его значение равно TRUE.

SUMMARY

Включить сводку (например, суммарное время) после плана запроса. Сводка включается по умолчанию, когда используется ANALYZE, но этот параметр позволяет получить её и с другими вариантами команды. Время планирования в

EXPLAIN EXECUTE включает время извлечения плана из кеша и время перепланирования, если оно потребовалось.

FORMAT

Установить один из следующих форматов вывода: TEXT, XML, JSON или YAML. Последние три формата содержат ту же информацию, что и текстовый, но больше подходят для программного разбора. По умолчанию выбирается формат TEXT.

boolean

Включает или отключает заданный параметр. Для включения параметра можно написать TRUE, ON или 1, а для отключения – FALSE, OFF или 0. Значение boolean можно опустить, в этом случае подразумевается TRUE.

оператор

Любой оператор SELECT, INSERT, UPDATE, DELETE, VALUES, EXECUTE, DECLARE, CREATE TABLE AS и CREATE MATERIALIZED VIEW AS, план выполнения которого вас интересует.

33.4.Выводимая информация

Результатом команды будет текстовое описание плана, выбранного для оператора, возможно, дополненное статистикой выполнения.

33.5.Замечания

Чтобы планировщик запросов Синергия-БД был достаточно информирован для эффективной оптимизации запросов, данные в pg_statistic должны быть актуальными для всех таблиц, задействованных в запросе. Обычно об этом автоматически заботится демон автоочистки. Но если в таблице недавно произошли значительные изменения, может потребоваться вручную выполнить ANALYZE, не дожидаясь, пока автоочистка обработает эти изменения.

Измеряя фактическую стоимость выполнения каждого узла в плане, текущая реализация EXPLAIN ANALYZE приносит накладные расходы профилирования в выполнение запроса. В результате этого, при запуске запроса командой EXPLAIN ANALYZE он может выполняться значительно дольше, чем при обычном

выполнении. Объем накладных расходов зависит от природы запроса, а также от используемой платформы. Худшая ситуация наблюдается для узлов плана, которые сами по себе выполняются очень быстро, и в операционных системах, где получение текущего времени относительно длительная операция.

33.6.Примеры

Получение плана простого запроса для таблицы, содержащей единственную колонку типа `integer`, с 10000 строк:

```
EXPLAIN SELECT * FROM foo;
```

QUERY PLAN

```
-----
-
Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

План того же запроса, но выведенный в формате JSON:

```
EXPLAIN (FORMAT JSON) SELECT * FROM foo;
```

QUERY PLAN

```
-----
[
  {
    «Plan»: {
      «Node Type»: «Seq Scan»,
      «Relation Name»: «foo»,
      «Alias»: «foo»,
      «Startup Cost»: 0.00,
      «Total Cost»: 155.00,
      «Plan Rows»: 10000,
      «Plan Width»: 4
    }
  }
]
(1 row)
```

Если в таблице есть индекс, а в запросе присутствует условие WHERE, для которого полезен этот индекс, EXPLAIN может показать другой план:

```
EXPLAIN SELECT * FROM foo WHERE i = 4;
```

QUERY PLAN

```
-----
Index Scan using fi on foo (cost=0.00..5.98 rows=1
width=4)
```

```
    Index Cond: (i = 4)
(2 rows)
```

План того же запроса, но в формате YAML:

```
EXPLAIN (FORMAT YAML) SELECT * FROM foo WHERE i='4';
```

QUERY PLAN

```
-----
- Plan:                                     +
  Node Type: «Index Scan»                  +
  Scan Direction: «Forward»                +
  Index Name: «fi»                         +
  Relation Name: «foo»                     +
  Alias: «foo»                             +
  Startup Cost: 0.00                       +
  Total Cost: 5.98                         +
  Plan Rows: 1                             +
  Plan Width: 4                            +
  Index Cond: «(i = 4)»                    +
(1 row)
```

Рассмотрение формата XML оставлено в качестве упражнения для читателя.

План того же запроса без вывода оценок стоимости:

```
EXPLAIN (COSTS FALSE) SELECT * FROM foo WHERE i = 4;
```

QUERY PLAN

```
-----
Index Scan using fi on foo
```

Index Cond: (i = 4)

(2 rows)

Пример плана для запроса с агрегатной функцией:

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;
```

QUERY PLAN

```
-----
Aggregate  (cost=23.93..23.93 rows=1 width=4)
  -> Index Scan using fi on foo  (cost=0.00..23.92
rows=6 width=4)
        Index Cond: (i < 10)
(3 rows)
```

Пример использования EXPLAIN EXECUTE для отображения плана выполнения подготовленного запроса:

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
WHERE id > $1 AND id < $2
GROUP BY foo;
```

```
EXPLAIN ANALYZE EXECUTE query(100, 200);
```

QUERY PLAN

```
-----
HashAggregate  (cost=9.54..9.54 rows=1 width=8) (actual
time=0.156..0.161 rows=11 loops=1)
  Group Key: foo
  -> Index Scan using test_pkey on test
(cost=0.29..9.29 rows=50 width=8) (actual
time=0.039..0.091 rows=99 loops=1)
        Index Cond: ((id > $1) AND (id < $2))
Planning time: 0.197 ms
Execution time: 0.225 ms
(6 rows)
```

Разумеется, конкретные числа, показанные здесь, зависят от фактического содержимого задействованных таблиц. Кроме того, команда ANALYZE при обработке статистических данных производит случайные выборки, так что оценки стоимости могут меняться при каждом чистом запуске ANALYZE, даже когда фактическое распределение данных в таблице не меняется.

33.7.Совместимость

Оператор EXPLAIN отсутствует в стандарте SQL.

34. FETCH

FETCH – получить результат запроса через курсор.

34.1.Синтаксис

```
FETCH [ direction [ FROM | IN ] ] cursor_name
```

Здесь `direction` может быть пустым или принимать следующее значение:

```
NEXT
PRIOR
FIRST
LAST
ABSOLUTE число
RELATIVE число
число
ALL
FORWARD
FORWARD число
FORWARD ALL
BACKWARD
BACKWARD число
BACKWARD ALL
```

34.2.Описание

FETCH получает строки через ранее созданный курсор.

Курсор связан с определенным положением, что и использует команда FETCH.

Курсор может располагаться перед первой строкой результата запроса, на любой строке этого результата, либо после последней строки. При создании курсор оказывается перед первой строкой. Когда FETCH доходит до конца набора строк, курсор остается в положении после последней строки, либо перед первой, при движении назад. После команд FETCH ALL и FETCH BACKWARD ALL курсор всегда оказывается после последней строки или перед первой, соответственно.

Формы NEXT, PRIOR, FIRST, LAST, ABSOLUTE и RELATIVE выбирают одну строку после соответствующего перемещения курсора. Если в этом положении строки не оказывается, возвращается пустой результат, а курсор остается в достигнутом положении перед первой строкой или после последней.

Формы FORWARD и BACKWARD получают указанное число строк, сдвигаясь соответственно вперед или назад; в результате курсор оказывается на последней выданной строке (или перед/после всех строк, если число превышает количество доступных строк).

Формы RELATIVE 0, FORWARD 0 и BACKWARD 0 действуют одинаково – они считывают текущую строку, не перемещая курсор, то есть, повторно выбирая строку, выбранную последней. Эта операция будет успешна, только если курсор не расположен до первой или после последней строки; в этом случае строка возвращена не будет.

Примечание. На этой странице описывается применение курсоров на уровне команд SQL. Если вы попытаетесь использовать курсоры внутри функции PL/pgSQL, правила будут другими.

34.3.Параметры

direction

Параметр направление задает направление движения и число выбираемых строк. Он может принимать одно из следующих значений:

NEXT

Выбрать следующую строку. Это действие подразумевается по умолчанию, если направление опущено.

PRIOR

Выбрать предыдущую строку.

FIRST

Выбрать первую строку запроса (аналогично указанию ABSOLUTE 1).

LAST

Выбрать последнюю строку запроса (аналогично ABSOLUTE -1).

ABSOLUTE число

Выбрать строку под номером число с начала, либо под номером abs (число) с конца, если число отрицательно. Если число выходит за границы набора строк, курсор размещается перед первой или после последней строки; в частности, с ABSOLUTE 0 курсор оказывается перед первой строкой.

RELATIVE число

Выбрать строку под номером число, считая со следующей вперед, либо под номером abs (число), считая с предыдущей назад, если число отрицательно. RELATIVE 0 повторно считывает текущую строку, если таковая имеется.

число

Выбрать следующее число строк (аналогично FORWARD число).

ALL

Выбрать все оставшиеся строки (аналогично FORWARD ALL).

FORWARD

Выбрать следующую строку (аналогично NEXT).

FORWARD число

Выбрать следующее число строк. FORWARD 0 повторно выбирает текущую строку.

FORWARD ALL

Выбрать все оставшиеся строки.

BACKWARD

Выбрать предыдущую строку (аналогично PRIOR).

BACKWARD число

Выбрать предыдущее число строк (с перемещением назад). BACKWARD 0 повторно выбирает текущую строку.

BACKWARD ALL

Выбрать все предыдущие строки (с перемещением назад).

число

Здесь число – целочисленная константа, возможно со знаком, определяющая смещение или количество выбираемых строк. Для вариантов FORWARD и BACKWARD указание отрицательного числа равнозначно смене направления FORWARD на BACKWARD и наоборот.

имя_курсора

Имя открытого курсора.

34.4.Выводимая информация

При успешном выполнении FETCH возвращает метку команды вида

FETCH число

Здесь count – количество выбранных строк (может быть и нулевым). Заметьте, что в psql метка команды не выдается, так как вместо нее psql выводит выбранные строки.

34.5.Замечания

Если перемещение курсора в FETCH не ограничивается вариантами FETCH NEXT или FETCH FORWARD с положительным числом, курсор должен быть объявлен с указанием SCROLL. Для простых запросов Синергия-БД допускает обратное перемещение курсора, объявленного без SCROLL, но на это поведение лучше не рассчитывать. Если курсор объявлен с указанием NO SCROLL, перемещение назад запрещается.

Вариант ABSOLUTE несколько не быстрее, чем перемещение к требуемой строке с относительным сдвигом: нижележащий механизм все равно должен прочитать все промежуточные строки. Выборки по абсолютному отрицательному положению ещё хуже: сначала запрос необходимо прочитать до конца и найти последнюю строку, а затем вернуться назад к указанной строке. Однако перемотка к началу запроса (FETCH ABSOLUTE 0) выполняется быстро.

Определить курсор позволяет команда DECLARE, а переместить его, не читая данные, – команда MOVE.

34.6.Примеры

Следующий пример демонстрирует перемещение курсора в таблице:

```
BEGIN WORK;
```

```
-- Создание курсора:
```

```
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;
```

```
-- Получение первых 5 строк через курсор liahona:
```

```
FETCH FORWARD 5 FROM liahona;
```

code	title	did	date_prod	kind	len
BL101	The Third Man	101	1949-12-23	Drama	01:44
BL102	The African Queen	101	1951-08-11	Romantic	01:43
JL201	Une Femme est une Femme	102	1961-03-12	Romantic	01:25
P_301	Vertigo	103	1958-11-14	Action	02:08
P_302	Becket	103	1964-02-03	Drama	02:28

```
-- Получение предыдущей строки:
```

```
FETCH PRIOR FROM liahona;
```

code	title	did	date_prod	kind	len
P_301	Vertigo	103	1958-11-14	Action	02:08

```
-- Закрытие курсора и завершение транзакции:
```

```
CLOSE liahona;
```

```
COMMIT WORK;
```

34.7. Совместимость

В стандарте SQL команда `FETCH` определена только для встраиваемого SQL. Описанная здесь реализация `FETCH` возвращает данные подобно оператору `SELECT`, а не помещает их в переменные исполняющей среды. В остальном, `FETCH` полностью прямо-совместима со стандартом SQL.

Формы `FETCH` с `FORWARD` и `BACKWARD`, а также формы `FETCH` число и `FETCH ALL` (в которых `FORWARD` подразумевается) являются расширениями Синергия-БД.

В стандарте SQL перед именем курсора допускается только указание `FROM`; возможность указать `IN` или опустить оба указания относится к расширениям.

35. GRANT

`GRANT` – определить права доступа.

35.1. Синтаксис

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE |
REFERENCES | TRIGGER }
```

```
    [, ...] | ALL [ PRIVILEGES ] }
```

```
ON { [ TABLE ] имя_таблицы [, ...]
```

```
    | ALL TABLES IN SCHEMA имя_схемы [, ...] }
```

```
TO указание_роли [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { { SELECT | INSERT | UPDATE | REFERENCES }
```

```
    ( имя_столбца [, ...] )
```

```
    [, ...] | ALL [ PRIVILEGES ] ( имя_столбца
    [, ...] ) }
```

```
ON [ TABLE ] имя_таблицы [, ...]
```

```
TO указание_роли [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { { USAGE | SELECT | UPDATE }
```

```
    [, ...] | ALL [ PRIVILEGES ] }
```

```
ON { SEQUENCE имя_последовательности [, ...]
```

```
    | ALL SEQUENCES IN SCHEMA имя_схемы [, ...] }
```

TO указание_роли [, ...] [WITH GRANT OPTION]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...]
| ALL [PRIVILEGES] }

ON DATABASE имя_бд [, ...]

TO указание_роли [, ...] [WITH GRANT OPTION]

GRANT { USAGE | ALL [PRIVILEGES] }

ON DOMAIN имя_домена [, ...]

TO указание_роли [, ...] [WITH GRANT OPTION]

GRANT { USAGE | ALL [PRIVILEGES] }

ON FOREIGN DATA WRAPPER имя_обёртки_сторонних_данных
[, ...]

TO указание_роли [, ...] [WITH GRANT OPTION]

GRANT { USAGE | ALL [PRIVILEGES] }

ON FOREIGN SERVER имя_сервера [, ...]

TO указание_роли [, ...] [WITH GRANT OPTION]

GRANT { EXECUTE | ALL [PRIVILEGES] }

ON { { FUNCTION | PROCEDURE | ROUTINE }
имя_подпрограммы [([[режим_аргумента]
[имя_аргумента] тип_аргумента [, ...]])] [, ...]
| ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN
SCHEMA имя_схемы [, ...] }

TO указание_роли [, ...] [WITH GRANT OPTION]

GRANT { USAGE | ALL [PRIVILEGES] }

ON LANGUAGE имя_языка [, ...]

TO указание_роли [, ...] [WITH GRANT OPTION]

GRANT { { SELECT | UPDATE } [, ...] | ALL
[PRIVILEGES] }

ON LARGE OBJECT oid_BO [, ...]

```
TO указание_роли [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { { CREATE | USAGE } [, ...] | ALL  
[ PRIVILEGES ] }  
ON SCHEMA имя_схемы [, ...]  
TO указание_роли [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { CREATE | ALL [ PRIVILEGES ] }  
ON TABLESPACE табл_пространство [, ...]  
TO указание_роли [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { USAGE | ALL [ PRIVILEGES ] }  
ON TYPE имя_типа [, ...]  
TO указание_роли [, ...] [ WITH GRANT OPTION ]
```

Здесь указание_роли:

```
[ GROUP ] имя_роли  
| PUBLIC  
| CURRENT_USER  
| SESSION_USER
```

```
GRANT имя_роли [, ...] TO имя_роли [, ...] [ WITH ADMIN  
OPTION ]
```

35.2.Описание

Команда GRANT имеет две основные разновидности: первая назначает права для доступа к объектам баз данных (таблицам, колонкам, представлениям, сторонним таблицам, последовательностям, базам данных, оберткам сторонних данных, сторонним серверам, функциям, процедурным языкам, схемам или табличным пространствам), а вторая назначает одни роли членами других. Эти разновидности во многом похожи, но имеют достаточно отличий, чтобы рассматривать их отдельно.

35.3.GRANT для объектов баз данных

Эта разновидность команды GRANT дает одной или нескольким ролям

определенные права для доступа к объекту базы данных. Эти права добавляются к списку имеющихся, если роль уже наделена какими-то правами.

Также можно дать роли некоторое право для всех объектов одного типа в одной или нескольких схемах. Эта функциональность в настоящее время поддерживается только для таблиц, последовательностей и функций (но заметьте, что указание `ALL TABLES` распространяется также на представления и сторонние таблицы).

Ключевое слово `PUBLIC` означает, что права даются всем ролям, включая те, что могут быть созданы позже. `PUBLIC` можно воспринимать как неявно определенную группу, в которую входят все роли. Любая конкретная роль получит в сумме все права, данные непосредственно ей и ролям, членом которых она является, а также права, данные роли `PUBLIC`.

Если указано `WITH GRANT OPTION`, получатель права, в свою очередь, может давать его другим. Без этого указания распоряжаться своим правом он не сможет. Группе `PUBLIC` право передачи права дать нельзя.

Нет необходимости явно давать права для доступа к объекту его владельцу (обычно это пользователь, создавший объект), так как по умолчанию он имеет все права. Однако владелец может лишиться себя прав в целях безопасности.

Право удалять объект или изменять его определение произвольным образом не считается назначаемым; оно неотъемлемо связано с владельцем, так что отозвать это право или дать его кому-то другому нельзя. Однако похожий эффект можно получить, управляя членством в роли, владеющей объектом; см. ниже. Владелец также неявно получает право распоряжения всеми правами для своего объекта.

Синергия-БД по умолчанию назначает группе `PUBLIC` определенные права для некоторых типов объектов. Для таблиц, колонок, схем и табличных пространств `PUBLIC` по умолчанию никаких прав не имеет. Для других типов объектов `PUBLIC` имеет следующие права: `CONNECT` и `CREATE TEMP TABLE` – для баз данных; `EXECUTE` – для функций; `USAGE` – для языков. Владелец объекта, конечно же, может отозвать (с помощью `REVOKE`) как явно назначенные права, так и права, назначаемые по умолчанию. Для максимальной безопасности команду `REVOKE`

нужно выполнять в транзакции, создающей объект; тогда не образуется окно, в котором другой пользователь смог бы обратиться к объекту. Кроме того, эти изначально назначаемые права по умолчанию можно изменить, воспользовавшись командой ALTER DEFAULT PRIVILEGES.

SELECT

Позволяет выполнять SELECT для любой колонки или перечисленных колонок в заданной таблице, представлении или последовательности. Также позволяет выполнять COPY TO. Помимо того, это право требуется для обращения к существующим значениям колонок в UPDATE или DELETE. Для последовательностей это право позволяет пользоваться функцией currval. Для больших объектов это право позволяет читать содержимое объекта.

INSERT

Позволяет вставлять строки в заданную таблицу с помощью INSERT. Если право ограничивается несколькими колонками, только их значение можно будет задать в команде INSERT (другие колонки получают значения по умолчанию). Также позволяет выполнять COPY FROM.

UPDATE

Позволяет изменять (с помощью UPDATE) данные во всех, либо только перечисленных, колонках в заданной таблице. На практике для любой нетривиальной команды UPDATE потребуется и право SELECT, так как она должна обратиться к колонкам таблицы, чтобы определить, какие строки подлежат изменению, и/или вычислить новые значения колонок. Для SELECT ... FOR UPDATE и SELECT ... FOR SHARE также необходимо это право как минимум для одной колонки, помимо права SELECT. Для последовательностей это право позволяет пользоваться функциями nextval и setval. Для больших объектов это правило позволяет записывать данные в объект или обрезать его.

DELETE

Позволяет удалять строки из заданной таблицы с помощью DELETE. На практике для любой нетривиальной команды DELETE потребуется также право SELECT, так как она должна обратиться к колонкам таблицы, чтобы определить,

какие строки подлежат удалению.

TRUNCATE

Позволяет опустошить заданную таблицу с помощью TRUNCATE.

REFERENCES

Это право требуется для создания ограничений внешнего ключа, как для ссылающихся, так и для целевых колонок. Дать это право можно для всех, либо только для некоторых колонок таблицы.

TRIGGER

Позволяет создавать триггеры в заданной таблице.

CREATE

Для баз данных это право позволяет создавать схемы в заданной базе.

Для схем это право позволяет создавать новые объекты в заданной схеме. Чтобы переименовать существующий объект, необходимо быть владельцем этого объекта и иметь это право для схемы, содержащей его.

Для табличных пространств это право позволяет создавать таблицы, индексы и временные файлы в заданном табличном пространстве, а также создавать базы данных, для которых это пространство будет основным. Учтите, что когда это право отзывается, существующие объекты остаются в прежнем расположении.

CONNECT

Позволяет пользователю подключаться к указанной базе данных. Это право проверяется при установлении соединения (в дополнение к условиям, определенным в конфигурации `pg_hba.conf`).

TEMPORARY

TEMP

Позволяет создавать временные таблицы в заданной базе данных.

EXECUTE

Позволяет выполнять заданную функцию и применять любые определенные поверх нее операторы. Это единственный тип прав, применимый к функциям. Этот синтаксис распространяется и на агрегатные функции.

USAGE

Для процедурных языков это право позволяет создавать функции на заданном языке. Это единственный тип прав, применимый к процедурным языкам.

Для схем это право дает доступ к объектам, содержащимся в заданной схеме (предполагается, что при этом имеются права, необходимые для доступа к самим объектам). По сути это право позволяет субъекту «просматривать» объекты внутри схемы. Без этого разрешения имена объектов все же можно будет узнать, например, обратившись к системным таблицам. Кроме того, если отозвать это право, в существующих сеансах могут оказаться операторы, для которых просмотр имен объектов был выполнен ранее, так что это право не позволяет абсолютно надежно перекрыть доступ к объектам.

Для последовательностей это право позволяет использовать функции `currval` и `nextval`.

Для типов и доменов это право позволяет использовать заданный тип или домен при создании таблиц, функций или других объектов схемы. Заметьте, что это право не ограничивает общее «использование» типа, например, обращение к значениям типа в запросах. Не имея этого права, субъект лишается только возможности создавать объекты, зависящие от заданного типа. Основное предназначение этого права в том, чтобы ограничить круг пользователей, способных создавать зависимости от типа, которые могут впоследствии помешать владельцу типа изменить его.

Для оберток сторонних данных это право позволяет субъекту создавать определения сторонних серверов, использующих заданную обертку сторонних данных.

Для серверов это право позволяет субъекту создавать определения сторонних таблиц на заданном сервере, а также создавать, изменять и удалять сопоставление для собственного имени пользователя, связанное с этим сервером.

ALL PRIVILEGES

Дает целевой роли все права сразу. Ключевое слово `PRIVILEGES` является необязательным в Синергия-БД, хотя в строгом SQL оно требуется.

Права, требующиеся для других команд, указаны на страницах справки этих команд.

35.4.GRANT для ролей

Эта разновидность команды GRANT включает роль в члены одной или нескольких других ролей. Членство в ролях играет важную роль, так как права, данные роли, распространяются и на всех её членов.

С указанием WITH ADMIN OPTION новоиспеченный член роли сможет, в свою очередь, включать в члены этой роли, а также исключать из нее другие роли. Без этого указания обычные пользователи не могут это делать. Считается, что роль не имеет права WITH ADMIN OPTION для самой себя, но ей позволено управлять своими членами из сеанса, в котором пользователь сеанса соответствует данной роли. Суперпользователи баз данных могут включать или исключать любые роли из любых ролей. Роли с правом CREATEROLE могут управлять членством в любых ролях, кроме ролей суперпользователей.

В отличие от прав, членство в ролях нельзя назначить группе PUBLIC. Заметьте также, что эта форма команды не принимает избыточное слово GROUP.

35.5.Замечания

Для лишения субъектов прав доступа применяется команда REVOKE.

В Синергия-БД концепции пользователей и групп объединены в единую сущность, названную ролью. Таким образом, теперь нет необходимости добавлять ключевое слово GROUP, чтобы показать, что субъект является группой, а не пользователем. Слово GROUP все ещё принимается этой командой, но оно лишено смысловой нагрузки.

Пользователь может выполнять SELECT, INSERT и подобные команды с колонкой таблицы, если он имеет такое право для данной колонки или для всей таблицы. Если назначить пользователю требуемое право на уровне таблицы, а затем отозвать его для одной из колонок, это не даст эффекта, которого можно было бы ожидать: операция с правами на уровне колонок не затронет право на уровне таблицы.

Если назначить право доступа к объекту (с помощью GRANT) попытается не владелец объекта, команда завершится ошибкой, если пользователь не имеет

никаких прав для этого объекта. Если же пользователь имеет какие-то права, команда будет выполняться, но пользователь сможет давать другим только те права, которые даны ему с правом передачи. Формы `GRANT ALL PRIVILEGES` будут выдавать предупреждение, если у него вовсе нет таких прав, тогда как другие формы будут выдавать предупреждения, если пользователь не имеет прав распоряжаться именно правами, указанными в команде. В принципе, эти утверждения применимы и к владельцу объекта, но ему разрешено распоряжаться всеми правами, поэтому такие ситуации невозможны.

Следует отметить, что суперпользователи баз данных могут обращаться к любым объектам, вне зависимости от наличия каких-либо прав. Это сравнимо с привилегиями пользователя `root` в системе Unix. И так же, как `root`, роль суперпользователя следует использовать только когда это абсолютно необходимо.

Если суперпользователь решит выполнить команду `GRANT` или `REVOKE`, она будет выполнена, как если бы её выполнял владелец заданного объекта. В частности, права, назначенные такой командой, будут представлены как права, назначенные владельцем объекта. Если так же установить членство в роли, оно будет представлено как назначенное самой ролью.

`GRANT` и `REVOKE` также могут быть выполнены ролью, которая не является владельцем заданного объекта, но является членом роли-владельца, либо членом роли, имеющей права `WITH GRANT OPTION` для этого объекта. В этом случае, права будут записаны как назначенные ролью, которая действительно владеет объектом, либо имеет право `WITH GRANT OPTION`. Например, если таблица `t1` принадлежит роли `g1`, членом которой является `u1`, то `u1` может дать права на использование `t1` роли `u2`, но эти права будут представлены, как назначенные непосредственно ролью `g1`. Отозвать эти права позже сможет любой член роли `g1`.

Если роль, выполняющая команду `GRANT`, получает требуемое право по нескольким путям членства, какая именно роль будет выбрана в качестве назначающей право, не определено. Если это важно, в таких случаях рекомендуется воспользоваться командой `SET ROLE` и переключиться на роль, которую хочется видеть в качестве выполняющей `GRANT`.

При назначении прав для доступа к таблице они автоматически не распространяются на последовательности, используемые этой таблицей, в том числе, на последовательности, связанные с колонками SERIAL. Права доступа к последовательностям нужно назначать отдельно.

Чтобы получить информацию о существующих правах, назначенных для таблиц и колонок, воспользуйтесь командой \dp в psql. Например:

```
=> \dp mytable
```

```

                                     Access privileges
 Schema | Name      | Type  | Access privileges |
Column access privileges
-----+-----+-----+-----+-----
 public | mytable  | table | miriam=arwdDxt/miriam |
coll:
                                     : =r/miriam                                     :
miriam_rw=rw/miriam
                                     : admin=arw/miriam
(1 row)
```

Записи, выводимые командой \dp, интерпретируются так:

```
имя_роли=xxxx -- права, назначенные роли
               =xxxx -- права, назначенные PUBLIC
```

```

r -- SELECT («read», чтение)
w -- UPDATE («write», запись)
a -- INSERT («append», добавление)
d -- DELETE
D -- TRUNCATE
x -- REFERENCES
t -- TRIGGER
X -- EXECUTE
U -- USAGE
C -- CREATE
c -- CONNECT
T -- TEMPORARY
```

```
arwdDxt -- ALL PRIVILEGES (все права для таблиц;  
для других объектов другие)
```

```
* -- право передачи заданного права
```

```
/yyyy -- роль, назначившая это право
```

В примере выше показано, что увидит пользователь miriam, если создаст таблицу mytable и выполнит:

```
GRANT SELECT ON mytable TO PUBLIC;  
GRANT SELECT, UPDATE, INSERT ON mytable TO admin;  
GRANT SELECT (col1), UPDATE (col1) ON mytable TO  
miriam_rw;
```

Для других объектов есть другие команды \d, которые также позволяют просмотреть назначенные права.

Если колонка «Права доступа» для данного объекта пуста, это значит, что для объекта действуют стандартные права (то есть колонка прав содержит NULL). Права по умолчанию всегда включают все права для владельца и могут также включать некоторые права для PUBLIC в зависимости от типа объекта, как разъяснялось выше. Первая команда GRANT или REVOKE для объекта приводит к проявлению записи прав по умолчанию (например, {miriam=arwdDxt/miriam}), а затем изменяет эту запись в соответствии с заданным запросом. Подобным образом, строки, показанные в колонке «Права доступа к колонкам», выводятся только для колонок с нестандартными правами доступа. Заметьте, что в данном контексте под «стандартными правами» всегда подразумевается встроенный набор прав, предопределенный для типа объекта. Если с объектом связан набор прав по умолчанию, полученный после изменения в результате ALTER DEFAULT PRIVILEGES, измененные права будут всегда выводиться явно, показывая эффект команды ALTER.

Заметьте, что право распоряжения правами, которое имеет владелец, не отмечается в выводимой сводке. Знаком * отмечаются только те права с правом передачи, которые были явно назначены кому-либо.

35.6.Примеры

Следующая команда разрешает всем добавлять записи в таблицу `films`:
`GRANT INSERT ON films TO PUBLIC;`

Эта команда дает пользователю `manuel` все права для представления `kinds`:
`GRANT ALL PRIVILEGES ON kinds TO manuel;`

Учтите, что если её выполнит суперпользователь или владелец представления `kinds`, эта команда действительно даст субъекту все права, но если её выполнит обычный пользователь, субъект получит только те права, которые даны этому пользователю с правом передачи.

Включение в роль `admins` пользователя `joe`:
`GRANT admins TO joe;`

35.7.Совместимость

Согласно стандарту SQL, слово `PRIVILEGES` в указании `ALL PRIVILEGES` является обязательным. Стандарт SQL не позволяет назначать права сразу для нескольких объектов одной командой.

Синергия-БД позволяет владельцу объекта лишить себя своих обычных прав: например, владелец таблицы может разрешить себе только чтение таблицы, отозвав собственные права `INSERT`, `UPDATE`, `DELETE` и `TRUNCATE`. В стандарте SQL это невозможно. Это объясняется тем, что Синергия-БД воспринимает права владельца как назначенные им же себе; поэтому их можно и отозвать. В стандарте SQL права владельца даются ему предполагаемой сущностью «`_SYSTEM`». Так как владелец объекта отличается от «`_SYSTEM`», лишить себя этих прав он не может.

Согласно стандарту SQL, право с правом передачи можно дать субъекту `PUBLIC`; однако Синергия-БД может давать право с правом передачи только ролям.

В стандарте SQL право `USAGE` распространяется и на другие типы объектов: наборы символов, правила сортировки и преобразования.

В стандарте SQL право `USAGE` для последовательностей управляет использованием выражения `NEXT VALUE FOR`, которое равнозначно функции `nextval` в Синергия-БД. Права `SELECT` и `UPDATE` для последовательностей являются расширениями Синергия-БД. То, что право `USAGE` для

последовательностей управляет использованием функции `currval`, так же относится к расширениям Синергия-БД (как и сама функция).

Права для баз данных, табличных пространств, схем и языков относятся к расширениям Синергия-БД.

36. IMPORT FOREIGN SCHEMA

`IMPORT FOREIGN SCHEMA` -- импортировать определения таблиц со стороннего сервера.

36.1. Синтаксис

```
IMPORT FOREIGN SCHEMA удалённая_схема
    [ { LIMIT TO | EXCEPT } ( имя_таблицы [, ...] ) ]
FROM SERVER имя_сервера
INTO локальная_схема
    [ OPTIONS ( параметр 'значение' [, ...] ) ]
```

36.2. Описание

`IMPORT FOREIGN SCHEMA` создаёт сторонние таблицы, которые представляют таблицы, существующие на стороннем сервере. Новые сторонние таблицы будут принадлежать пользователю, выполняющему команду, и будут содержать корректные определения столбцов и параметры, соответствующие удалённым таблицам.

По умолчанию импортируются все таблицы и представления, существующие в определённой схеме на стороннем сервере. По желанию список таблиц можно ограничить некоторым подмножеством, или исключить из него конкретные таблицы. Новые сторонние таблицы создаются в целевой схеме, которая должна уже существовать.

Чтобы использовать `IMPORT FOREIGN SCHEMA`, необходимо иметь право `USAGE` для стороннего сервера, а также право `CREATE` в целевой схеме.

36.3. Параметры

удалённая_схема

Удалённая схема, из которой будут импортированы объекты. Что именно

представляет собой удалённая схема, зависит от применяемой обёртки сторонних данных.

```
LIMIT TO ( имя_таблицы [, ...] )
```

Импортировать только сторонние таблицы с заданными именами. Другие таблицы, существующие в сторонней схеме, будут проигнорированы.

```
EXCEPT ( имя_таблицы [, ...] )
```

Исключить из импорта указанные сторонние таблицы. Данная команда импортирует все таблицы, существующие в сторонней схеме, за исключением перечисленных в этом предложении.

```
имя_сервера
```

Сторонний сервер, с которого импортируется схема.

```
локальная_схема
```

Схема, в которой будут созданы импортируемые сторонние таблицы.

```
OPTIONS ( параметр 'значение' [, ...] )
```

Параметры, которые должны применяться при импорте. Допустимые имена параметров и их значения зависят от обёртки сторонних данных.

36.4.Примеры

Импорт определений таблиц из удалённой схемы `foreign_films` на сервере `film_server` с созданием сторонних таблиц в локальной схеме `films`:

```
IMPORT FOREIGN SCHEMA foreign_films  
FROM SERVER film_server INTO films;
```

Та же операция, но импортируются только таблицы `actors` и `directors` (если они существуют):

```
IMPORT FOREIGN SCHEMA foreign_films LIMIT TO (actors,  
directors)  
FROM SERVER film_server INTO films;
```

36.5.Совместимость

Команда `IMPORT FOREIGN SCHEMA` соответствует стандарту SQL, за исключением параметра `OPTIONS`, являющегося расширением Синергия-БД.

37. INSERT

INSERT – добавить строки в таблицу.

37.1. Синтаксис

```
[ WITH [ RECURSIVE ] запрос_WITH [ , ... ] ]  
INSERT INTO имя_таблицы [ AS псевдоним ] [ ( имя_столбца  
[ , ... ] ) ]  
    [ OVERRIDING { SYSTEM | USER } VALUE ]  
    { DEFAULT VALUES | VALUES ( { выражение | DEFAULT }  
[ , ... ] ) [ , ... ] | запрос }  
    [ ON CONFLICT [ объект_конфликта ]  
действие_при_конфликте ]  
    [ RETURNING * | выражение_результата [ [ AS ]  
имя_результата ] [ , ... ] ]
```

Здесь допускается объект_конфликта:

```
( { имя_столбца_индекса | ( выражение_индекса ) }  
[ COLLATE правило_сортировки ] [ класс_операторов ]  
[ , ... ] ) [ WHERE предикат_индекса ]  
ON CONSTRAINT имя_ограничения
```

И действие_при_конфликте может быть следующим:

```
DO NOTHING  
  
DO UPDATE SET { имя_столбца = { выражение |  
DEFAULT } |  
                ( имя_столбца [ , ... ] ) = [ ROW ]  
( { выражение | DEFAULT } [ , ... ] ) |  
                ( имя_столбца [ , ... ] ) =  
( вложенный_SELECT )  
                } [ , ... ]  
[ WHERE условие ]
```

37.2. Описание

INSERT добавляет строки в таблицу. Эта команда может добавить одну или несколько строк, сформированных выражениями значений, либо ноль или более строк, выданных дополнительным запросом.

Имена целевых колонок могут перечисляться в любом порядке. Если список с именами колонок отсутствует, по умолчанию целевыми колонками становятся все

колонки заданной таблицы; либо первые N из них, если только N колонок поступает от предложения `VALUES` или запроса. Значения, получаемые от предложения `VALUES` или запроса, связываются с явно или неявно определенным списком колонок слева направо.

Все колонки, не представленные в явном или неявном списке колонок, получают значения по умолчанию, если для них заданы эти значения, либо `NULL` в противном случае.

Если выражение для любой колонки выдает другой тип данных, система попытается автоматически привести его к нужному.

Предложение `ON CONFLICT` позволяет задать действие, заменяющее возникновение ошибки при нарушении ограничения уникальности или ограничения-исключения.

С необязательным предложением `RETURNING` команда `INSERT` вычислит и возвратит значения для каждой фактически добавленной строки (или изменённой, если применялось предложение `ON CONFLICT DO UPDATE`). В основном это полезно для получения значений, присвоенных по умолчанию, например, последовательного номера записи. Однако в этом предложении можно задать любое выражение со столбцами таблицы. Список `RETURNING` имеет тот же синтаксис, что и список результатов `SELECT`. В результате будут возвращены те строки, которые были успешно вставлены или изменены. Например, если строка была заблокирована, но не изменена, из-за того, что условие в предложении `ON CONFLICT DO UPDATE ... WHERE` не удовлетворено, эта строка возвращена не будет.

Чтобы добавлять строки в таблицу, необходимо иметь право `INSERT` для неё. Если присутствует предложение `ON CONFLICT DO UPDATE`, также требуется иметь право `UPDATE` для этой таблицы.

Если указывается список столбцов, достаточно иметь право `INSERT` только для перечисленных столбцов. Аналогично, с предложением `ON CONFLICT DO UPDATE` достаточно иметь право `UPDATE` только для столбцов, которые будут изменены. Однако предложение `ON CONFLICT DO UPDATE` также требует

наличия права `SELECT` для всех столбцов, значения которых считываются в выражениях `ON CONFLICT DO UPDATE` или в условии.

Для применения предложения `RETURNING` требуется право `SELECT` для всех столбцов, перечисленных в `RETURNING`. Если для добавления строк применяется запрос, для всех таблиц или столбцов, задействованных в этом запросе, разумеется, необходимо иметь право `SELECT`.

37.3.Параметры

37.3.1. Добавление

В этом разделе рассматриваются параметры, применяемые только при добавлении новых строк. Параметры, применяемые *исключительно* с предложением `ON CONFLICT`, описываются отдельно.

запрос_WITH

Предложение `WITH` позволяет задать один или несколько подзапросов, на которые затем можно сослаться по имени в запросе `INSERT`. Подробнее об этом см. `SELECT`.

Заданный запрос (оператор `SELECT`) также может содержать предложение `WITH`. В этом случае в запросе можно обращаться к обоим запросам_WITH, но второй будет иметь приоритет, так как он вложен ближе.

имя_таблицы

Имя (возможно, дополненное схемой) существующей таблицы.

псевдоним

Альтернативное имя, заменяющее имя_таблицы. Когда указывается это имя, оно полностью скрывает реальное имя таблицы. Это особенно полезно, когда в предложении `ON CONFLICT DO UPDATE` фигурирует таблица с именем `excluded`, так как это имя дано и специальной таблице, представляющей строки, предназначенные для добавления.

имя_столбца

Имя столбца в таблице имя_таблицы. Это имя столбца при необходимости может быть дополнено именем вложенного поля или индексом в массиве. Когда

данные вставляются только в некоторые поля столбца составного типа, в другие поля записывается NULL. Обращаясь к столбцу в предложении ON CONFLICT DO UPDATE, включать имя таблицы в ссылку на целевой столбец не нужно. Например, запись INSERT INTO table_name ... ON CONFLICT DO UPDATE SET table_name.col = 1 некорректна (это согласуется с общим поведением команды UPDATE).

OVERRIDING SYSTEM VALUE

Без этого предложения не допускается задание явного значения (отличного от DEFAULT) для столбца идентификации, определённого с характеристикой GENERATED ALWAYS. Данное предложение перекрывает это ограничение.

OVERRIDING USER VALUE

Если указывается это предложение, то значения, представляемые для столбцов идентификации, определённых с характеристикой GENERATED BY DEFAULT, игнорируются и вместо них применяются значения, выдаваемые последовательностью по умолчанию.

Это предложение полезно, например, при копировании значений между таблицами. Команда INSERT INTO tbl2 OVERRIDING USER VALUE SELECT * FROM tbl1 скопирует из tbl1 все столбцы, кроме столбцов идентификации в tbl2, а значения столбцов идентификации в tbl2 будут сгенерированы последовательностями в tbl2.

DEFAULT VALUES

Все колонки получают значения по умолчанию.

выражение

Выражение или значение, которое будет присвоено соответствующей колонке.

DEFAULT

Соответствующая колонка получит значение по умолчанию.

запрос

Запрос (оператор SELECT), который выдаст строки для добавления в таблицу.

Его синтаксис описан в справке оператора SELECT.

выражение_результата

Выражение, которое будет вычисляться и возвращаться командой INSERT

после добавления каждой строки. В этом выражении можно использовать имена любых колонок таблицы `имя_таблицы`. Чтобы получить все колонки, достаточно написать `*`.

`имя_результата`

Имя, назначаемое возвращаемой колонке.

37.3.2. Предложение ON CONFLICT

Необязательное предложение `ON CONFLICT` задаёт действие, заменяющее возникновение ошибки при нарушении ограничения уникальности или ограничения-исключения. Для каждой отдельной строки, предложенной для добавления, добавление либо выполняется успешно, либо, если нарушается решающее ограничение или индекс, задаваемые как `объект_конфликта`, выполняется альтернативное действие `конфликта`. Вариант `ON CONFLICT DO NOTHING` в качестве альтернативного действия просто отменяет добавление строки. Вариант `ON CONFLICT DO UPDATE` изменяет существующую строку, вызвавшую конфликт со строкой, предложенной для добавления.

Задаваемый `объект_конфликта` может выбирать уникальный индекс. Определение объекта, позволяющее выбрать индекс, включает один или несколько столбцов (их определяет `имя_столбца_индекса`) и/или выражение `индекса` и необязательный предикат `индекса`. Все уникальные индексы в таблице `имя_таблицы`, которые, без учёта порядка столбцов, содержат в точности столбцы/выражения, определяющие `объект_конфликта`, выбираются как решающие индексы. Если указывается предикат `индекса`, он должен, в качестве дополнительного требования выбора, удовлетворять индексам. Заметьте, что это означает, что не частичный уникальный индекс (уникальный индекс без предиката) будет выбран (и будет использоваться в `ON CONFLICT`), если такой индекс удовлетворяет всем остальным критериям. Если попытка выбрать индекс оказывается неудачной, выдаётся ошибка.

`ON CONFLICT DO UPDATE` гарантирует атомарный результат команды `INSERT` или `UPDATE`; при отсутствии внешних ошибок гарантируется один из двух

этих исходов, даже при большой параллельной активности. Эта операция также известна как UPSERT – «UPDATE или INSERT».

объект_конфликта

Определяет, для какого именно конфликта в ON CONFLICT будет предпринято альтернативное действие, устанавливая решающие индексы. Это указание позволяет осуществить выбор уникального индекса или явно задаёт имя ограничения. Для ON CONFLICT DO NOTHING объект_конфликта может не указываться; в этом случае игнорироваться будут все конфликты с любыми ограничениями (и уникальными индексами). Для ON CONFLICT DO UPDATE объект_конфликта должен указываться.

действие_при_конфликте

Параметр действие_при_конфликте задаёт альтернативное действие в случае конфликта. Это может быть либо DO NOTHING (не делать ничего), либо предложение DO UPDATE (произвести изменение), в котором указываются точные детали операции UPDATE, выполняемой в случае конфликта. Предложения SET и WHERE в ON CONFLICT DO UPDATE могут обращаться к существующей строке по имени таблицы (или псевдониму) или к строкам, предлагаемым для добавления, используя специальную таблицу excluded. Для чтения столбцов excluded необходимо иметь право SELECT для соответствующих столбцов в целевой таблице.

Заметьте, что эффект действий всех триггеров уровня строк BEFORE INSERT отражается в значениях excluded, так как в результате этих действий строка может быть исключена из множества добавляемых.

имя_столбца_индекса

Имя столбца в таблице имя_таблицы. Используется для выбора решающих индексов. Задаётся в формате CREATE INDEX. Чтобы запрос выполнялся, для столбца имя_столбца_индекса требуется право SELECT.

выражение_индекса

Подобно указанию имя_столбца_индекса, но применяется для выбора индекса по выражениям со столбцами таблицы имя_таблицы, фигурирующим в определениях индексов (не по простым столбцам). Задаётся в формате CREATE

INDEX. Для всех столбцов, к которым обращается выражение_индекса, необходимо иметь право SELECT.

правило_сортировки

Когда задаётся, устанавливает, что соответствующие имя_столбца_индекса или выражение_индекса должны использовать определённый порядок сортировки, чтобы этот индекс мог быть выбран. Обычно это указание опускается, так как от правил сортировки чаще всего не зависит, произойдёт ли нарушение ограничений или нет. Задаётся в формате CREATE INDEX.

класс_операторов

Когда задаётся, устанавливает, что соответствующие имя_столбца_индекса или выражение_индекса должны использовать определённый класс, чтобы индекс мог быть выбран. Обычно это указание опускается, потому что семантика равенства часто всё равно одна и та же в разных классах операторов типа, или потому что достаточно рассчитывать на то, что заданные уникальные индексы имеют адекватное определение равенства. Задаётся в формате CREATE INDEX.

предикат_индекса

Используется для выбора частичных уникальных индексов. Выбраны могут быть любые индексы, удовлетворяющие предикату (при этом они могут не быть собственно частичными индексами). Задаётся в формате CREATE INDEX. Для всех столбцов, задействованных в предикате_индекса, требуется право SELECT.

имя_ограничения

Явно задаёт решающее ограничение по имени, что заменяет неявный выбор ограничения или индекса.

условие

Выражение, выдающее значение типа boolean. Изменены будут только те строки, для которых это выражение выдаст true, хотя при выборе действия ON CONFLICT DO UPDATE заблокируются все строки. Заметьте, что условие вычисляется в конце, после того как конфликт был признан претендующим на

выполнение изменения.

Заметьте, что ограничения-исключения не могут быть решающими в `ON CONFLICT DO UPDATE`. Во всех случаях в качестве решающих поддерживаются только неоткладываемые (`NOT DEFERRABLE`) ограничения и уникальные индексы.

Команда `INSERT` с предложением `ON CONFLICT DO UPDATE` является «детерминированной». Это означает, что этой команде не разрешено воздействовать на любую существующую строку больше одного раза; в случае такой ситуации возникнет ошибка нарушения мощности множества. Строки, предлагаемые для добавления, не должны дублироваться с точки зрения атрибутов, ограничиваемых решающим индексом или ограничением.

Заметьте, что в настоящий момент не поддерживается ситуация, когда конструкция `ON CONFLICT DO UPDATE` команды `INSERT`, применяемой к секционированной таблице, изменяет ключ разбиения в конфликтующей строке так, что эта строка должна быть перенесена в новую секцию.

Примечание. Часто предпочтительнее использовать неявный выбор уникального индекса вместо непосредственного указания ограничения в виде `ON CONFLICT ON CONSTRAINT имя_ограничения`. Выбор продолжит корректно работать, когда нижележащий индекс будет заменён другим более или менее равнозначным индексом методом наложения, например, с использованием `CREATE UNIQUE INDEX ... CONCURRENTLY` и последующим удалением заменяемого индекса.

37.3.3. Выводимая информация

В случае успешного завершения, `INSERT` возвращает метку команды в виде:

```
INSERT oid число
```

Здесь число представляет количество добавленных строк. Если число равняется одному, а целевая таблица содержит `oid`, то вместо `oid` выводится `OID`, назначенный добавленной строке. В противном случае вместо `oid` выводится ноль.

Если команда `INSERT` содержит предложение `RETURNING`, её результат будет похож на результат оператора `SELECT` (с теми же колонками и значениями, что содержатся в списке `RETURNING`), полученный для строк, добавленных этой командой.

Примечание. Если целевая таблица является секционированной, каждая строка

перенаправляется в соответствующую секцию и вставляется в неё. Если целевая таблица является секцией и какая-либо из входных строк нарушает ограничение этой секции, происходит ошибка.

37.4.Примеры

Добавление одной строки в таблицу films:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy',
    '82 minutes');
```

В этом примере колонка len опускается и, таким образом, получает значение по умолчанию:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16',
    'Drama');
```

В этом примере для колонки с датой задается указание DEFAULT, а не явное значение:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82
minutes');
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

Добавление строки, полностью состоящей из значений по умолчанию:

```
INSERT INTO films DEFAULT VALUES;
```

Добавление нескольких строк с использованием многострочного синтаксиса VALUES:

```
INSERT INTO films (code, title, did, date_prod, kind)
VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT,
    'Comedy');
```

В этом примере в таблицу films вставляются некоторые строки из таблицы tmp_films, имеющей ту же структуру колонок, что и films:

```
INSERT INTO films SELECT * FROM tmp_films WHERE
date_prod < '2004-05-07';
```

Этот пример демонстрирует добавление данных в колонки с типом массива:

```
-- Создание пустого поля 3x3 для игры в крестики-нолики
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1, '{{« «,» «,» «},{« «,» «,» «},{« «,» «,»
«}}}') ;
-- Указания индексов в предыдущей команда могут быть
опущены
INSERT INTO tictactoe (game, board)
    VALUES (2, '{{X,» «,» «},{« «,O,» «},{« «,X,» «}}') ;
```

Добавление одной строки в таблицу distributors и получение последовательного номера, сгенерированного благодаря указанию DEFAULT:

```
INSERT INTO distributors (did, dname) VALUES (DEFAULT,
'XYZ Widgets')
    RETURNING did;
```

Увеличение счетчика продаж для продавца, занимающегося компанией Acme Corporation, и сохранение всей измененной строки вместе с текущим временем в таблице журнала:

```
WITH upd AS (
    UPDATE employees SET sales_count = sales_count + 1
    WHERE id =
        (SELECT sales_person FROM accounts WHERE name =
        'Acme Corporation')
    RETURNING *
)
INSERT INTO employees_log SELECT *, current_timestamp
FROM upd;
```

Добавить дистрибьюторов или изменить существующие данные должным образом. Предполагается, что в таблице определён уникальный индекс, ограничивающий значения в столбце did. Заметьте, что для обращения к значениям, изначально предлагаемым для добавления, используется специальная таблица excluded:

```
INSERT INTO distributors (did, dname)
    VALUES (5, 'Gizmo Transglobal'), (6, 'Associated
Computing, Inc')
```

```
ON CONFLICT (did) DO UPDATE SET dname =
EXCLUDED.dname;
```

Добавить дистрибьютора или не делать ничего для строк, предложенных для добавления, если уже есть существующая исключаяющая строка (строка, содержащая конфликтующие значения в столбце или столбцах после срабатывания триггеров перед добавлением строки). В данном примере предполагается, что определён уникальный индекс, ограничивающий значения в столбце did:

```
INSERT INTO distributors (did, dname) VALUES (7,
'Redline GmbH')
ON CONFLICT (did) DO NOTHING;
```

Добавить дистрибьюторов или изменить существующие данные должным образом. В данном примере предполагается, что в таблице определён уникальный индекс, ограничивающий значения в столбце did. Предложение WHERE позволяет ограничить набор фактически изменяемых строк (однако любая существующая строка, не подлежащая изменению, всё же будет заблокирована):

```
-- Не менять данные существующих дистрибьюторов в
-- зависимости от почтового индекса
INSERT INTO distributors AS d (did, dname) VALUES (8,
'Anvil Distribution')
ON CONFLICT (did) DO UPDATE
SET dname = EXCLUDED.dname || ' (formerly ' ||
d.dname || ')'
WHERE d.zipcode <> '21201';

-- Указать имя ограничения непосредственно в операторе
-- (связанный индекс
-- применяется для принятия решения о выполнении
-- действия DO NOTHING)
INSERT INTO distributors (did, dname) VALUES (9,
'Antwerp Design')
ON CONFLICT ON CONSTRAINT distributors_pkey DO
NOTHING;
```

Добавить дистрибьютора, если возможно; в противном случае не делать ничего (DO NOTHING). В данном примере предполагается, что в таблице определён уникальный индекс, ограничивающий значения в столбце did по подмножеству

строка, в котором булевский столбец `is_active` содержит `true`:

```
-- Этот оператор может выбрать частичный уникальный
индекс по «did»

-- с предикатом «WHERE is_active», а может и просто
использовать

-- обычное ограничение уникальности по столбцу «did»
INSERT INTO distributors (did, dname) VALUES (10,
'Conrad International')
    ON CONFLICT (did) WHERE is_active DO NOTHING;
```

37.5. Совместимость

`INSERT` соответствует стандарту SQL, но предложение `RETURNING` относится к расширениям Синергия-БД, как и возможность применять `WITH` с `INSERT` и возможность задавать альтернативное действие с `ON CONFLICT`. Кроме того, ситуация, когда список колонок опущен, но не все колонки получают значения из предложения `VALUES` или запроса, стандартом не допускается.

В стандарте SQL говорится, что предложение `OVERRIDING SYSTEM VALUE` может присутствовать, только если существует столбец идентификации, для которого всегда генерируется значение. Синергия-БД допускает это предложение в любом случае и игнорирует его в случае неприменимости.

Возможные ограничения предложения запрос описаны в справке `SELECT`.

38. LISTEN

`LISTEN` – ожидать уведомления.

38.1. Синтаксис

`LISTEN` канал

38.2. Описание

`LISTEN` регистрирует текущий сеанс для получения уведомления через канал с заданным именем (канал). Если текущий сеанс уже зарегистрирован и ожидает уведомления через этот канал, ничего не происходит.

Когда вызывается команда `NOTIFY` канал (в текущем или другом сеансе,

подключенном к той же базе данных), все сеансы, ожидающие уведомления через заданный канал, получают уведомление и каждый, в свою очередь, передает его подключенному клиентскому приложению.

Сеанс может отказаться от получения уведомлений через определенный канал с помощью команды `UNLISTEN`. Кроме того, подписка на любые уведомления автоматически отменяется при завершении сеанса.

Способ получения уведомлений клиентским приложением определяется программным интерфейсом Синергия-БД, который оно использует. Приложение, использующее библиотеку `libpq`, выполняет команду `LISTEN` как обычную команду SQL, а затем оно должно периодически вызывать функцию `PQnotifies`, чтобы проверить, не поступили ли новые уведомления. Другие интерфейсы, например `libpgtcl`, предоставляют более высокоуровневые методы для обработки событий уведомлений; на самом деле с `libpgtcl` разработчик приложения даже не должен непосредственно выполнять команды `LISTEN` и `UNLISTEN`. За дополнительными подробностями обратитесь к документации интерфейса, который вы используете.

В описании `NOTIFY` использование `LISTEN` и `NOTIFY` рассматривается более подробно.

38.3.Параметры

канал

Имя канала уведомлений (любой идентификатор).

38.4.Замечания

`LISTEN` начинает действовать при фиксировании транзакции. Если `LISTEN` или `UNLISTEN` выполняется в транзакции, которая затем откатывается, состояние подписки этого сеанса на уведомления не меняется.

Транзакция, в которой выполняется `LISTEN`, не может быть подготовлена для двухфазной фиксации.

38.5.Примеры

Демонстрация процедуры ожидания/получения уведомления в `psql`:

```
LISTEN virtual;
```

NOTIFY virtual;

Asynchronous notification «virtual» received from server process with PID 8448.

38.6.Совместимость

Оператор LISTEN отсутствует в стандарте SQL.

39. LOAD

LOAD – загрузить файл разделяемой библиотеки.

39.1.Синтаксис

LOAD 'имя_файла'

39.2.Описание

Эта команда загружает файл разделяемой библиотеки в адресное пространство сервера Синергия-БД. Если указанный файл был загружен ранее, эта команда не делает ничего. Файлы библиотек, содержащие функции на С, загружаются автоматически при первом вызове любой из этих функций. Поэтому явно выполнять LOAD обычно требуется только для загрузки библиотек, которые изменяют поведение сервера, внедряя свои обработчики, а не предоставляют некоторый набор функций.

Имя файла библиотеки обычно задаётся собственно именем файла, а полный путь определяется при просмотре пути поиска библиотек сервера (задаваемого в `dynamic_library_path`). Также в качестве имени может быть передан непосредственно полный путь. В любом случае расширение имени файла, стандартное для файлов разделяемых библиотек на данной платформе, можно опустить.

Обычные пользователи (не суперпользователи) могут использовать LOAD только для загрузки файлов библиотек, расположенных в `$libdir/plugins/` – заданное имя_файла должно начинаться именно с этой строки. Ответственность за то, чтобы в этом каталоге находились только «безопасные» библиотеки, лежит на администраторе баз данных.

39.3.Совместимость

LOAD является расширением Синергия-БД.

40. LOCK

LOCK – заблокировать таблицу.

40.1.Синтаксис

```
LOCK [ TABLE ] [ ONLY ] имя [ * ] [, ...] [ IN  
режим_блокировки MODE ] [ NOWAIT ]
```

Где возможные варианты режим_блокировки:

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE  
UPDATE EXCLUSIVE  
| SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS  
EXCLUSIVE
```

40.2.Описание

LOCK TABLE получает блокировку на уровне таблицы, при необходимости ожидая освобождения таблицы от других конфликтующих блокировок. Если указано NOWAIT, LOCK TABLE не ждет, пока таблица освободится: если блокировку нельзя получить немедленно, команда прерывается и выдается ошибка. Как только блокировка получена, она удерживается до завершения текущей транзакции. Команды UNLOCK TABLE не существует; блокировки всегда освобождаются в конце транзакции.

Когда блокируется представление, блокировка с тем же режимом рекурсивно распределяется на все отношения, фигурирующие в определяющем представлении запросе.

Запрашивая автоматические блокировки для команд, работающих с таблицами, Синергия-БД всегда выбирает наименее ограничивающий режим блокировки из возможных. Оператор LOCK TABLE предназначен для случаев, когда требуется более сильная блокировка. Например, предположим, что приложение выполняет транзакцию на уровне изоляции READ COMMITTED и оно должно получать неизменные данные на протяжении всей транзакции. Для достижения этой цели можно получить для таблицы блокировку в режиме SHARE, прежде чем обращаться

к ней. В результате параллельные изменения данных будут исключены и при последующих чтениях будет получено стабильное представление зафиксированных данных, так как режим блокировки SHARE конфликтует с блокировкой ROW EXCLUSIVE, запрашиваемой при записи, а LOCK TABLE имя IN SHARE MODE будет ждать, пока параллельные транзакции с блокировкой ROW EXCLUSIVE не будут зафиксированы или отменены. Таким образом, в момент получения такой блокировки не останется ни одной открытой незафиксированной операции записи; кроме того, никто не сможет записывать в таблицу, пока блокировка не будет снята.

Чтобы получить похожий эффект в транзакции на уровне изоляции REPEATABLE READ или SERIALIZABLE, необходимо выполнить оператор LOCK TABLE перед первым SELECT или оператором, изменяющим данные. Представление данных для транзакции уровня REPEATABLE READ или SERIALIZABLE будет заморожено в момент, когда начнет выполняться этот запрос. Если команда LOCK TABLE выполняется в транзакции позже, она так же исключает параллельную запись, но не дает гарантии, что транзакция будет читать последние зафиксированные данные.

Если в транзакции такого рода требуется изменять данные в таблице, для нее следует использовать режим блокировки SHARE ROW EXCLUSIVE вместо SHARE. Этот режим гарантирует, что в один момент времени будет выполняться только одна транзакция такого типа. Без этого ограничения возможна взаимоблокировка: две транзакции могут одновременно получить блокировки SHARE, после чего они не смогут получить блокировку ROW EXCLUSIVE, чтобы собственно выполнить изменения. Заметьте, что собственные блокировки транзакции никогда не конфликтуют, так что транзакция может получить блокировку ROW EXCLUSIVE, когда она владеет блокировкой SHARE – но не тогда, когда блокировку SHARE удерживает другая транзакция. Чтобы не допустить взаимоблокировок, убедитесь, что все транзакции запрашивают блокировки одних объектов в одинаковом порядке, и если для одного объекта запрашиваются блокировки в разных режимах, транзакции всегда должны запрашивать самую строгую блокировку.

40.3.Параметры

ИМЯ

Имя (возможно, дополненное схемой) существующей таблицы, для которой запрашивается блокировка. Если перед именем таблицы указано ONLY, блокируется только заданная таблица. Без ONLY блокируется и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно также добавить необязательное указание *, чтобы явно обозначить, что блокировка затрагивает и все дочерние таблицы.

Команда LOCK TABLE a, b; равнозначна последовательности LOCK TABLE a; LOCK TABLE b;. Таблицы блокируются по одной в порядке, заданном в команде LOCK TABLE.

режим_блокировки

Режим блокировки определяет, с какой блокировкой будет конфликтовать данная.

Если режим блокировки не указан, применяется самый строгий режим, ACCESS EXCLUSIVE.

NOWAIT

Указывает, что LOCK TABLE не должна ожидать освобождения конфликтующих блокировок: если запрошенная блокировка не может быть получена немедленно, транзакция прерывается.

40.4.Замечания

LOCK TABLE ... IN ACCESS SHARE MODE требует наличия права SELECT в целевой таблице. LOCK TABLE ... IN ROW EXCLUSIVE MODE требует наличия прав INSERT, UPDATE, DELETE или TRUNCATE для целевой таблицы. Все другие формы LOCK требуют наличия права UPDATE, DELETE или TRUNCATE на уровне таблицы.

Пользователь, выполняющий операцию блокировки представления, должен иметь соответствующее право для этого представления. Кроме того, владелец представления должен иметь сопутствующие права в нижележащих базовых

отношениях, хотя пользователь, устанавливающую блокировку, может этих прав не иметь.

Вне блока транзакции команда `LOCK TABLE` бесполезна: блокировка сохранится только до завершения операции. Поэтому Синергия-БД выдает ошибку при попытке применить `LOCK` не в блоке транзакции. Чтобы определить блок транзакции, используйте `BEGIN` и `COMMIT` (или `ROLLBACK`).

`LOCK TABLE` может устанавливать только блокировки на уровне таблицы, так что все имена режимов, включающие слово `ROW` (строка), не совсем корректны. Следует воспринимать их так, что в этих режимах пользователь намеревается получать в заблокированной таблице блокировки уровня строк. Также учтите, что в режиме `ROW EXCLUSIVE` устанавливается разделяемая блокировка таблицы. Заметьте, что применительно к `LOCK TABLE` все режимы блокировки действуют одинаково, отличаются только правила, определяющие, какой режим с каким конфликтует. Чтобы узнать, как получить блокировку именно на уровне строк, обратитесь к Предложение блокировки в справочной документации `SELECT`.

40.5.Примеры

Получение блокировки `SHARE` для первичного ключа таблицы при добавлении записи в подчиненную таблицу:

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom
Menace';
-- Если запись не будет возвращена, произойдет откат
транзакции
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

Установление блокировки `SHARE ROW EXCLUSIVE` в таблице первичного ключа перед выполнением операции удаления:

```
BEGIN WORK;
```

```
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;  
DELETE FROM films_user_comments WHERE id IN  
    (SELECT id FROM films WHERE rating < 5);  
DELETE FROM films WHERE rating < 5;  
COMMIT WORK;
```

40.6.Совместимость

Команда LOCK TABLE отсутствует в стандарте SQL, в нем уровни изоляции транзакции определяются командой SET TRANSACTION. Синергия-БД поддерживает и этот вариант; подробнее это описано в SET TRANSACTION.

За исключением ACCESS SHARE, ACCESS EXCLUSIVE и SHARE UPDATE EXCLUSIVE, режимы блокировки в Синергия-БД и синтаксис LOCK TABLE совместимы с теми, что представлены в СУБД Oracle.

41. MOVE

MOVE – переместить курсор.

41.1.Синтаксис

MOVE [direction [FROM | IN]] имя_курсора

Здесь direction может быть пустым или принимать следующее значение:

```
NEXT  
PRIOR  
FIRST  
LAST  
ABSOLUTE число  
RELATIVE число  
число  
ALL  
FORWARD  
FORWARD число  
FORWARD ALL  
BACKWARD  
BACKWARD число
```

BACKWARD ALL

41.2.Описание

MOVE перемещает курсор, не получая данные. Команда MOVE работает точно так же, как FETCH, но она не возвращает данные строк, а только перемещает курсор.

Команда MOVE поддерживает те же параметры, что и FETCH; за подробным описанием её синтаксиса и использования обратитесь к FETCH.

41.3.Выводимая информация

В случае успешного завершения, MOVE возвращает метку команды в виде:

MOVE число

Здесь число показывает количество строк, которое бы выдала команда FETCH с такими же параметрами (оно может быть нулевым).

41.4.Примеры

```
BEGIN WORK;
```

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

```
-- Пропустить первые 5 строк:
```

```
MOVE FORWARD 5 IN liahona;
```

```
MOVE 5
```

```
-- Выбрать 6-ую строку из курсора liahona:
```

```
FETCH 1 FROM liahona;
```

```
code | title | did | date_prod | kind | len
-----+-----+-----+-----+-----+-----
P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)
```

```
-- Закрыть курсор liahona и завершить транзакцию:
```

```
CLOSE liahona;
```

```
COMMIT WORK;
```

41.5.Совместимость

Оператор MOVE отсутствует в стандарте SQL.

42. NOTIFY

NOTIFY – сгенерировать уведомление.

42.1.Синтаксис

NOTIFY канал [, сообщение]

42.2.Описание

Команда NOTIFY отправляет событие уведомления вместе с дополнительной строкой «сообщения» всем клиентским приложениям, которые до этого выполнили в текущей базе данных LISTEN канал с указанным именем канала. Уведомления видны всем пользователям.

NOTIFY предоставляет простой механизм межпроцессного взаимодействия для множества процессов, работающих с одной базой данных Синергия-БД. Вместе с уведомлением может быть передана строка сообщения, а передавая дополнительные данные через таблицы базы данных, можно создать более высокоуровневые механизмы обмена структурированными данными.

Информация, передаваемая клиенту с уведомлением, включает имя канала уведомлений, PID серверного процесса, управляющего сеансом, который выдал уведомление, и строку сообщения (она будет пустой, если сообщение не задано).

Выбор подходящих имен каналов и их назначения – дело проектировщика базы данных. Обычно имя канала совпадает с именем какой-либо таблицы в базе, а событие уведомления по сути означает «я изменила эту таблицу, посмотри, что она содержит теперь». Однако команды NOTIFY и LISTEN не навязывают именно такой подход. Например, проектировщик базы данных может выбрать разные имена каналов, чтобы сигнализировать о разных типах изменений в одной таблице. Кроме того, строку сообщения тоже можно использовать для выделения различных событий.

Если требуется сигнализировать о факте изменений в определенной таблице,

используя NOTIFY, можно применить полезный программный прием – поместить NOTIFY в правило, которое будет срабатывать при изменениях в таблице. При таком подходе уведомление будет выдаваться автоматически, так что прикладной программист не рискует случайно оставить какое-либо изменение без уведомления.

Транзакции оказывают значительное влияние на работу NOTIFY. Во-первых, если NOTIFY выполняется внутри транзакции, уведомления доставляются получателям после фиксирования транзакции и только в этом случае. Это разумно, так как в случае прерывания транзакции действие всех команд в ней аннулируется, включая NOTIFY. Однако это может обескуражить тех, кто ожидает, что уведомления будут приходить немедленно. Во-вторых, если ожидающий сеанс получает уведомление внутри транзакции, это событие не будет доставлено подключенному клиенту до завершения (фиксации или отката) транзакции. Это опять же объясняется тем, что если уведомление будет доставлено в рамках транзакции, которая затем будет прервана, может возникнуть желание как-то отменить его – но сервер не может «забрать назад» уведомление после того, как оно было отправлено клиенту. Поэтому уведомления доставляются только между транзакциями. Учитывая вышесказанное, в приложениях, применяющих NOTIFY для сигнализации в реальном времени, следует минимизировать размер транзакций.

Если в рамках одной транзакции в один канал поступило несколько уведомлений с одинаковым сообщением, сервер может решить доставить только одно уведомление. Если же сообщения различаются, уведомления будут всегда доставлены по отдельности. Так же уведомления, поступающие от разных транзакций, никогда не будут объединены в одно. Не считая фильтрации последующих экземпляров дублирующихся уведомлений, NOTIFY гарантирует, что уведомления от одной транзакции всегда поступают в том же порядке, в каком были отправлены. Также гарантируется, что сообщения от разных транзакций поступают в порядке фиксации этих транзакций.

Часто бывает, что клиент, выполнивший NOTIFY, ожидает уведомления на этом же канале. В этом случае он получит свое же уведомление, как и любой другой сеанс, ожидающий уведомления. В зависимости от логики приложения, это может

привести к бессмысленным операциям, например, поиску изменений в таблице, которые и были внесены этим же сеансом. Этой дополнительной работы можно избежать, если проверить, не совпадает ли PID сигнализирующего процесса (указанный в данных события) с собственным PID сеанса (его можно узнать, обратившись к `libpq`). Если они совпадают, значит сеанс получил уведомление о собственных действиях, так что его можно игнорировать.

42.3.Параметры

канал

Имя канала для передачи уведомления (любой идентификатор).

сообщение

Строка «сообщения», которая будет передана вместе с уведомлением. Она должна задаваться простой текстовой константой. В стандартной конфигурации её длина должна быть меньше 8000 байт. Если требуется передать двоичные данные или большой объем информации, лучше поместить их в таблицу базы данных и передать ключ этой записи.

42.4.Замечания

Уведомления, которые были отправлены, но ещё не обработаны всеми ожидающими сеансами, содержатся в очереди. Если эта очередь переполняется, транзакции, в которых вызывается `NOTIFY`, будут завершены ошибкой при попытке фиксации. Очередь довольно велика (8 ГБ в стандартной конфигурации), так что её размера должно хватать практически во всех случаях, но если в сеансе выполняется `LISTEN`, а затем продолжается очень длительная транзакция, очередь не очищается. Как только эта очередь заполняется наполовину, в журнал записываются предупреждения, в которых указывается, какой сеанс препятствует очистке очереди. В этом случае следует добиться завершения текущей транзакции в указанном сеансе, чтобы очередь была очищена.

Функция `pg_notification_queue_usage` показывает, какой процент очереди в данный момент занят ожидающими уведомлениями.

Транзакция, в которой выполняется `NOTIFY`, не может быть подготовлена для

двухфазной фиксации.

42.5.pg_notify

Также отправить уведомление можно, используя функцию `pg_notify(text, text)`. Эта функция принимает в первом аргументе имя канала, а во втором текст сообщения. Гораздо удобнее использовать ее, когда требуется работать с динамическими именами каналов и сообщениями.

42.6.Примеры

Демонстрация процедуры ожидания/получения уведомления в psql:

```
LISTEN virtual;
```

```
NOTIFY virtual;
```

```
Asynchronous notification «virtual» received from server  
process with PID 8448.
```

```
NOTIFY virtual, 'This is the payload';
```

```
Asynchronous notification «virtual» with payload «This  
is the payload» received from server process with PID  
8448.
```

```
LISTEN foo;
```

```
SELECT pg_notify('fo' || 'o', 'pay' || 'load');
```

```
Asynchronous notification «foo» with payload «payload»  
received from server process with PID 14728.
```

42.7.Совместимость

Оператор `NOTIFY` отсутствует в стандарте SQL.

43. PREPARE

`PREPARE` – подготовить оператор к выполнению.

43.1.Синтаксис

```
PREPARE имя [ ( тип_данных [, ...] ) ] AS оператор
```

43.2.Описание

`PREPARE` создает подготовленный оператор. Подготовленный оператор

представляет собой объект на стороне сервера, позволяющий оптимизировать производительность приложений. Когда выполняется `PREPARE`, указанный оператор разбирается, анализируется и переписывается. При последующем выполнении команды `EXECUTE` подготовленный оператор планируется и исполняется. Такое разделение труда исключает повторный разбор запроса, при этом позволяет выбрать наилучший план выполнения в зависимости от определенных значений параметров.

Подготовленные операторы могут принимать параметры – значения, которые подставляются в оператор, когда он собственно выполняется. При создании подготовленного оператора к этим параметрам можно обращаться по порядковому номеру, используя запись `$1`, `$2` и т.д. Дополнительно можно указать список соответствующих типов данных параметров. Если тип данных параметра не указан или объявлен как `unknown` (неизвестный), тип выводится из контекста, в котором используется этот параметр (если это возможно). При выполнении оператора фактические значения параметров передаются команде `EXECUTE`. За подробностями обратитесь к `EXECUTE`.

Подготовленные операторы существуют только в рамках текущего сеанса работы с БД. Когда сеанс завершается, система забывает подготовленный оператор, так что его надо будет создать снова, чтобы использовать дальше. Это также означает, что один подготовленный оператор не может использоваться одновременно несколькими клиентами базы данных; но каждый клиент может создать собственный подготовленный оператор и использовать его. Освободить подготовленный оператор можно вручную, выполнив команду `DEALLOCATE`.

Подготовленные операторы дают наибольший выигрыш в производительности, когда в одном сеансе выполняется большое число однотипных операторов. Отличие в производительности особенно значительно, если операторы достаточно сложны для планирования или перезаписи, например, когда в запросе объединяется множество таблиц или необходимо применить несколько правил. Если оператор относительно прост в этом плане, но сложен для выполнения, выигрыш от использования подготовленных операторов будет менее заметным.

43.3.Параметры

ИМЯ

Произвольное имя, назначаемое данному подготовленному оператору. Оно должно быть уникальным в рамках одного сеанса; это имя затем используется для выполнения или освобождения ранее подготовленного оператора.

тип_данных

Тип данных параметра подготовленного оператора. Если тип данных конкретного параметра не задан или задан как unknown, он будет выводиться из контекста, в котором используется параметр. Для обращения к параметрам в самом подготовленном операторе используется запись \$1, \$2 и т.д.

оператор

Любой оператор SELECT, INSERT, UPDATE, DELETE или VALUES.

43.4.Замечания

Подготовленные операторы могут использовать общие планы, а не перестраивать план для каждого набора переданных значений EXECUTE. Для подготовленных операторов без параметров это происходит сразу; иначе общий план выбирается после пяти и более выполнений, при которых получаются планы с ожидаемой средней стоимостью (включая издержки планирования), превышающей оценку стоимости общего плана. Когда общий план выбран, он будет использоваться до конца жизни подготовленного оператора. При использовании значений EXECUTE, которые редко встречаются в столбцах со множеством дублирующихся значений, могут быть построены специализированные планы настолько выгоднее общего плана, что даже с издержками планирования общий план может не использоваться никогда.

Для общего плана предполагается, что значения, передаваемые в EXECUTE, являются уникальными значениями в столбце и что эти значения распределены равномерно. Например, если в статистике записаны три различных значения столбца, с общим планом предполагается, что проверке на равенство для столбца будут соответствовать 33% обработанных строк. Статистика по столбцам также позволяет общим планам точно вычислять избирательность для уникальных

столбцов. Сравнения по столбцам с неоднородным распределением и указания несуществующих значений влияют на среднюю стоимость плана и следовательно, на то, будет ли выбран общий план и когда.

Чтобы узнать, какой план выполнения выбирает Синергия-БД для подготовленного оператора, воспользуйтесь EXPLAIN (например, напишите EXPLAIN EXECUTE). Если применяется общий план, он будет содержать символы параметров \$n, тогда как в специализированном плане будут подставлены фактические значения параметров. Оценки строк в общем плане отражают избирательность, вычисленную для конкретных параметров. Более подробно о планировании запросов и статистике, которую собирает Синергия-БД для этих целей, можно узнать в документации ANALYZE.

Хотя основной смысл подготовленных операторов в том, чтобы избежать многократного разбора и планирования оператора, Синергия-БД будет принудительно заново анализировать и планировать выполнение оператора всякий раз, когда объекты базы данных, задействованные в операторе, подвергаются изменениям определения (DDL) со времени предыдущего использования подготовленного оператора. Кроме того, если от одного использования оператора к другому меняется значение search_path, оператор будет так же разобран заново с новым search_path. С этими правилами использование подготовленного оператора по сути почти не отличается от выполнения одного и того же запроса снова и снова, но дает выигрыш по скорости (если определения объектов не меняются), особенно если оптимальный план от раза к разу не меняется. Однако различия все же могут проявиться – например, когда оператор обращается к таблице по неполному имени, а затем в схеме, стоящей в пути search_path раньше, создается другая таблица с таким же именем, автоматический пересмотр запроса не происходит, так как никакой объект в определении оператора не изменился. Однако, если автоматический пересмотр произойдет в результате других изменений, при последующем выполнении запроса будет задействована новая таблица.

Получить список всех доступных в сеансе подготовленных операторов можно, обратившись к системному представлению pg_prepared_statements.

43.5.Примеры

Создание подготовленного оператора для команды INSERT, который затем выполняется:

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Создание подготовленного оператора для команды SELECT, который затем выполняется:

```
PREPARE usrrptplan (int) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND
    u.usrid=l.usrid
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

Заметьте, что тип данных второго параметра не указывается, так что он выводится из контекста, в котором используется \$2.

43.6.Совместимость

В стандарте SQL есть оператор PREPARE, но он предназначен только для применения во встраиваемом SQL. Эта версия оператора PREPARE имеет также несколько другой синтаксис.

44. PREPARE TRANSACTION

PREPARE TRANSACTION – подготовить текущую транзакцию для двухфазной фиксации.

44.1.Синтаксис

```
PREPARE TRANSACTION id_транзакции
```

44.2.Описание

PREPARE TRANSACTION подготавливает текущую транзакцию для двухфазной фиксации. После этой команды транзакция перестает быть связанной с текущим сеансом; её состояние полностью сохраняется на диске, и есть очень большая вероятность, что она будет успешно зафиксирована, даже если до этого

времени работа базы данных будет прервана аварийно.

Подготовленную транзакцию затем можно зафиксировать или отменить командами `COMMIT PREPARED` и `ROLLBACK PREPARED`, соответственно. Эти команды можно вызывать из любого сеанса, не только из того, в котором эта транзакция создавалась.

С точки зрения сеанса, выполняющего команду, `PREPARE TRANSACTION` не отличается от `ROLLBACK`: после её выполнения не активна никакая транзакция, а результат действия подготовленной транзакции становится невидимым (он окажется видимым снова, если транзакция будет зафиксирована).

Если при выполнении команды `PREPARE TRANSACTION` по какой-то причине происходит сбой, команда действует как `ROLLBACK`: текущая транзакция откатывается.

44.3.Параметры

`id_транзакции`

Произвольный идентификатор, по которому затем на эту транзакцию будут ссылаться команды `COMMIT PREPARED` или `ROLLBACK PREPARED`. Идентификатор должен задаваться строковой константой не длиннее 200 байтов и должен отличаться от идентификаторов любых других подготовленных на данный момент транзакций.

44.4.Замечания

`PREPARE TRANSACTION` не предназначена для использования в приложениях или интерактивных сеансах. Ее задача – дать возможность внешнему менеджеру транзакций выполнять атомарные глобальные транзакции, охватывающие несколько баз данных или другие транзакционные ресурсы. Обычно применять `PREPARE TRANSACTION` следует только при разработке собственного менеджера транзакций.

Эта команда должна выполняться внутри блока транзакции. Начинает блок транзакции команда `BEGIN`.

В настоящее время команда `PREPARE` неспособна подготавливать транзакции,

в которых выполнялись какие-либо действия с временными таблицами, создавались курсоры `WITH HOLD` или выполнялись команды `LISTEN` и `UNLISTEN`. Эти функции слишком тесно связаны с текущим сеансом, так что в подготовленной транзакции они не были бы полезны.

Если транзакция меняет какие-либо параметры времени выполнения командой `SET` (без указания `LOCAL`), их значения сохраняются после `PREPARE TRANSACTION` и не зависят от последующих команд `COMMIT PREPARED` и `ROLLBACK PREPARED`. Так что в этом отношении `PREPARE TRANSACTION` больше похожа на `COMMIT`, чем на `ROLLBACK`.

Все существующие в текущий момент подготовленные транзакции показываются в системном представлении `pg_prepared_xacts`.

Примечание. Оставлять транзакции в подготовленном состоянии на долгое время не рекомендуется. Это повлияет на способность команды `VACUUM` высвобождать пространство, а в крайнем случае может привести к отключению базы данных для предотвращения заикливания ID транзакций. Также учтите, что транзакция продолжит удерживать все свои блокировки. Это сделано с расчётом на то, что подготовленная транзакция будет зафиксирована или отменена как только внешний менеджер транзакций убедится, что все другие базы данных так же готовы к фиксации. В отсутствие настроенного внешнего менеджера транзакций, который бы отслеживал подготовленные транзакции и своевременно закрывал их, лучше вообще отключить поддержку подготовленных транзакций, установив `max_prepared_transactions` равным нулю. Это не позволит случайно создать подготовленные транзакции, которые могут быть забыты и в конце концов станут причиной проблем.

44.5.Примеры

Текущая транзакция подготавливается для двухфазной фиксации, при этом ей назначается идентификатор `foobar`:

```
PREPARE TRANSACTION 'foobar';
```

44.6.Совместимость

Оператор `PREPARE TRANSACTION` является расширением Синергия-БД. Он предназначен для использования внешними системами управления транзакциями, некоторые из которых работают по стандартам (например, X/Open XA), но сторона SQL в этих системах не стандартизирована.

45. REASSIGN OWNED

REASSIGN OWNED – сменить владельца объектов базы данных, принадлежащих заданной роли.

45.1. Синтаксис

```
REASSIGN OWNED BY { старая_роль | CURRENT_USER |  
SESSION_USER } [, ...]  
TO { новая_роль | CURRENT_USER |  
SESSION_USER }
```

45.2. Описание

REASSIGN OWNED указывает системе сменить владельца объектов баз данных, принадлежащих одной из старых_ролей, на новую_роль.

45.3. Параметры

старая_роль

Имя роли. Все объекты в текущей базе данных и все общие объекты (базы данных, табличные пространства), принадлежащие этой роли, станут принадлежать новой_роли.

новая_роль

Имя роли, которая станет новым владельцем затронутых объектов.

45.4. Замечания

REASSIGN OWNED часто применяется при подготовке к удалению одной или нескольких ролей. Так как команда REASSIGN OWNED затрагивает объекты только в текущей базе данных, обычно её нужно выполнять в каждой базе данных, которая содержит объекты, принадлежащие удаляемой роли.

Для выполнения REASSIGN OWNED необходимо быть членом и исходной, и целевой роли.

Команда DROP OWNED предлагает альтернативное решение, просто удаляя все объекты в базе данных, принадлежащие одной или нескольким заданным ролям.

Команда REASSIGN OWNED не затрагивает никакие права, которые даны старым_ролям для объектов, им не принадлежащим. Также она не затрагивает права

по умолчанию, установленные командой ALTER DEFAULT PRIVILEGES. Отозвать эти права можно, воспользовавшись командой DROP OWNED.

45.5.Совместимость

Оператор REASSIGN OWNED является расширением Синергия-БД.

46. REFRESH MATERIALIZED VIEW

REFRESH MATERIALIZED VIEW – заменить содержимое материализованного представления.

46.1.Синтаксис

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] имя  
[ WITH [ NO ] DATA ]
```

46.2.Описание

REFRESH MATERIALIZED VIEW полностью заменяет содержимое материализованного представления. Эту команду разрешено выполнять только владельцам мат. представления. Старое его содержимое при этом аннулируется. Если добавлено указание WITH DATA (или нет никакого), нижележащий запрос выполняется и выдает новые данные, так что материализованное представление остается в сканируемом состоянии. Если указано WITH NO DATA, новые данные не выдаются и представление оказывается в несканируемом состоянии.

Указать CONCURRENTLY вместе с WITH NO DATA нельзя.

46.3.Параметры

CONCURRENTLY

Обновить материализованное представление, не блокируя параллельные выборки из него. Без данного параметра обновление, затрагивающее много строк, обычно задействует меньше ресурсов и выполнится быстрее, но может препятствовать чтению этого материализованного представления другими сеансами. При этом данный режим может быть быстрее при небольшом количестве строк.

Данный параметр допускается, только если в материализованном

представлении есть хотя бы один индекс UNIQUE, построенный только по именам колонок и включающий все строки; то есть, в нем не должно быть выражений и предложения WHERE.

Этот параметр нельзя использовать, когда материализованное представление ещё не наполнено.

Даже с этим параметром в один момент времени допускается только одно обновление (REFRESH) материализованного представления.

Имя

Имя (возможно, дополненное схемой) материализованного представления, подлежащего обновлению.

46.4.Замечания

Тогда как индекс по умолчанию для операций CLUSTER команда REFRESH MATERIALIZED VIEW сохраняет, она не упорядочивает генерируемые строки по нему. Если генерируемые данные необходимо упорядочить, в определяющем запросе нужно явно указать ORDER BY.

46.5.Примеры

Эта команда заменяет содержимое материализованного представления order_summary, используя запрос из определения материализованного представления, и оставляет его в сканируемом состоянии:

```
REFRESH MATERIALIZED VIEW order_summary;
```

Эта команда освобождает пространство, связанное с материализованным представлением annual_statistics_basis, и оставляет это представление в несканируемом состоянии:

```
REFRESH MATERIALIZED VIEW annual_statistics_basis WITH  
NO DATA;
```

46.6.Совместимость

REFRESH MATERIALIZED VIEW – расширение Синергия-БД.

47. REINDEX

REINDEX – перестроить индексы.

47.1.Синтаксис

```
REINDEX [ ( VERBOSE ) ] { INDEX | TABLE | SCHEMA |  
DATABASE | SYSTEM } имя
```

47.2.Описание

REINDEX перестраивает индекс, обрабатывая данные таблицы, к которой относится индекс, и в результате заменяет старую копию индекса. Команда REINDEX применяется в следующих ситуациях:

1. Индекс был поврежден, его содержимое стало некорректным. Хотя в теории этого не должно случаться, на практике индексы могут испортиться из-за программных ошибок или аппаратных сбоев. В таких случаях REINDEX служит методом восстановления индекса.
2. Индекс стал «раздутым», то есть в нем оказалось много пустых или почти пустых страниц. Это может происходить с индексами В-дерева в Синергия-БД при определенных, достаточно редких сценариях использования. REINDEX дает возможность сократить объем, занимаемый индексом, записывая новую версию индекса без «мертвых» страниц.
3. Параметр хранения индекса (например, фактор заполнения) был изменен, и теперь требуется, чтобы это изменение вступило в силу в полной мере.
4. Построение индекса с параметром CONCURRENTLY завершилось ошибкой, в результате чего индекс оказался «нерабочим». Такие индексы бесполезны, но команда REINDEX предоставляет удобный способ перестроить их. Однако заметьте, что REINDEX перестраивает индекс не в параллельном режиме. Чтобы перестроить такой индекс, минимизируя влияние на производственную среду, его следует удалить, а затем снова выполнить команду CREATE INDEX CONCURRENTLY.

47.3.Параметры

INDEX

Перестраивает указанный индекс.

TABLE

Перестраивает все индексы в указанной таблице. Если у таблицы имеется дополнительная таблица «TOAST», она так же переиндексируется.

SCHEMA

Перестраивает все индексы в указанной схеме. Если таблица в этой схеме имеет вторичную таблицу «TOAST», она также будет переиндексирована. При этом обрабатываются и индексы в общих системных каталогах. Эту форму REINDEX нельзя выполнить в блоке транзакции.

DATABASE

Перестраивает все индексы в текущей базе данных. При этом обрабатываются также индексы в общих системных каталогах. Эту форму REINDEX нельзя выполнить в блоке транзакции.

SYSTEM

Перестраивает все индексы в системных каталогах текущей базы данных. При этом обрабатываются также индексы в общих системных каталогах, но индексы в таблицах пользователя не затрагиваются. Эту форму REINDEX нельзя выполнить в блоке транзакции.

ИМЯ

Имя определенного индекса, таблицы или базы данных, подлежащих переиндексации. В настоящее время REINDEX DATABASE и REINDEX SYSTEM могут переиндексировать только текущую базу данных, так что их параметр должен соответствовать имени текущей базы данных.

VERBOSE

Выводит отчёт о прогрессе после переиндексации каждого индекса.

47.4.Замечания

В случае подозрений в повреждении индекса таблицы пользователя, этот индекс или все индексы таблицы можно перестроить, используя команду REINDEX INDEX или REINDEX TABLE.

Все усложняется, если возникает необходимость восстановить поврежденный

индекс системной таблицы. В этом случае важно, чтобы система сама не использовала этот индекс. На самом деле в таких случаях вы, скорее всего, столкнетесь с падением процессов сервера в момент запуска, как раз вследствие испорченных индексов. Чтобы надежно восстановить рабочее состояние, сервер следует запускать с параметром `-P`, который отключает использование индексов при поиске в системных каталогах.

Один из вариантов сделать это – выключить Синергия-БД и запустить его снова в однопользовательском режиме, с параметром `-P` в командной строке. Затем можно выполнить `REINDEX DATABASE`, `REINDEX SYSTEM`, `REINDEX TABLE` или `REINDEX INDEX`, в зависимости от того, что вы хотите восстановить. В случае сомнений выполните `REINDEX SYSTEM`, чтобы перестроить все системные индексы в базе данных. Затем завершите однопользовательский сеанс сервера и перезапустите сервер в обычном режиме. Чтобы подробнее узнать, как работать с сервером в однопользовательском интерфейсе, обратитесь к справочной странице `postgres`.

Можно так же запустить обычный экземпляр сервера, но добавить в параметры командной строки `-P`. В разных клиентах это может делаться по-разному, но во всех клиентах на базе `libpq` можно установить для переменной окружения `PGOPTIONS` значение `-P` до запуска клиента. Учтите, что хотя этот метод не препятствует работе других клиентов, все же имеет смысл не позволять им подключаться к поврежденной базе данных до завершения восстановления.

Действие `REINDEX` подобно удалению и пересозданию индекса в том смысле, что содержимое индекса пересоздается с нуля, но блокировки при этом устанавливаются другие. `REINDEX` блокирует запись, но не чтение родительской таблицы индекса. Эта команда также устанавливает исключительную блокировку для обрабатываемого индекса, что блокирует чтение таблицы, при котором задействуется этот индекс. `DROP INDEX`, напротив, моментально устанавливает исключительную блокировку на родительскую таблицу, блокируя и запись, и чтение. Последующая команда `CREATE INDEX` блокирует запись, но не чтение; так как индекс отсутствует, обращений к нему ни при каком чтении не будет, что означает,

что блокироваться чтение не будет, но выполняться оно будет как дорогостоящее последовательное сканирование.

Для перестраивания одного индекса или индексов таблицы необходимо быть владельцем этого индекса или таблицы. Для переиндексирования схемы или базы данных необходимо быть владельцем этой схемы или базы. Заметьте, что вследствие этого в некоторых случаях не только суперпользователи могут перестраивать индексы таблиц, принадлежащих другим пользователям. Однако из этих правил есть исключение – когда команду REINDEX DATABASE, REINDEX SCHEMA или REINDEX SYSTEM выполняет не суперпользователь, индексы общих каталогов будут пропускаться, если только данный каталог не принадлежит этому пользователю (как правило, это так). Разумеется, суперпользователи могут переиндексировать всё без ограничений.

Переиндексирование секционированных таблиц или секционированных индексов не поддерживается. Переиндексировать можно каждую секцию по отдельности.

47.5.Примеры

Перестроение одного индекса:

```
REINDEX INDEX my_index;
```

Перестроение всех индексов таблицы my_table:

```
REINDEX TABLE my_table;
```

Перестроение всех индексов в определенной базе данных, в предположении, что целостность системных индексов под сомнением:

```
$ export PGOPTIONS=«-P»
```

```
$ psql broken_db
```

```
...
```

```
broken_db=> REINDEX DATABASE broken_db;
```

```
broken_db=> \q
```

47.6.Совместимость

Команда REINDEX отсутствует в стандарте SQL.

48. RELEASE SAVEPOINT

RELEASE SAVEPOINT – высвободить ранее определенную точку сохранения.

48.1.Синтаксис

RELEASE [SAVEPOINT] имя_точки_сохранения

48.2.Описание

RELEASE SAVEPOINT уничтожает точку сохранения, определенную ранее в текущей транзакции.

После уничтожения точка сохранения становится неприменимой в качестве точки возврата, но никаких других проявлений, видимых для пользователя, эта команда не имеет. Она не отменяет эффекта команд, выполненных после установки точки сохранения. Для этого предназначена команда ROLLBACK TO SAVEPOINT. Уничтожение точки сохранения, когда она становится не нужна, позволяет системе освобождать некоторые ресурсы раньше, чем завершается транзакция.

RELEASE SAVEPOINT также уничтожает все точки сохранения, установленные после заданной точки.

48.3.Параметры

имя_точки_сохранения

Имя точки сохранения, подлежащей уничтожению.

48.4.Замечания

Указание имени точки сохранения, не определенной ранее, считается ошибкой.

Освободить точку сохранения в транзакции, находящейся в прерванном состоянии, нельзя.

Если одно имя дано нескольким точкам сохранения, освобождена будет только последняя из них.

48.5.Примеры

Этот пример показывает, как установить и затем уничтожить точку сохранения:

BEGIN;

```
INSERT INTO table1 VALUES (3);  
SAVEPOINT my_savepoint;  
INSERT INTO table1 VALUES (4);  
RELEASE SAVEPOINT my_savepoint;  
COMMIT;
```

Данная транзакция вставит значения 3 и 4.

48.6.Совместимость

Эта команда соответствует стандарту SQL. В стандарте говорится, что ключевое слово `SAVEPOINT` является обязательным, но Синергия-БД позволяет опускать его.

49. RESET

`RESET` – восстановить значение по умолчанию заданного параметра времени выполнения.

49.1.Синтаксис

```
RESET параметр_конфигурации  
RESET ALL
```

49.2.Описание

`RESET` сбрасывает параметры времени выполнения к значениям по умолчанию. `RESET` – альтернативное написание команды:

```
SET параметр_конфигурации TO DEFAULT
```

За подробностями обратитесь к `SET`.

Значение по умолчанию определяется как значение, которое имел бы этот параметр, если бы для него не выполнялась команда `SET` в текущем сеансе. Фактическое значение может быть задано в компилируемом коде, файле конфигурации, параметрах командной строки или в параметрах по умолчанию для базы данных или пользователя. Если определить его как «значение, которое имеет параметр в начале сеанса», это будет не вполне корректно, так как оно будет сброшено к тому, что задано в файле конфигурации в данный момент.

Транзакционное поведение команды `RESET` не отличается от `SET`: её действие

будет отменено при откате транзакции.

49.3.Параметры

параметр_конфигурации

Имя устанавливаемого параметра конфигурации времени выполнения.

Доступные параметры описаны справочной странице SET.

ALL

Сбрасывает к значениям по умолчанию все устанавливаемые параметры времени выполнения.

49.4.Примеры

Сброс конфигурационной переменной `timezone` к значению по умолчанию:

```
RESET timezone;
```

49.5.Совместимость

RESET является расширением Синергия-БД.

50. REVOKE

REVOKE – отозвать права доступа.

50.1.Синтаксис

```
REVOKE [ GRANT OPTION FOR ]
      { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE |
REFERENCES | TRIGGER }
      [, ...] | ALL [ PRIVILEGES ] }
ON { [ TABLE ] имя_таблицы [, ...]
    | ALL TABLES IN SCHEMA имя_схемы [, ...] }
FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
      { { SELECT | INSERT | UPDATE | REFERENCES }
      ( имя_столбца [, ...] )
      [, ...] | ALL [ PRIVILEGES ] ( имя_столбца
      [, ...] ) }
```



```
ON [ TABLE ] имя_таблицы [, ...]
FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
{ { USAGE | SELECT | UPDATE }
  [, ...] | ALL [ PRIVILEGES ] }
ON { SEQUENCE имя_последовательности [, ...]
     | ALL SEQUENCES IN SCHEMA имя_схемы [, ...] }
FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
{ { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] |
  ALL [ PRIVILEGES ] }
ON DATABASE имя_бд [, ...]
FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
{ USAGE | ALL [ PRIVILEGES ] }
ON DOMAIN имя_домена [, ...]
FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
{ USAGE | ALL [ PRIVILEGES ] }
ON FOREIGN DATA WRAPPER имя_обёртки_сторонних_данных
[, ...]
FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
{ USAGE | ALL [ PRIVILEGES ] }
```

```
ON FOREIGN SERVER имя_сервера [, ...]
FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
      { EXECUTE | ALL [ PRIVILEGES ] }
      ON { { FUNCTION | PROCEDURE | ROUTINE } имя_функции
[ ( [ [ режим_аргумента ] [ имя_аргумента ]
тип_аргумента [, ...] ] ) ] [, ...]
      | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN
SCHEMA имя_схемы [, ...] }
      FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
      [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
      { USAGE | ALL [ PRIVILEGES ] }
      ON LANGUAGE имя_языка [, ...]
      FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
      [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
      { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
      ON LARGE OBJECT oid_БО [, ...]
      FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
      [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
      { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
      ON SCHEMA имя_схемы [, ...]
      FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
      [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
      { CREATE | ALL [ PRIVILEGES ] }
      ON TABLESPACE табл_пространство [, ...]
```

```
FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]  
    { USAGE | ALL [ PRIVILEGES ] }  
ON TYPE имя_типа [, ...]  
FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ]
```

```
REVOKE [ ADMIN OPTION FOR ]  
    имя_роли [, ...] FROM имя_роли [, ...]  
[ CASCADE | RESTRICT ]
```

50.2.Описание

Команда REVOKE лишает одну или несколько ролей прав, назначенных ранее. Ключевое слово PUBLIC обозначает неявно определенную группу всех ролей.

Различные типы прав подробно рассматриваются в описании команды GRANT.

Заметьте, что любая конкретная роль получает в сумме права, данные непосредственно ей, права, данные любой роли, в которую она включена, а также права, данные группе PUBLIC. Поэтому, например, лишение PUBLIC права SELECT не обязательно будет означать, что все роли лишатся права SELECT для данного объекта: оно сохранится у тех ролей, которым оно дано непосредственно или косвенно, через другую роль. Подобным образом, лишение права SELECT какого-либо пользователя может не повлиять на его возможность пользоваться правом SELECT, если это право дано группе PUBLIC или другой роли, в которую он включен.

Если указано GRANT OPTION FOR, отзывается только право передачи права, но не само право. Без этого указания отзывается и право, и право распоряжаться им.

Если пользователь обладает правом с правом передачи и он дал его другим пользователям, последнее право считается зависимым. Когда первый пользователь лишается самого права или права передачи и существуют зависимые права, эти зависимые права также отзываются, если дополнительно указано CASCADE; в

противном случае операция завершается ошибкой. Это рекурсивное лишение прав затрагивает только права, полученные через цепочку пользователей, которую можно проследить до пользователя, являющегося субъектом команды REVOKE. Таким образом, пользователи могут в итоге сохранить это право, если оно было также получено через других пользователей.

Когда отзывается право доступа к таблице, с ним вместе автоматически отзываются соответствующие права для каждой колонки таблицы (если такие права заданы). С другой стороны, если роли были даны права для таблицы, лишение роли таких же прав на уровне отдельных колонок ни на что не влияет.

При лишении пользователя членства в роли GRANT OPTION меняется на ADMIN OPTION, в остальном поведение команды похожее. Заметьте также, что эта форма команды не принимает избыточное слово GROUP.

50.3.Замечания

Для просмотра прав, назначенных для существующих таблиц и колонок, можно воспользоваться командой \dp в psql. Формат её вывода рассматривается в описании GRANT. Для других, не табличных объектов предусмотрены другие команды \d, которые могут показывать в том числе и назначенные для них права.

Пользователь может отзывать только те права, которые он дал другому непосредственно. Если, например, пользователь А дал право с правом передачи пользователю В, а пользователь В, в свою очередь, дал это право пользователю С, то пользователь А не сможет лишить этого права непосредственно С. Вместо этого, пользователь А может лишить права передачи права пользователя В и использовать параметр CASCADE, чтобы этого права по цепочке лишился пользователь С. Если же, в другом примере, и А, и В дали одно и то же право С, то А сможет отозвать право, которое дал он, но не пользователь В, так что в результате С все равно будет иметь это право.

Если отозвать право доступа к объекту (с помощью REVOKE) попытается не владелец объекта, команда завершится ошибкой, если пользователь не имеет никаких прав для этого объекта. Если же пользователь имеет какие-то права,

команда будет выполняться, но пользователь сможет отозвать только те права, которые даны ему с правом распоряжения ими. Формы REVOKE ALL PRIVILEGES будут выдавать предупреждение, если у него вовсе нет таких прав, тогда как другие формы будут выдавать предупреждения, если пользователь не имеет права распоряжаться именно правами, указанными в команде. В принципе, эти утверждения применимы и к владельцу объекта, но ему разрешено распоряжаться всем правами, поэтому такие ситуации невозможны.

Если команду GRANT или REVOKE выполняет суперпользователь, эта команда выполняется так, как будто её выполняет владелец затрагиваемого объекта. Так как все права в конце концов исходят от владельца объекта (возможно, косвенно по цепочке или через право распоряжением правом), суперпользователь может отозвать все права, но это может потребовать применения режима CASCADE, как описывалось выше.

REVOKE также может быть выполнена ролью, которая не является владельцем заданного объекта, но является членом роли-владельца, либо членом роли, имеющей права WITH GRANT OPTION для этого объекта. В этом случае команда будет выполнена, как если бы её выполняла содержащая роль, действительно владеющая объектом или имеющая права WITH GRANT OPTION. Например, если таблица `t1` принадлежит роли `g1`, членом которой является роль `u1`, то `u1` может отзывать права на использование `t1`, которые записаны как данные ролью `g1`. В том числе это могут быть права, данные ролью `u1`, а также другими членами роли `g1`.

Если роль, выполняющая команду REVOKE, получила указанные права косвенно по нескольким путям членства ролей, какая именно роль будет выбрана для выполнения команды, не определено. В таких случаях рекомендуется воспользоваться командой SET ROLE и переключиться на роль, которую хочется видеть в качестве выполняющей REVOKE. Если этого не сделать, могут быть отозваны не те права, что планировалось, либо отозвать права вообще не удастся.

50.4.Примеры

Лишение группы `public` права добавлять данные в таблицу `films`:

```
REVOKE INSERT ON films FROM PUBLIC;
```

Лишение пользователя manuel всех прав для представления kinds:

```
REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

Заметьте, что на самом деле это означает «лишить всех прав, которые дал я».

Исключение из членов роли admins пользователя joe:

```
REVOKE admins FROM joe;
```

50.5.Совместимость

Замечания по совместимости, приведенные для команды GRANT, справедливы и для REVOKE. Стандарт требует обязательного указания ключевого слова RESTRICT или CASCADE, но Синергия-БД подразумевает RESTRICT по умолчанию.

51. ROLLBACK

ROLLBACK – прервать текущую транзакцию.

51.1.Синтаксис

```
ROLLBACK [ WORK | TRANSACTION ]
```

51.2.Описание

ROLLBACK откатывает текущую транзакцию и приводит к аннулированию всех изменений, произведённых транзакцией.

51.3.Параметры

WORK

TRANSACTION

Необязательные ключевые слова, не оказывают никакого влияния.

51.4.Замечания

Чтобы завершить и зафиксировать транзакцию, используйте COMMIT.

При выполнении команды ROLLBACK вне блока транзакции выдаётся предупреждение и больше ничего не происходит.

51.5.Примеры

Чтобы прервать все операции:

```
ROLLBACK;
```

51.6.Совместимость

В стандарте SQL описаны только две формы: ROLLBACK и ROLLBACK WORK. В остальном эта команда полностью соответствует стандарту.

52. ROLLBACK PREPARED

ROLLBACK PREPARED – отменить транзакцию, которая ранее была подготовлена для двухфазной фиксации.

52.1.Синтаксис

```
ROLLBACK PREPARED id_транзакции
```

52.2.Описание

ROLLBACK PREPARED откатывает транзакцию в подготовленном состоянии.

52.3.Параметры

id_транзакции

Идентификатор транзакции, которую нужно откатить.

52.4.Замечания

Откатить подготовленную транзакцию может либо пользователь, выполнявший её изначально, либо суперпользователь. При этом не обязательно работать в том же сеансе, где выполнялась транзакция.

Эту команду нельзя выполнить внутри блока транзакции. Подготовленная транзакция откатывается немедленно.

Все существующие в текущий момент подготовленные транзакции показываются в системном представлении pg_prepared_xacts.

52.5.Примеры

Откат транзакции, имеющей идентификатор foobar:

```
ROLLBACK PREPARED 'foobar';
```

52.6.Совместимость

Оператор ROLLBACK PREPARED является расширением Синергия-БД. Он

предназначен для использования внешними системами управления транзакциями, некоторые из которых работают по стандартам (например, X/Open XA), но сторона SQL в этих системах не стандартизирована.

53. ROLLBACK TO SAVEPOINT

ROLLBACK TO SAVEPOINT – откатиться к точке сохранения.

53.1.Синтаксис

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ]  
имя_точки_сохранения
```

53.2.Описание

Откатывает все команды, выполненные после установления точки сохранения. Точка сохранения остается действующей и при необходимости можно снова откатиться к ней позже.

ROLLBACK TO SAVEPOINT неявно уничтожает все точки сохранения, установленные после заданной точки.

53.3.Параметры

имя_точки_сохранения

Точка сохранения, к которой нужно откатиться.

53.4.Замечания

Чтобы уничтожить точку сохранения, не отменяя действия команд, выполненных после нее, применяется команда RELEASE SAVEPOINT.

Указание имени точки сохранения, не установленной ранее, считается ошибкой.

Курсоры проявляют не совсем транзакционное поведение применительно к точкам сохранения. Любой курсор, открытый внутри точки сохранения, будет закрыт при откате к этой точке. Если ранее открытый курсор был перемещен командой FETCH или MOVE внутри точки сохранения, к которой затем произошел откат, курсор остается в той позиции, в которой он остался после FETCH (то есть, перемещение курсора, производимое командой FETCH, не откатывается). Также при

откате не отменяется и закрытие курсора. Однако другие побочные эффекты, вызываемые запросом курсора (например, побочные действия изменчивых функций, вызываемых в запросе) отменяются, если они производятся после точки сохранения, к которой затем происходит откат. Курсор, выполнение которого приводит к прерыванию транзакции, переводится в нерабочее состояние, так что даже если восстановить транзакцию, выполнив `ROLLBACK TO SAVEPOINT`, этот курсор нельзя будет использовать.

53.5.Примеры

Отмена действия команд, выполненных после установки точки сохранения

`my_savepoint:`

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

Откат к точке сохранения не отражается на положении курсора:

```
BEGIN;
```

```
DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;
```

```
SAVEPOINT foo;
```

```
FETCH 1 FROM foo;
```

```
  ?column?
```

```
-----
```

```
      1
```

```
ROLLBACK TO SAVEPOINT foo;
```

```
FETCH 1 FROM foo;
```

```
  ?column?
```

```
-----
```

```
      2
```

```
COMMIT;
```

53.6.Совместимость

В стандарте SQL говорится, что ключевое слово SAVEPOINT является обязательным, но Синергия-БД и Oracle позволяют опускать его. SQL допускает WORK, но не TRANSACTION, в качестве избыточного слова после ROLLBACK. Кроме того, в SQL есть дополнительное предложение AND [NO] CHAIN, которое в настоящее время не поддерживается в Синергия-БД. В остальном эта команда соответствует стандарту SQL.

54. SAVEPOINT

SAVEPOINT – определить новую точку сохранения в текущей транзакции.

54.1.Синтаксис

SAVEPOINT имя_точки_сохранения

54.2.Описание

SAVEPOINT устанавливает новую точку сохранения в текущей транзакции.

Точка сохранения – это специальная отметка внутри транзакции, которая позволяет откатить все команды, выполненные после нее, и восстановить таким образом состояние на момент установки этой точки.

54.3.Параметры

имя_точки_сохранения

Имя, назначаемое новой точке сохранения.

54.4.Замечания

Для отката к установленной точке сохранения предназначена команда ROLLBACK TO SAVEPOINT. Чтобы уничтожить точку сохранения, сохраняя изменения, произведенные после того, как она была установлена, применяется команда RELEASE SAVEPOINT.

Точки сохранения могут быть установлены только внутри блока транзакции. В одной транзакции можно определить несколько точек сохранения.

54.5.Примеры

Установление точки сохранения и затем отмена действия всех команд, выполненных после установленной точки:

```
BEGIN;  
    INSERT INTO table1 VALUES (1);  
    SAVEPOINT my_savepoint;  
    INSERT INTO table1 VALUES (2);  
    ROLLBACK TO SAVEPOINT my_savepoint;  
    INSERT INTO table1 VALUES (3);  
COMMIT;
```

Показанная транзакция вставит в таблицу значения 1 и 3, но не 2.

Этот пример показывает, как установить и затем уничтожить точку сохранения:

```
BEGIN;  
    INSERT INTO table1 VALUES (3);  
    SAVEPOINT my_savepoint;  
    INSERT INTO table1 VALUES (4);  
    RELEASE SAVEPOINT my_savepoint;  
COMMIT;
```

Данная транзакция вставит значения 3 и 4.

54.6.Совместимость

Стандарт SQL требует, чтобы точка сохранения уничтожалась автоматически, когда устанавливается другая точка сохранения с тем же именем. В Синергия-БД старая точка сохранения остается, хотя при откате или уничтожении будет выбираться только самая последняя. После уничтожения последней точки командой `RELEASE SAVEPOINT` доступной для команд `ROLLBACK TO SAVEPOINT` и `RELEASE SAVEPOINT` становится следующая. В остальном оператор `SAVEPOINT` полностью соответствует стандарту.

55. SELECT

`SELECT`, `TABLE`, `WITH` – получить строки из таблицы или представления.

55.1.Синтаксис

```
[ WITH [ RECURSIVE ] запрос_WITH [, ...] ]  
SELECT [ ALL | DISTINCT [ ON ( выражение [, ...] ) ] ]  
      [ * | выражение [ [ AS ] имя_результата ] [, ...] ]  
      [ FROM элемент_FROM [, ...] ]  
      [ WHERE условие ]  
      [ GROUP BY элемент_группирования [, ...] ]  
      [ HAVING условие [, ...] ]  
      [ WINDOW имя_окна AS ( определение_окна ) [, ...] ]  
      [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ]  
выборка ]  
      [ ORDER BY выражение [ ASC | DESC | USING оператор ]  
[ NULLS { FIRST | LAST } ] [, ...] ]  
      [ LIMIT { число | ALL } ]  
      [ OFFSET начало [ ROW | ROWS ] ]  
      [ FETCH { FIRST | NEXT } [ число ] { ROW | ROWS }  
ONLY ]  
      [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE }  
[ OF table_name [, ...] ] [ NOWAIT ] [...] ]
```

Здесь допускается элемент_FROM:

```
      [ ONLY ] table_name [ * ] [ [ AS ] псевдоним  
[ ( псевдоним_колонки [, ...] ) ] ]  
      [ TABLESAMPLE метод_выборки ( аргумент  
[, ...] ) [ REPEATABLE ( затравка ) ] ]  
      [ LATERAL ] ( выборка ) [ AS ] псевдоним  
[ ( псевдоним_колонки [, ...] ) ]  
      имя_запроса_WITH [ [ AS ] псевдоним  
[ ( псевдоним_колонки [, ...] ) ] ]  
      [ LATERAL ] имя_функции ( [ argument [, ...] ] )  
      [ WITH ORDINALITY ] [ [ AS ] псевдоним  
[ ( псевдоним_колонки [, ...] ) ] ]  
      [ LATERAL ] имя_функции ( [ argument [, ...] ] )  
[ AS ] псевдоним ( определение_колонки [, ...] )  
      [ LATERAL ] имя_функции ( [ argument [, ...] ] ) AS  
( определение_колонки [, ...] )  
      [ LATERAL ] ROWS FROM( имя_функции ( [ argument  
[, ...] ] ) [ AS ( определение_колонки [, ...] ) ]
```

[, ...])

[WITH ORDINALITY] [[AS] псевдоним
[(псевдоним_колонки [, ...])]]

элемент_FROM [NATURAL] тип_соединения элемент_FROM
[ON условие_соединения | USING (колонка_соединения
[, ...])]

И элемент_группирования может быть следующим:

()
выражение
(выражение [, ...])
ROLLUP ({ выражение | (выражение [, ...]) }
[, ...])
CUBE ({ выражение | (выражение [, ...]) }
[, ...])
GROUPING SETS (элемент_группирования [, ...])

И запрос_WITH:

имя_запроса_WITH [(имя_колонки [, ...])] AS
(выборка | values | insert | update | delete)

TABLE [ONLY] table_name [*]

55.2.Описание

SELECT получает строки из множества таблиц (возможно, пустого). Общая процедура выполнения SELECT следующая:

1. Выполняются все запросы в списке WITH. По сути они формируют временные таблицы, к которым затем можно обращаться в списке FROM. Запрос в WITH выполняется только один раз, даже если он фигурирует в списке FROM неоднократно.
2. Вычисляются все элементы в списке FROM. Каждый элемент в списке FROM представляет собой реальную или виртуальную таблицу. Если список FROM содержит несколько элементов, они объединяются перекрестным соединением.
3. Если указано предложение WHERE, все строки, не удовлетворяющие условию, исключаются из результата.

4. Если присутствует указание `GROUP BY`, либо в запросе вызываются агрегатные функции, вывод разделяется по группам строк, соответствующим одному или нескольким значениям, а затем вычисляются результаты агрегатных функций. Если добавлено предложение `HAVING`, оно исключает группы, не удовлетворяющие заданному условию.
5. Вычисляются фактические выходные строки по заданным в `SELECT` выражениям для каждой выбранной строки или группы строк.
6. `SELECT DISTINCT` исключает из результата повторяющиеся строки. `SELECT DISTINCT ON` исключает строки, совпадающие по всем указанным выражениям. `SELECT ALL` (по умолчанию) возвращает все строки результата, включая дубликаты.
7. Операторы `UNION`, `INTERSECT` и `EXCEPT` объединяют вывод нескольких операторов `SELECT` в один результирующий набор. Оператор `UNION` возвращает все строки, представленные в одном, либо обоих наборах результатов. Оператор `INTERSECT` возвращает все строки, представленные строго в обоих наборах. Оператор `EXCEPT` возвращает все строки, представленные в первом наборе, но не во втором. Во всех трех случаях повторяющиеся строки исключаются из результата, если явно не указано `ALL`. Чтобы явно обозначить, что выдаваться должны только неповторяющиеся строки, можно добавить избыточное слово `DISTINCT`. Заметьте, что в данном контексте по умолчанию подразумевается `DISTINCT`, хотя в самом `SELECT` по умолчанию подразумевается `ALL`.
8. Если присутствует предложение `ORDER BY`, возвращаемые строки сортируются в указанном порядке. В отсутствие `ORDER BY` строки возвращаются в том порядке, в каком системе будет проще их выдать.
9. Если указано предложение `LIMIT` (или `FETCH FIRST`), либо `OFFSET`, оператор `SELECT` возвращает только подмножество строк результата.
10. Если указано `FOR UPDATE`, `FOR NO KEY UPDATE`, `FOR SHARE` или `FOR KEY SHARE`, оператор `SELECT` блокирует выбранные строки, защищая их от одновременных изменений.

Для всех колонок, задействованных в команде SELECT, необходимо иметь право SELECT. Применение блокировок FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE или FOR KEY SHARE требует также права UPDATE (как минимум для одной колонки в каждой выбранной для блокировки таблице).

55.3.Параметры

55.3.1. Предложение WITH

Предложение WITH позволяет задать один или несколько подзапросов, к которым затем можно обратиться по имени в основном запросе. Эти подзапросы по сути действуют как временные таблицы или представления в процессе выполнения главного запроса. Каждый подзапрос может представлять собой оператор SELECT, TABLE, VALUES, INSERT, UPDATE или DELETE. При использовании в WITH оператора, изменяющего данные, (INSERT, UPDATE или DELETE) обычно добавляется предложение RETURNING. Заметьте, что именно результат RETURNING, а не лежащая таблица, изменяемая запросом, формирует временную таблицу, которую затем читает основной запрос. Если RETURNING опущено, оператор, тем не менее, выполняется, но не выдает никакого результата, так что на него нельзя сослаться как на таблицу в основном запросе.

Имя (без схемы) должно быть указано для каждого запроса WITH. Также можно задать необязательный список с именами колонок; если он опущен, имена колонок формируются из результата подзапроса.

Если указано RECURSIVE, подзапрос SELECT может ссылаться сам на себя по имени. Такой подзапрос должен иметь форму:

```
нерекурсивная_часть UNION [ ALL | DISTINCT ]  
рекурсивная_часть
```

где рекурсивная ссылка на сам запрос может находиться только справа от UNION. Для одного запроса допускается только одна рекурсивная ссылка на него же. Операторы, изменяющие данные, не могут быть рекурсивными, но результат рекурсивного запроса SELECT в таких операторах можно использовать.

Ещё одна особенность RECURSIVE в том, что запросы WITH могут быть

неупорядоченными: запрос может ссылаться на другой, идущий в списке после него. Однако циклические ссылки или взаимная рекурсия не поддерживаются. Без `RECURSIVE` запрос в `WITH` может ссылаться только на запросы того же уровня в `WITH`, предшествующие ему в списке `WITH`.

Ключевое свойство запросов `WITH` состоит в том, что они вычисляются один раз для всего основного запроса, даже если в основном запросе содержатся несколько ссылок на них. В частности, гарантируется, что операторы, изменяющие данные, будут выполняться ровно один раз, вне зависимости от того, будет ли их результат прочитан основным запросом и в каком объеме.

Основной запрос и все запросы `WITH`, условно говоря, выполняются одновременно. Это значит, что действие оператора, изменяющего данные в `WITH`, не будут видеть другие части запроса, кроме как прочитав его вывод `RETURNING`. Если два таких оператора попытаются изменить одну строку, результат будет неопределенным.

55.3.2. Предложение **FROM**

В предложении `FROM` перечисляются одна или несколько таблиц, служащих источниками данных для `SELECT`. Если указано несколько источников, результатом будет декартово произведение (перекрестное соединение) всех их строк. Но обычно в запрос добавляются уточняющие условия (в предложении `WHERE`), которые ограничивают набор строк небольшим подмножеством этого произведения.

Предложение `FROM` может содержать следующие элементы:

- `table_name`

Имя (возможно, дополненное схемой) существующей таблицы или представления. Если перед именем таблицы указано `ONLY`, считывается только заданная таблица. Без `ONLY` считывается и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно также добавить необязательное указание `*`, чтобы явно обозначить, что блокировка затрагивает и все дочерние таблицы;

- псевдоним

Альтернативное имя для элемента списка FROM. Этот псевдоним используется для краткости или для исключения неоднозначности с замкнутыми соединениями (когда одна таблица читается неоднократно). Когда задается псевдоним, он полностью скрывает настоящее имя таблицы или функции; например, при записи `FROM foo AS f`, в продолжении запроса `SELECT` к этому элементу FROM нужно обращаться по имени `f`, а не `foo`. Если задан псевдоним таблицы, за ним можно также написать список псевдонимов колонок, который определит альтернативные имена для колонок таблицы;

```
- [ TABLESAMPLE метод_выборки ( аргумент [, ...] )  
  [ REPEATABLE ( затравка ) ] ]
```

Предложение `TABLESAMPLE`, сопровождающее имя_таблицы, показывает, что для получения подмножества строк в этой таблице должен применяться указанный метод_выборки. Эта выборка предшествует применению любых других фильтров, например, в предложении `WHERE`. В стандартный дистрибутив Синергия-БД включены два метода выборки, `BERNOULLI` и `SYSTEM`; другие методы выборки можно установить в базу данных через расширения.

Методы выборки `BERNOULLI` и `SYSTEM` принимают единственный аргумент, определяющий, какой процент таблицы должен попасть в выборку, от 0 до 100. Этот аргумент может задаваться любым выражением со значением типа `real`. Другие методы выборки могут принимать дополнительные или другие параметры. Оба этих метода возвращают случайную выборку таблицы, содержащую примерно указанный процент строк таблицы. Метод `BERNOULLI` сканирует всю таблицу и выбирает или игнорирует отдельные строки независимо, с заданной вероятностью. Метод `SYSTEM` строит выборку на уровне блоков, определяя для каждого блока шанс его задействовать, и возвращает все строки из каждого задействуемого блока. Метод `SYSTEM` работает значительно быстрее `BERNOULLI`, когда выбирается небольшой процент строк, но он может выдавать менее случайную выборку таблицы из-за эффектов кучности.

В необязательном предложении `REPEATABLE` задаётся затравка – число или

выражение, задающее отправное значение для генератора случайных чисел в методе выборки. Значением затравки может быть любое отличное от NULL число с плавающей точкой. Два запроса, в которых указаны одинаковые значения затравки и аргумента, выдадут одну и ту же выборку таблицы при условии неизменности содержимого таблицы. Но с разными значениями затравки выборки обычно получаются разными. В отсутствие предложения REPEATABLE для каждого запроса выбирается новая случайная затравка. Заметьте, что некоторые дополнительные методы выборки не принимают предложение REPEATABLE и выдают разные выборки при каждом использовании;

- выборка

Предложение FROM может содержать вложенный запрос SELECT. Можно считать, что из его результата создается временная таблица на время выполнения основной команды SELECT. Заметьте, что вложенный запрос SELECT должен заключаться в скобки и для него должен задаваться псевдоним. Здесь также можно использовать команду VALUES;

- имя_запроса_WITH

На запрос WITH можно ссылаться по имени, как если бы имя запроса представляло имя таблицы. На самом деле запрос WITH скрывает любую реальную таблицу с тем же именем для основного запроса. Если необходимо обратиться к одноименной реальной таблице, можно дополнить имя этой таблицы именем схемы. Для этого имени можно задать псевдоним, так же, как и для имени таблицы;

- имя_функции

В предложении FROM могут содержаться вызовы функций. (Это особенно полезно для функций, возвращающих множества, но в принципе можно использовать любые функции.) Можно считать, что из результата функции создается временная таблица на время выполнения основной команды SELECT. Если вызов функции сопровождается необязательным предложением WITH ORDINALITY, после всех выдаваемых функцией колонок в вывод добавляется ещё одна колонка с номерами строк.

Псевдоним для функции можно задать так же, как и для таблицы. Если этот псевдоним задан, за ним можно также написать список псевдонимов колонок, который определит альтернативные имена для атрибутов составного типа результата функции, включая имя колонки, которая может быть добавлена предложением `ORDINALITY`.

Несколько вызовов функций можно объединить в одном элементе предложения `FROM`, заключив их в конструкцию `ROWS FROM( ... )`. Выводом такого элемента будет соединение первых строк всех функций, затем вторых строк, и т.д. Если одни функции выдают меньше строк, чем другие, недостающие данные заменяются `NULL`, так что общее число возвращаемых строк всегда будет равняться максимальному числу строк из возвращенных всеми функциями.

Если функция определена как возвращающая тип данных `record`, для нее нужно указать псевдоним или ключевое слово `AS`, за которым должен идти список определений колонок в форме (`имя_колонки тип_данных [ , ... ]`). Список определений колонок должен соответствовать фактическому количеству и типу колонок, возвращаемых функцией.

Если при использовании синтаксиса `ROWS FROM( ... )` одна из функций требует наличия списка определений колонок, этот список лучше разместить после вызова функции внутри `ROWS FROM( ... )`. Список определений колонок можно поместить после конструкции `ROWS FROM( ... )`, только если вызывается всего одна функция, а предложение `WITH ORDINALITY` отсутствует.

Чтобы использовать `ORDINALITY` со списком определений колонок, необходимо применить запись `ROWS FROM( ... )` и поместить список с определениями колонок внутрь `ROWS FROM( ... )`;

- `тип_соединения`

Один из следующих вариантов:

[`INNER`] `JOIN`

`LEFT` [`OUTER`] `JOIN`

RIGHT [OUTER] JOIN

FULL [OUTER] JOIN

CROSS JOIN

Для типов соединений INNER и OUTER необходимо указать условие соединения, а именно одно из предложений NATURAL, ON условие_соединения или USING (колонка_соединения [, ...]). Эти предложения описываются ниже. Для CROSS JOIN ни одно из этих предложений не допускается.

Предложение JOIN объединяет два элемента списка FROM, которые мы для простоты дальше будем называть «таблицами», хотя на самом деле это может быть любой объект, допустимый в качестве элемента FROM. Для определения порядка вложенности при необходимости следует использовать скобки. В отсутствие скобок предложения JOIN обрабатывается слева направо. В любом случае, JOIN связывает элементы сильнее, чем запятые, разделяющие элементы в списке FROM.

CROSS JOIN и INNER JOIN формируют простое декартово произведение, то же, что можно получить, указав две таблицы на верхнем уровне FROM, но ограниченное возможным условием соединения. Предложение CROSS JOIN равнозначно INNER JOIN ON (TRUE), то есть, никакие строки по условию не удаляются. Эти типы соединений введены исключительно для удобства записи, они не дают ничего такого, что нельзя было бы получить, используя просто FROM и WHERE.

LEFT OUTER JOIN возвращает все строки ограниченного декартова произведения (т.е. все объединенные строки, удовлетворяющие условию соединения) плюс все строки в таблице слева, для которых не находится строк в таблице справа, удовлетворяющих условию. Строка, взятая из таблицы слева, дополняется до полной ширины объединенной таблицы значениями NULL в колонках таблицы справа. Заметьте, что для определения, какие строки двух таблиц соответствуют друг другу, проверяется только условие самого предложения JOIN. Внешние условия проверяются позже.

RIGHT OUTER JOIN, напротив, возвращает все соединенные строки плюс одну строку для каждой строки справа, не имеющей соответствия слева (эта строка дополняется значениями NULL влево). Это предложение введено исключительно для удобства записи, так как его можно легко свести к LEFT OUTER JOIN, поменяв левую и правую таблицы местами.

FULL OUTER JOIN возвращает все соединенные строки плюс все строки слева, не имеющие соответствия справа, (дополненные значениями NULL вправо) плюс все строки справа, не имеющие соответствия слева (дополненные значениями NULL влево);

- ON условие_соединения

Задаваемое условие_соединения представляет собой выражение, выдающее значение типа boolean (как в предложении WHERE), которое определяет, какие строки считаются соответствующими при соединении;

- USING (колонка_соединения [, ...])

Предложение вида USING (a, b, ...) представляет собой сокращенную форму записи ON таблица_слева.a = таблица_справа.a AND таблица_слева.b = таблица_справа.b Кроме того, USING подразумевает, что в результат соединения будет включена только одна из пары равных колонок, но не обе;

- NATURAL

NATURAL представляет собой краткую запись USING со списком, в котором перечисляются все колонки двух таблиц, имеющие одинаковые имена;

- LATERAL

Ключевое слово LATERAL может предварять вложенный запрос SELECT в списке FROM. Оно позволяет обращаться в этом вложенном SELECT к колонкам элементов FROM, предшествующим ему в списке FROM. Без LATERAL все вложенные подзапросы SELECT обрабатываются независимо и не могут ссылаться на другие элементы списка FROM.

Слово LATERAL можно также добавить перед вызовом функции в списке FROM, но в этом случае оно будет избыточным, так как выражения с

функциями могут ссылаться на предыдущие элементы списка FROM в любом случае.

Элемент LATERAL может находиться на верхнем уровне списка FROM или в дереве JOIN. В последнем случае он может также ссылаться на любые элементы в левой части JOIN, справа от которого он находится.

Когда элемент FROM содержит ссылки LATERAL, запрос выполняется следующим образом: сначала для строки элемента FROM с целевыми колонками, или набора строк из нескольких элементов FROM, содержащих целевые колонки, вычисляется элемент LATERAL со значениями этих колонок. Затем результирующие строки обычным образом соединяются со строками, из которых они были вычислены. Эта процедура повторяется для всех строк исходных таблиц.

Таблица, служащая источником колонок, должна быть связана с элементом LATERAL соединением INNER или LEFT, в противном случае не образуется однозначно определяемый набор строк, из которого можно будет получать наборы строк для элемента LATERAL. Таким образом, хотя конструкция X RIGHT JOIN LATERAL Y синтаксически правильная, Y в ней не может обращаться к X.

55.3.3. Предложение WHERE

Необязательное предложение WHERE имеет общую форму:

WHERE условие

где условие – любое выражение, выдающее результат типа boolean. Любая строка, не удовлетворяющая этому условию, исключается из результата. Строка удовлетворяет условию, если оно возвращает true при подстановке вместо ссылок на переменные фактических значений из этой строки.

55.3.4. Предложение GROUP BY

Необязательное предложение GROUP BY имеет общую форму:

GROUP BY элемент_группирования [, ...]

GROUP BY собирает в одну строку все выбранные строки, выдающие

одинаковые значения для выражений группировки. В качестве выражения внутри элемента_группирования может выступать имя входного столбца, либо имя или порядковый номер выходного столбца (из списка элементов `SELECT`), либо произвольное значение, вычисляемое по значениям входных столбцов. В случае неоднозначности имя в `GROUP BY` будет восприниматься как имя входного, а не выходного столбца.

Если в элементе группирования задаётся `GROUPING SETS`, `ROLLUP` или `CUBE`, предложение `GROUP BY` в целом определяет некоторое число независимых наборов группирования. Это даёт тот же эффект, что и объединение подзапросов (с `UNION ALL`) с отдельными наборами группирования в их предложениях `GROUP BY`.

Агрегатные функции, если они используются, вычисляются по всем строкам, составляющим каждую группу, и в итоге выдают отдельное значение для каждой группы. Если агрегатные функции используются без предложения `GROUP BY`, запрос выполняется как с одной группой, включающей все выбранные строки. Набор строк, поступающих в каждую агрегатную функцию, можно дополнительно отфильтровать, добавив предложение `FILTER` к вызову агрегатной функции; за дополнительными сведениями обратитесь к . С предложением `FILTER` на вход агрегатной функции поступают только те строки, которые соответствуют заданному фильтру.

Когда в запросе присутствует предложение `GROUP BY` или какая-либо агрегатная функция, выражения в списке `SELECT` не могут обращаться к негруппируемым колонкам, кроме как в агрегатных функциях или в случае функциональной зависимости, так как иначе в негруппируемой колонке нужно было бы вернуть более одного возможного значения. Функциональная зависимость образуется, если группируемые колонки (или их подмножество) составляют первичный ключ таблицы, содержащей негруппируемую колонку.

Имейте в виду, что все агрегатные функции вычисляются перед «скалярными» выражениями в предложении `HAVING` или списке `SELECT`. Это значит, что например, с помощью выражения `CASE` нельзя обойти вычисление агрегатной

функции.

В настоящее время указания FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE нельзя задать вместе с GROUP BY.

55.3.5. Предложение HAVING

Необязательное предложение HAVING имеет общую форму:

HAVING условие

Здесь условие задается так же, как и для предложения WHERE.

HAVING исключает из результата строки групп, не удовлетворяющих условию. HAVING отличается от WHERE: WHERE фильтрует отдельные строки до применения GROUP BY, а HAVING фильтрует строки групп, созданных предложением GROUP BY. Каждая колонка, фигурирующая в условии, должна однозначно ссылаться на группируемую колонку, за исключением случаев, когда эта ссылка находится внутри агрегатной функции или негруппируемая колонка функционально зависит от группируемых.

В присутствии HAVING запрос превращается в группируемый, даже если GROUP BY отсутствует. То же самое происходит, когда запрос содержит агрегатные функции, но не предложение GROUP BY. Все выбранные строки считаются формирующими одну группу, а в списке SELECT и предложении HAVING можно обращаться к колонкам таблицы только из агрегатных функций. Такой запрос будет выдавать единственную строку, если результат условия HAVING – true, и ноль строк в противном случае.

В настоящее время указания FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE нельзя задать вместе с HAVING.

55.3.6. Предложение WINDOW

Необязательное предложение WINDOW имеет общую форму:

WINDOW имя_окна AS (определение_окна) [, ...]

Здесь имя_окна – это имя, на которое можно ссылаться из предложений OVER или последующих определений окон, а определение_окна имеет следующий вид:

[имя_существующего_окна]


```
[ PARTITION BY выражение [, ...] ]
[ ORDER BY выражение [ ASC | DESC | USING оператор ]
[ NULLS { FIRST | LAST } ] [, ...] ]
[ предложение_рамки ]
```

Если указано имя_существующего_окна, оно должно ссылаться на предшествующую запись в списке WINDOW; новое окно копирует предложение разбиения из этой записи, а также предложение сортировки, если оно присутствует. В этом случае для нового окна нельзя задать собственное предложение PARTITION BY, а ORDER BY можно указать, только если его не было у копируемого окна. Новое окно всегда использует собственное предложение рамки; в копируемом окне оно задаваться не должно.

Элементы списка PARTITION BY интерпретируется во многом так же, как и элементы Предложение GROUP BY, за исключением того, что это всегда простые выражения, но не имя или номер выходной колонки. Другое различие состоит в том, что эти выражения могут содержать вызовы агрегатных функций, которые не допускаются в обычном предложении GROUP BY. Здесь они допускаются потому, что формирование окна происходит после группировки и агрегирования.

Подобным образом, элементы списка ORDER BY интерпретируются во многом так же, как и элементы Предложение ORDER BY, за исключением того, что выражения в нем всегда принимаются как простые выражения, но не имя или номер выходной колонки.

Необязательное предложение_рамки определяет рамку окна для оконных функций, которые зависят от рамки (не все функции таковы). Рамка окна – это набор связанных строк для каждой строки запроса (называемой текущей строкой). В качестве предложения_рамки может задаваться:

```
{ RANGE | ROWS | GROUPS } начало_рамки
[ исключение_рамки ]
{ RANGE | ROWS | GROUPS } BETWEEN начало_рамки AND
конец_рамки [ исключение_рамки ]
```

Здесь начало_рамки и конец_рамки может задаваться как:

```
UNBOUNDED PRECEDING
смещение PRECEDING
```

CURRENT ROW

смещение FOLLOWING

UNBOUNDED FOLLOWING

И исключение _рамки может быть таким:

EXCLUDE CURRENT ROW

EXCLUDE GROUP

EXCLUDE TIES

EXCLUDE NO OTHERS

Если конец_рамки опущен, по умолчанию подразумевается CURRENT ROW.

В качестве начала_рамки нельзя задать UNBOUNDED FOLLOWING, а в качестве конца_рамки – UNBOUNDED PRECEDING, к тому же выбранный вариант конца_рамки не может стоять в приведенном списке выше варианта начала_рамки – например, комбинация RANGE BETWEEN CURRENT ROW AND значение PRECEDING не допускается.

По умолчанию рамка образуется предложением RANGE UNBOUNDED PRECEDING, что по сути то же, что RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW; оно устанавливает рамку так, что она включает все строки от начала раздела до последней строки, родственной текущей (строки, которые согласно указанному для окна предложению ORDER BY считаются равными текущей; если ORDER BY отсутствует, все строки считаются родственными). Вообще UNBOUNDED PRECEDING означает, что рамка начинается с первой строки раздела, а UNBOUNDED FOLLOWING означает, что рамка заканчивается на последней строке раздела, независимо от режима RANGE, ROWS или GROUPS. В режиме ROWS указание CURRENT ROW означает, что рамка начинается или заканчивается текущей строкой; но в режиме RANGE или GROUPS оно означает, что рамка начинается или заканчивается первой или последней строкой, родственной текущей, согласно порядку ORDER BY. Варианты смещение PRECEDING и смещение FOLLOWING означают разное в зависимости от режима рамки. В режиме ROWS целочисленное смещение определяет сдвиг, с которым начало рамки позиционируется перед текущей строкой, а конец рамки – после текущей строки. В

режиме GROUPS целочисленное смещение аналогичным образом определяет сдвиг относительно группы строк, родственных текущей, где *группа родственных строк* – группа строк, считающихся равными согласно предложению ORDER BY для данного окна. В режиме RANGE для указания смещения необходимо присутствие в определении окна ровно одного столбца ORDER BY. Тогда рамка будет содержать те строки, в которых значение упорядочивающего столбца не более чем на смещение меньше (для PRECEDING) или больше (для FOLLOWING) значения упорядочивающего столбца в текущей строке. В этом случае тип данных выражения смещение зависит от типа данных упорядочивающего столбца. Для числовых столбцов это обычно тот же числовой тип, а для столбцов с типом дата/время – тип interval. Во всех этих случаях значение смещения должно быть отличным от NULL и неотрицательным. Кроме того, хотя смещение не обязательно должно быть простой константой, оно не может содержать переменные, агрегатные или оконные функции.

Дополнение `исключение_рамки` позволяет исключить из рамки строки, которые окружают текущую строку, даже если они должны быть включены согласно указаниям, определяющим начало и конец рамки. EXCLUDE CURRENT ROW исключает из рамки текущую строку. EXCLUDE GROUP исключает из рамки текущую строку и родственные ей согласно порядку сортировки. EXCLUDE TIES исключает из рамки все родственные строки для текущей, но не собственно текущую строку. EXCLUDE NO OTHERS просто явно выражает поведение по умолчанию – не исключает ни текущую строку, ни родственные ей.

Учтите, что в режиме ROWS могут выдаваться непредсказуемые результаты, если согласно порядку, заданному в ORDER BY, строки сортируются неоднозначно. Режимы RANGE и GROUPS предусмотрены для того, чтобы строки, являющиеся родственными в порядке ORDER BY, обрабатывались одинаково: все строки определённой группы попадут в одну рамку или будут исключены из неё.

Предложение WINDOW применяется для управления поведением оконных функций, фигурирующих в запросе, в Список SELECT или Предложение ORDER BY. Эти функции могут обращаться к элементам WINDOW по именам в своих

предложениях `OVER`. При этом элементы `WINDOW` не обязательно задействовать в запросе; если они не используются, они просто игнорируются. Оконные функции можно использовать вовсе без элементов `WINDOW`, так как в вызове оконной функции можно задать определение окна непосредственно в предложении `OVER`. Однако предложение `WINDOW` позволяет сократить текст запроса, когда одно и то же определение окна применяется при вызове нескольких оконных функций.

В настоящее время указания `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` и `FOR KEY SHARE` нельзя задать вместе с `WINDOW`.

55.3.7. Список `SELECT`

Список `SELECT` (между ключевыми словами `SELECT` и `FROM`) содержит выражения, которые формируют выходные строки оператора `SELECT`. Эти выражения могут обращаться (и обычно обращаются) к колонкам, вычисленным в предложении `FROM`.

Так же, как в таблице, каждая выходная колонка `SELECT` имеет имя. В простом предложении `SELECT` это имя просто помечает колонку при выводе, но когда `SELECT` представляет собой подзапрос большого запроса, это имя большой запрос видит как имя колонки виртуальной таблицы, созданной подзапросом. Чтобы задать имя для выходной колонки, нужно написать `AS выходное_имя` после выражения колонки. Слово `AS` можно опустить, но только если желаемое выходное имя не совпадает с каким-либо ключевым словом Синергия-БД. Чтобы не зависеть от появления новых ключевых слов в будущем, рекомендуется всегда писать `AS`, либо заключать имя в двойные кавычки. Если имя колонки не задать, Синергия-БД выберет его автоматически. Если выражение колонки представляет собой просто ссылку на колонку, то выбранное таким образом имя будет совпадать с именем колонки. В более сложных случаях может использоваться имя функции или типа, либо в отсутствие других вариантов система может сгенерировать имя вроде `?column?`.

По имени выходной колонки можно обратиться к её значению в предложениях `ORDER BY` и `GROUP BY`, но не в `WHERE` или `HAVING`; в них вместо имени надо

записывать все выражение.

Вместо выражения в выходном списке можно указать *, что будет обозначать все колонки выбранных строк. Кроме того, можно записать имя_таблицы.* как краткое обозначение всех колонок, получаемых из данной таблицы. В этих случаях нельзя задать новые имена колонок с помощью AS; именами выходных колонок будут имена колонок в таблице.

Согласно стандарту SQL, выражения в выходном списке должны вычисляться до применения DISTINCT, ORDER BY или LIMIT. Это, очевидно, необходимо для DISTINCT, так как иначе не будет ясно, какие значения должны выдаваться как уникальные. Однако во многих случаях выходные выражения удобнее вычислять после ORDER BY и LIMIT; в частности, если в выходном списке содержатся изменчивые или дорогостоящие функции. В этом случае порядок вычисления функций оказывается более интуитивным, а для строк, которые не попадут в результат, не будут производиться вычисления. Синергия-БД фактически будет вычислять выходные выражения после сортировки и ограничения их количества, если эти выражения не фигурируют в DISTINCT, ORDER BY или GROUP BY. Например, в запросе `SELECT f(x) FROM tab ORDER BY 1` функция `f(x)`, несомненно, должна вычисляться перед сортировкой. Выходные выражения, содержащие функции, возвращающие множества, фактически вычисляются после сортировки и до ограничения количества строк, так что LIMIT будет отбрасывать строки, выдаваемые функцией, возвращающей множество.

55.3.8. Предложение DISTINCT

Если указано `SELECT DISTINCT`, все повторяющиеся строки исключаются из результирующего набора (из каждой группы дубликатов остается одна строка). `SELECT ALL` делает противоположное: сохраняет все строки; это поведение по умолчанию.

`SELECT DISTINCT ON ( выражение [, ...] )` сохраняет только первую строку из каждого набора строк, для которого данное выражение дает одинаковые значения. Выражения `DISTINCT ON` обрабатываются по тем же

правилам, что и выражения ORDER BY. Заметьте, что «первая строка» каждого набора непредсказуема, если только не применяется предложение ORDER BY, определяющее, какие строки должны быть первыми. Например:

```
SELECT DISTINCT ON (location) location, time, report
FROM weather_reports
ORDER BY location, time DESC;
```

возвращает самую последнюю сводку погоды для каждого местоположения. Но если бы мы не добавили ORDER BY, чтобы значения времени убывали, мы бы получили сводки по местоположениям от непредсказуемого времени.

Выражения DISTINCT ON должны соответствовать самым левым выражениям в ORDER BY. Предложение ORDER BY обычно содержит и другие выражения, которые определяют желаемый порядок строк в каждой группе DISTINCT ON.

В настоящее время указания FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE нельзя задать вместе с DISTINCT.

55.3.9. Предложение UNION+

Предложение UNION имеет следующую общую форму:

```
оператор_SELECT UNION [ ALL | DISTINCT ] оператор_SELECT
```

Здесь оператор_SELECT – это любой подзапрос SELECT без предложений ORDER BY, LIMIT, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE. ORDER BY и LIMIT можно добавить к вложенному выражению, если оно заключено в скобки. Без скобок эти предложения будут восприняты как применяемые к результату UNION, а не к выражению в его правой части.

Оператор UNION вычисляет объединение множеств всех строк, возвращенных заданными запросами SELECT. Строка оказывается в объединении двух наборов результатов, если она присутствует минимум в одном наборе. Два оператора SELECT, представляющие прямые операнды UNION, должны выдавать одинаковое число колонок, а типы соответствующих колонок должны быть совместимыми.

Результат UNION не будет содержать повторяющихся строк, если не указан

параметр ALL. ALL предотвращает исключение дубликатов. Таким образом, UNION ALL обычно работает значительно быстрее, чем UNION; поэтому, везде, где возможно, следует указывать ALL. DISTINCT можно записать явно, чтобы обозначить, что дублирующиеся строки должны удаляться (это поведение по умолчанию).

При использовании в одном запросе SELECT нескольких операторов UNION они вычисляются слева направо, если иной порядок не определяется скобками.

В настоящее время указания FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE нельзя задать ни для результата UNION, ни для любого из подзапросов UNION.

55.3.10. Предложение INTERSECT

Предложение INTERSECT имеет следующую общую форму:

```
оператор_SELECT INTERSECT [ ALL | DISTINCT ]  
оператор_SELECT
```

Здесь оператор_SELECT – это любой подзапрос SELECT без предложений ORDER BY, LIMIT, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE.

Оператор INTERSECT вычисляет пересечение множеств всех строк, возвращенных заданными запросами SELECT. Строка оказывается в пересечении двух наборов результатов, если она присутствует в обоих наборах.

Результат INTERSECT не будет содержать повторяющихся строк, если не указан параметр ALL. С параметром ALL строка, повторяющаяся m раз в левой таблице и n раз в правой, будет выдана в результирующем наборе $\min(m,n)$ раз. DISTINCT можно записать явно, чтобы обозначить, что дублирующиеся строки должны удаляться (это поведение по умолчанию).

При использовании в одном запросе SELECT нескольких операторов INTERSECT они вычисляются слева направо, если иной порядок не диктуется скобками. INTERSECT связывает свои подзапросы сильнее, чем UNION. Другими словами, $A \text{ UNION } B \text{ INTERSECT } C$ будет восприниматься как $A \text{ UNION } (B$

INTERSECT C).

В настоящее время указания FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE нельзя задать ни для результата INTERSECT, ни для любого из подзапросов INTERSECT.

55.3.11. Предложение EXCEPT

Предложение EXCEPT имеет следующую общую форму:

```
оператор_SELECT EXCEPT [ ALL | DISTINCT ]
оператор_SELECT
```

Здесь оператор_SELECT – это любой подзапрос SELECT без предложений ORDER BY, LIMIT, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE.

Оператор EXCEPT вычисляет набор строк, которые присутствуют в результате левого запроса SELECT, но отсутствуют в результате правого.

Результат EXCEPT не будет содержать повторяющихся строк, если не указан параметр ALL. С параметром ALL строка, повторяющаяся m раз в левой таблице и n раз в правой, будет выдана в результирующем наборе $\max(m-n, 0)$ раз. DISTINCT можно записать явно, чтобы обозначить, что дублирующиеся строки должны удаляться (это поведение по умолчанию).

При использовании в одном запросе SELECT нескольких операторов EXCEPT они вычисляются слева направо, если иной порядок не диктуется скобками. EXCEPT связывает свои подзапросы так же сильно, как UNION.

В настоящее время указания FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE нельзя задать ни для результата EXCEPT, ни для любого из подзапросов EXCEPT.

55.3.12. Предложение ORDER BY

Необязательное предложение ORDER BY имеет следующую общую форму:

```
ORDER BY выражение [ ASC | DESC | USING оператор ]
[ NULLS { FIRST | LAST } ] [, ...]
```

Предложение ORDER BY указывает, что строки результата должны

сортироваться согласно заданным выражениям. Если две строки дают равные значения для самого левого выражения, проверяется следующее выражение и т.д. Если их значения оказываются равными для всех заданных выражений, строки возвращаются в порядке, определяемом реализацией.

В качестве выражения может задаваться имя или порядковый номер выходной колонки (элемента списка `SELECT`), либо произвольное выражение со значениями входных колонок.

Порядковым номером в данном случае считается последовательный номер (при нумерации слева направо) позиции выходной колонки. Возможность указать порядковый номер позволяет выполнить сортировку по колонке, не имеющей уникального имени. В принципе это не абсолютно необходимо, так как выходной колонке всегда можно присвоить имя, воспользовавшись предложением `AS`.

В предложении `ORDER BY` также можно использовать произвольные выражения, в том числе, и с колонками, отсутствующими в списке результатов `SELECT`. Таким образом, следующий оператор вполне корректен:

```
SELECT name FROM distributors ORDER BY code;
```

Однако, если `ORDER BY` применяется к результату `UNION`, `INTERSECT` или `EXCEPT`, в нем можно задать только имя или номер выходной колонки, но не выражение.

Если в качестве выражения `ORDER BY` задано простое имя, которому соответствует и выходная, и входная колонка, то `ORDER BY` будет воспринимать его как имя выходной колонки. Этот выбор противоположен тому, что делает `GROUP BY` в такой же ситуации. Такая несогласованность допущена для соответствия стандарту `SQL`.

Дополнительно после любого выражения в предложении `ORDER BY` можно добавить ключевое слово `ASC` (по возрастанию) или `DESC` (по убыванию). По умолчанию подразумевается `ASC`. Кроме того, можно задать имя специфического оператора сортировки в предложении `USING`. Оператор сортировки должен быть членом «меньше» или «больше» некоторого семейства операторов `B`-дерева. `ASC` обычно равнозначно `USING <` и `DESC` обычно равнозначно `USING >`. Хотя

создатель нестандартного типа данных может определить по-другому порядок сортировки по умолчанию и поставить ему в соответствие операторы с другими именами.

Если указано `NULLS LAST`, значения `NULL` при сортировке оказываются после значений не `NULL`; с указанием `NULLS FIRST` значения `NULL` оказываются перед значениями не `NULL`. Если не указано ни то, ни другое, по умолчанию подразумевается `NULLS LAST` при явно или неявно выбранном порядке `ASC`, либо `NULLS FIRST` при порядке `DESC` (то есть по умолчанию считается, что значения `NULL` больше значений не `NULL`). С предложением `USING` порядок `NULL` по умолчанию зависит от того, является ли указанный оператор оператором «меньше» или «больше».

Заметьте, что параметры сортировки применяются только к тому выражению, за которым они следуют; в частности, `ORDER BY x, y DESC` означает не то же самое, что `ORDER BY x DESC, y DESC`.

Данные символьных строк сортируются согласно правилу сортировки, установленному для сортируемой колонки. При необходимости это правило можно переопределить, добавив предложение `COLLATE` в выражение, например так: `ORDER BY mycolumn COLLATE «en_US»`.

55.3.13. Предложение `LIMIT`

Предложение `LIMIT` состоит из двух независимых вложенных предложений:

`LIMIT { число | ALL }`

`OFFSET начало`

Здесь `число` определяет максимальное количество строк, которое должно быть выдано, тогда как `начало` определяет, сколько строк нужно пропустить, прежде чем начать выдавать строки. Когда указаны оба значения, сначала строки пропускаются в количестве, заданном значением `начало`, а затем следующие строки выдаются в количестве, не превышающем значения `число`.

Если результатом выражения `число` оказывается `NULL`, предложение воспринимается как `LIMIT ALL`, т.е. число строк не ограничивается. Если `начало`

принимает значение NULL, предложение воспринимается как OFFSET 0.

SQL:2008 вводит другой синтаксис для получения того же результата, и его так же поддерживает Синергия-БД. Он выглядит так:

```
OFFSET начало { ROW | ROWS }  
FETCH { FIRST | NEXT } [ число ] { ROW | ROWS } ONLY
```

В этом синтаксисе, чтобы подставить в начало или число что-либо сложнее простой целочисленной константы, необходимо заключить это выражение в скобки. Если число опускается в предложении FETCH, оно принимает значение 1. ROW и ROWS так же, как и FIRST и NEXT являются избыточными словами, которые не влияют на действие этих предложений. Согласно стандарту, предложение OFFSET должно предшествовать предложению FETCH, если присутствуют они оба; но Синергия-БД менее строг и допускает любой порядок.

Применяя LIMIT, имеет смысл использовать также предложение ORDER BY, чтобы строки результата выдавались в определенном порядке. Иначе будут возвращаться непредсказуемые подмножества строк запроса – вы можете запросить строки с десятой по двадцатую, но какой порядок вы имеете в виду? Порядок будет неизвестен, если не добавить ORDER BY.

Планировщик запроса учитывает ограничение LIMIT, строя план выполнения запроса, поэтому, вероятнее всего, планы (а значит и порядок строк) будут меняться при разных LIMIT и OFFSET. Таким образом, различные значения LIMIT/OFFSET, выбирающие разные подмножества результатов запроса, приведут к несогласованности результатов, если не установить предсказуемую сортировку с помощью ORDER BY. Это не ошибка, а неизбежное следствие того, что SQL не гарантирует вывод результатов запроса в некотором порядке, если порядок не определен явно предложением ORDER BY.

Возможно даже, что при повторном выполнении одного и того же запроса с LIMIT будут получены разные подмножества строк таблицы, если предложение ORDER BY не диктует выбор определенного подмножества. Опять же, это не ошибка; в данном случае детерминированность результата просто не гарантируется.

55.3.14. Предложение блокировки

Предложения блокировки включают в себя `FOR UPDATE`, `FOR NO KEY UPDATE`, `FOR SHARE` и `FOR KEY SHARE`; они влияют на то, как `SELECT` блокирует строки, получаемые из таблицы.

Предложение блокировки имеет следующую общую форму:

```
FOR вариант_блокировки [ OF имя_таблицы [, ...] ]  
[ NOWAIT | SKIP LOCKED ]
```

Здесь вариант_блокировки может быть следующим:

`UPDATE`

`NO KEY UPDATE`

`SHARE`

`KEY SHARE`

Чтобы операция не ждала завершения других транзакций, к блокировке можно добавить указание `NOWAIT` или `SKIP LOCKED`. `NOWAIT` оператор выдаёт ошибку, а не ждёт, если выбранную строку нельзя заблокировать немедленно. С указанием `SKIP LOCKED` выбранные строки, которые нельзя заблокировать немедленно, пропускаются. При этом формируется несогласованное представление данных, так что этот вариант не подходит для общего применения, но может использоваться для исключения блокировок при обращении множества потребителей к таблице типа очереди. Заметьте, что указания `NOWAIT` и `SKIP LOCKED` применяются только к блокировкам на уровне строк – необходимая блокировка `ROW SHARE` уровня таблицы запрашивается обычным способом. Если требуется запросить блокировку уровня таблицы без ожидания, можно сначала выполнить команду `LOCK` с указанием `NOWAIT`.

Если в предложении блокировки указаны определенные таблицы, блокироваться будут только строки, получаемые из этих таблиц; другие таблицы, задействованные в `SELECT`, будут прочитаны как обычно. Предложение блокировки без списка таблиц затрагивает все таблицы, задействованные в этом операторе. Если предложение блокировки применяется к представлению или подзапросу, оно затрагивает все таблицы, которые используются в представлении или подзапросе. Однако эти предложения не применяются к запросам `WITH`, к которым обращается

основной запрос. Если требуется установить блокировку строк в запросе `WITH`, предложение блокировки нужно указать непосредственно в этом запросе `WITH`.

В случае необходимости задать для разных таблиц разное поведение блокировки, в запрос можно добавить несколько предложений. Если при этом одна и та же таблица упоминается (или неявно затрагивается) в нескольких предложениях блокировки, блокировка устанавливается так, как если бы была указана только одна, самая сильная из них. Подобным образом, если в одном из предложений указано `NOWAIT`, для этой таблицы блокировка будет запрашиваться без ожидания. В противном случае она будет обработана в режиме `SKIP LOCKED`, если он выбран в любом из затрагивающих её предложений.

Предложения блокировки не могут применяться в контекстах, где возвращаемые строки нельзя четко связать с отдельными строками таблицы; например, блокировка неприменима при агрегировании.

Когда предложение блокировки находится на верхнем уровне запроса `SELECT`, блокируются именно те строки, которые возвращаются запросом; в случае с запросом объединения, блокировке подлежат строки, из которых составляются возвращаемые строки объединения. В дополнение к этому, заблокированы будут строки, удовлетворяющие условиям запроса на момент создания снимка запроса, хотя они не будут возвращены, если с момента снимка они изменятся и перестанут удовлетворять условиям. Если применяется `LIMIT`, блокировка прекращается, как только будет получено достаточное количество строк для удовлетворения лимита (но заметьте, что строки, пропускаемые указанием `OFFSET`, будут блокироваться). Подобным образом, если предложение блокировки применяется в запросе курсора, блокироваться будут только строки, фактически полученные или пройденные курсором.

Когда предложение блокировки находится в подзапросе `SELECT`, блокировке подлежат те строки, которые будут получены внешним запросом от подзапроса. Таких строк может оказаться меньше, чем можно было бы предположить, проанализировав только сам подзапрос, так как условия из внешнего запроса могут способствовать оптимизации выполнения подзапроса. Например, запрос

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss  
WHERE col1 = 5;
```

заблокирует только строки, в которых `col1 = 5`, при том, что в такой записи условие не относится к подзапросу.

Предыдущие версии не могли сохранить блокировку, которая была повышена последующей точкой сохранения. Например, этот код:

```
BEGIN;  
SELECT * FROM mytable WHERE key = 1 FOR UPDATE;  
SAVEPOINT s;  
UPDATE mytable SET ... WHERE key = 1;  
ROLLBACK TO s;
```

не мог сохранить блокировку `FOR UPDATE` после `ROLLBACK TO`.

Примечание. Возможно, что команда `SELECT`, работающая на уровне изоляции `READ COMMITTED` и применяющая предложение `ORDER BY` вместе с блокировкой, будет возвращать строки не по порядку. Это связано с тем, что `ORDER BY` выполняется в первую очередь. Эта команда отсортирует результат, но затем может быть заблокирована, пытаясь получить блокировку одной или нескольких строк. К моменту, когда блокировка `SELECT` будет снята, некоторые из сортируемых столбцов могут уже измениться, в результате чего их порядок может быть нарушен (хотя они были упорядочены для исходных значений). При необходимости обойти эту проблему можно поместить `FOR UPDATE/SHARE` в подзапрос, например так:

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss  
ORDER BY column1;
```

Заметьте, что в результате это приведёт к блокированию всех строк в `mytable`, тогда как указание `FOR UPDATE` на верхнем уровне могло бы заблокировать только фактически возвращаемые строки. Это может значительно повлиять на производительность, особенно в сочетании `ORDER BY` с `LIMIT` или другими ограничениями. Таким образом, этот приём рекомендуется, только если ожидается параллельное изменение сортируемых столбцов, а результат должен быть строго отсортирован.

На уровнях изоляции `REPEATABLE READ` и `SERIALIZABLE` это приведёт к ошибке сериализации (с `SQLSTATE` равным '40001'), так что на этих уровнях получить строки не по порядку невозможно.

55.3.15. TABLE

Команда

TABLE имя

равнозначна

SELECT * FROM имя

Ее можно применять в качестве команды верхнего уровня или как более краткую запись внутри сложных запросов. С командой TABLE могут использоваться только предложения WITH, UNION, INTERSECT, EXCEPT, ORDER BY, LIMIT, OFFSET, FETCH и предложения блокировки FOR; предложение WHERE и какие-либо формы агрегирования не поддерживаются.

55.4.Примеры

Соединение таблицы films с таблицей distributors:

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
FROM distributors d, films f
WHERE f.did = d.did
```

kind	title	did	name	date_prod
	The Third Man	101	British Lion	1949-12-23
Drama				
	The African Queen	101	British Lion	1951-08-11
Romantic				
...				

Суммирование значений колонки len (продолжительность) для всех фильмов и группирование результатов по колонке kind (типу фильма):

```
SELECT kind, sum(len) AS total FROM films GROUP BY kind;
```

kind	total
Action	07:34

Comedy		02:58
Drama		14:28
Musical		06:42
Romantic		04:38

Суммирование значений колонки len для всех фильмов, группирование результатов по колонке kind и вывод только тех групп, общая продолжительность которых меньше 5 часов:

```
SELECT kind, sum(len) AS total
FROM films
GROUP BY kind
HAVING sum(len) < interval '5 hours';
```

kind		total
-----+-----		
Comedy		02:58
Romantic		04:38

Следующие два запроса демонстрируют равнозначные способы сортировки результатов по содержимому второй колонки (name):

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;
```

did		name
-----+-----		
109		20th Century Fox
110		Bavaria Atelier
101		British Lion
107		Columbia
102		Jean Luc Godard
113		Luso films
104		Mosfilm
103		Paramount
106		Toho
105		United Artists

111 | Walt Disney

112 | Warner Bros.

108 | Westward

Следующий пример показывает объединение таблиц distributors и actors, ограниченное именами, начинающимися с буквы W в каждой таблице. Интерес представляют только неповторяющиеся строки, поэтому ключевое слово ALL опущено.

distributors:	actors:
did name	id name
-----+-----	-----+-----
108 Westward	1 Woody Allen
111 Walt Disney	2 Warren Beatty
112 Warner Bros.	3 Walter Matthau
...	...

```
SELECT distributors.name
      FROM distributors
      WHERE distributors.name LIKE 'W%'
UNION
```

```
SELECT actors.name
      FROM actors
      WHERE actors.name LIKE 'W%';
```

```
      name
-----
Walt Disney
Walter Matthau
Warner Bros.
Warren Beatty
Westward
Woody Allen
```

Этот пример показывает, как использовать функцию в предложении FROM, со

списком определений колонок и без него:

```
CREATE FUNCTION distributors(int) RETURNS SETOF
distributors AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM distributors(111);
 did |      name
-----+-----
 111 | Walt Disney
```

```
CREATE FUNCTION distributors_2(int) RETURNS SETOF record
AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM distributors_2(111) AS (f1 int, f2 text);
 f1  |      f2
-----+-----
 111 | Walt Disney
```

Пример функции с добавленной колонкой нумерации:

```
SELECT * FROM unnest(ARRAY['a','b','c','d','e','f'])
WITH ORDINALITY;
```

```
unnest | ordinality
-----+-----
 a     |          1
 b     |          2
 c     |          3
 d     |          4
 e     |          5
 f     |          6
```

(6 rows)

Этот пример показывает, как использовать простое предложение WITH:

```
WITH t AS (
```

```

SELECT random() as x FROM generate_series(1, 3)
)
SELECT * FROM t
UNION ALL
SELECT * FROM t

```

```

      x
-----
0.534150459803641
0.520092216785997
0.0735620250925422
0.534150459803641
0.520092216785997
0.0735620250925422

```

Заметьте, что запрос WITH выполняется всего один раз, поэтому мы получаем два одинаковых набора по три случайных значения.

В этом примере WITH RECURSIVE применяется для поиска всех подчиненных Мери (непосредственных или косвенных) и вывода их уровня косвенности в таблице с информацией только о непосредственных подчиненных:

```

WITH RECURSIVE employee_recursive(distance,
employee_name, manager_name) AS (
    SELECT 1, employee_name, manager_name
    FROM employee
    WHERE manager_name = 'Mary'
    UNION ALL
    SELECT er.distance + 1, e.employee_name,
e.manager_name
    FROM employee_recursive er, employee e
    WHERE er.employee_name = e.manager_name
)
SELECT distance, employee_name FROM employee_recursive;

```

Заметьте, что это типичная форма рекурсивных запросов: начальное условие, последующий UNION, а затем рекурсивная часть запроса. Убедитесь в том, что

рекурсивная часть запроса в конце концов перестанет возвращать строки, иначе запрос окажется в бесконечном цикле.

В этом примере используется `LATERAL` для применения функции `get_product_names()`, возвращающей множество, для каждой строки таблицы `manufacturers`:

```
SELECT m.name AS mname, pname
FROM manufacturers m, LATERAL get_product_names(m.id)
pname;
```

Производители, с которыми в данный момент не связаны никакие продукты, не попадут в результат, так как это внутреннее соединение. Если бы мы захотели включить названия и этих производителей, мы могли бы сделать так:

```
SELECT m.name AS mname, pname
FROM manufacturers m LEFT JOIN LATERAL
get_product_names(m.id) pname ON true;
```

55.5.Совместимость

Разумеется, оператор `SELECT` совместим со стандартом SQL. Однако не все описанные в стандарте возможности реализованы, а некоторые, наоборот, являются расширениями.

55.5.1. Необязательное предложение FROM

Синергия-БД разрешает опустить предложение `FROM`. Это позволяет очень легко вычислять результаты простых выражений:

```
SELECT 2+2;
```

```
?column?
-----
4
```

Некоторые другие базы данных SQL не допускают этого, требуя задействовать в `SELECT` фиктивную таблицу с одной строкой.

Заметьте, что если предложение `FROM` не указано, запрос не может обращаться ни к каким таблицам базы данных. Например, следующий запрос недопустим:

```
SELECT distributors.* WHERE distributors.name =
```

'Westward';

55.5.2. Пустые списки SELECT

Список выходных выражений после SELECT может быть пустым, что в результате даст таблицу без колонок. Стандарт SQL не считает такой синтаксис допустимым, но Синергия-БД допускает его, так как это согласуется с возможностью иметь таблицы с нулем колонок. Однако, когда используется DISTINCT, пустой список не допускается.

55.5.3. Необязательное ключевое слово AS

В стандарте SQL необязательное ключевое слово AS можно опустить перед именем выходной колонки, если это имя является допустимым именем колонки (то есть не совпадает с каким-либо зарезервированным ключевым словом). Синергия-БД несколько более строг: AS требуется, если имя колонки совпадает с любым ключевым словом, зарезервированным или нет. Тем не менее, рекомендуется использовать AS или заключать имена выходных колонок в кавычки, во избежание конфликтов, возможных при появлении в будущем новых ключевых слов.

В списке FROM и стандарт, и Синергия-БД позволяют опускать AS перед псевдонимом, который является незарезервированным ключевым словом. Но для имен выходных колонок это не подходит из-за синтаксической неоднозначности.

55.5.4. ONLY и наследование

Стандарт SQL требует заключать в скобки имя таблицы после ONLY, например `SELECT * FROM ONLY (tab1), ONLY (tab2) WHERE . . .`. Синергия-БД считает эти скобки необязательными.

Синергия-БД позволяет добавлять в конце *, чтобы явно обозначить, что дочерние таблицы включаются в рассмотрение, в отличие от поведения с ONLY. Стандарт не позволяет этого.

55.5.5. Ограничения предложения TABLESAMPLE

Предложение TABLESAMPLE в настоящий момент принимается только для обычных таблиц и материализованных представлений. Однако согласно стандарту

SQL оно должно применяться к любым элементам списка FROM.

55.5.6. Вызовы функций в предложении FROM

Синергия-БД позволяет записать вызов функции непосредственно в виде элемента списка FROM. В стандарте SQL такой вызов функции требуется помещать во вложенный SELECT; то есть, запись FROM функция(...) псевдоним примерно равнозначна записи FROM LATERAL (SELECT функция(...)) псевдоним. Заметьте, что указание LATERAL считается неявным; это связано с тем, что стандарт требует поведения LATERAL для элемента UNNEST() в предложении FROM. Синергия-БД обрабатывает UNNEST() так же, как и другие функции, возвращающие множества.

55.5.7. Пространства имен в GROUP BY и ORDER BY

В стандарте SQL-92 предложение ORDER BY может содержать ссылки только на выходные колонки по именам или номерам, тогда как GROUP BY может содержать выражения с именами только входных колонок. Синергия-БД расширяет оба эти предложения, позволяя также применять другие варианты (но если возникает неоднозначность, он разрешает её согласно стандарту). Синергия-БД также позволяет задавать произвольные выражения в обоих предложениях. Заметьте, что имена, фигурирующие в выражениях, всегда будут восприниматься как имена входных, а не выходных колонок.

В SQL:1999 и более поздних стандартах введено несколько другое определение, которое не полностью совместимо с SQL-92. Однако в большинстве случаев Синергия-БД будет интерпретировать выражение ORDER BY или GROUP BY так, как требует SQL:1999.

55.5.8. Функциональные зависимости

Синергия-БД распознает функциональную зависимость (что позволяет опускать колонки в GROUP BY), только когда первичный ключ таблицы присутствует в списке GROUP BY. В стандарте SQL оговариваются дополнительные условия, которые следует учитывать.

55.5.9. LIMIT и OFFSET

Предложения LIMIT и OFFSET относятся к специфическим особенностям Синергия-БД и поддерживаются также в MySQL. В стандарте SQL:2008 для той же цели вводятся предложения OFFSET ... FETCH {FIRST|NEXT} ..., рассмотренные ранее в Предложение LIMIT. Этот синтаксис также используется в IBM DB2.

55.5.10. FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE, FOR KEY SHARE

Хотя указание FOR UPDATE есть в стандарте SQL, стандарт позволяет использовать его только в предложении DECLARE CURSOR. Синергия-БД допускает его использование в любом запросе SELECT, а также в любых подзапросах SELECT, но это является расширением. Варианты FOR NO KEY UPDATE, FOR SHARE и FOR KEY SHARE, а также указание NOWAIT в стандарте отсутствуют.

55.5.11. Изменение данных в WITH

Синергия-БД разрешает использовать INSERT, UPDATE и DELETE в качестве запросов WITH. Стандарт SQL этого не предусматривает.

55.5.12. Нестандартные предложения

DISTINCT ON (...) – расширение стандарта SQL.

ROWS FROM(...) – расширение стандарта SQL.

56. SELECT INTO

SELECT INTO – создать таблицу из результатов запроса.

56.1.Синтаксис

```
[ WITH [ RECURSIVE ] запрос_WITH [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( выражение [, ...] ) ] ]
      * | выражение [ [ AS ] имя_результата ] [, ...]
      INTO [ TEMPORARY | TEMP | UNLOGGED ] [ TABLE ]
новая_таблица
      [ FROM элемент_FROM [, ...] ]
```

```

[ WHERE условие ]
[ GROUP BY выражение [, ...] ]
[ HAVING условие [, ...] ]
[ WINDOW имя_окна AS ( определение_окна ) [, ...] ]
[ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ]
выборка ]
[ ORDER BY выражение [ ASC | DESC | USING оператор ]
[ NULLS { FIRST | LAST } ] [, ...] ]
[ LIMIT { число | ALL } ]
[ OFFSET начало [ ROW | ROWS ] ]
[ FETCH { FIRST | NEXT } [ число ] { ROW | ROWS }
ONLY ]
[ FOR { UPDATE | SHARE } [ OF table_name [, ...] ]
[ NOWAIT ] [...] ]

```

56.2.Описание

SELECT INTO создает новую таблицу и заполняет её данными, полученными из запроса. Данные не передаются клиенту, как с обычной командой SELECT. Колонки новой таблицы получают имена и типы данных, связанные с выходными колонками SELECT.

56.3.Параметры

TEMPORARY или TEMP

Если указано, создаваемая таблица будет временной.

UNLOGGED

Если указано, создаваемая таблица будет нежурналируемой.

новая_таблица

Имя (возможно, дополненное схемой) создаваемой таблицы.

Все другие параметры подробно описываются в SELECT.

56.4.Замечания

Команда SELECT INTO действует подобно CREATE TABLE AS, но рекомендуется использовать CREATE TABLE AS, так как SELECT INTO не поддерживается в ECPG и PL/pgSQL, вследствие того, что они воспринимают

предложение INTO по-своему. К тому же, CREATE TABLE AS предоставляет больший набор возможностей, чем SELECT INTO.

Чтобы добавить колонку OID в таблицу, создаваемую командой SELECT INTO, необходимо установить конфигурационную переменную default_with_oids. С другой стороны, можно использовать CREATE TABLE AS с предложением WITH OIDS.

56.5.Примеры

Создание таблицы films_recent, содержащей только последние записи из таблицы films:

```
SELECT * INTO films_recent FROM films WHERE date_prod >=
'2002-01-01';
```

56.6.Совместимость

В стандарте SQL команда SELECT INTO применяется для передачи скалярных значений клиентской программе, но не для создания новой таблицы. Именно это применение имеет место в ECPG и в PL/pgSQL. В Синергия-БД команда SELECT INTO связана с созданием таблицы по историческим причинам. В новом коде для этих целей лучше использовать CREATE TABLE AS.

57. SET

SET – изменить параметр времени выполнения.

57.1.Синтаксис

```
SET [ SESSION | LOCAL ] параметр_конфигурации { TO | = }
{ значение | 'значение' | DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { часовой_пояс | LOCAL
| DEFAULT }
```

57.2.Описание

Команда SET изменяет конфигурационные параметры времени выполнения. С помощью SET можно «на лету» изменить многие из параметров. Но для изменения некоторых могут потребоваться права суперпользователя, а другие нельзя изменять

после запуска сервера или сеанса. SET влияет на значение параметра только в рамках текущего сеанса.

Если команда SET (или равнозначная SET SESSION) выполняется внутри транзакции, которая затем прерывается, эффект команды SET пропадает, когда транзакция откатывается. Если же окружающая транзакция фиксируется, этот эффект сохраняется до конца сеанса, если его не переопределит другая команда SET.

Действие SET LOCAL продолжается только до конца текущей транзакции, независимо от того, фиксируется она или нет. Особый случай представляет использование SET с последующей SET LOCAL в одной транзакции: значение, заданное SET LOCAL, будет сохраняться до конца транзакции, но после этого (если транзакция фиксируется) восстановится значение, заданное командой SET.

Действия SET или SET LOCAL также отменяются при откате к точке сохранения, установленной до выполнения этих команд.

Если SET LOCAL применяется в функции, параметр SET для которой устанавливает значение той же переменной, действие команды SET LOCAL прекращается при выходе из функции; то есть, в любом случае восстанавливается значение, существовавшее при вызове функции. Это позволяет использовать SET LOCAL для динамических и неоднократных изменений параметра в рамках функции, и при этом иметь удобную возможность использовать параметр SET для сохранения и восстановления значения, полученного извне. Однако обычная команда SET переопределяет любой параметр SET окружающей функции; её действие сохраняется, если не происходит откат транзакции.

57.3.Параметры

SESSION

Указывает, что команда действует в рамках текущего сеанса. Это поведение по умолчанию, если не указано ни SESSION, ни LOCAL.

LOCAL

Указывает, что команда действует только в рамках текущей транзакции. После COMMIT или ROLLBACK в силу вновь вступает значение, определенное на уровне

сеанса. При выполнении такой команды вне блока транзакции выдается предупреждение и больше ничего не происходит.

параметр_конфигурации

Имя устанавливаемого параметра времени выполнения.

значение

Новое значение параметра. Параметры могут задаваться в виде строковых констант, идентификаторов, чисел или списков перечисленных типов через запятую, в зависимости от конкретного параметра. Указание DEFAULT в данном контексте позволяет сбросить параметр к значению по умолчанию (то есть, к тому значению, которое он имел бы, если в текущем сеансе не выполнялись бы команды SET).

Помимо обычных конфигурационных параметров, есть ещё несколько параметров, которые можно изменить только командой SET или которые имеют особый синтаксис:

SCHEMA

SET SCHEMA 'значение' – альтернативное написание команды SET search_path TO значение. Такой синтаксис позволяет указать только одну схему.

NAMES

SET NAMES значение – альтернативное написание команды SET client_encoding TO значение.

SEED

Устанавливает внутреннее начальное значение для генератора случайных чисел (функции random). В качестве значения допускаются числа с плавающей точкой от -1 до 1, для внутреннего применения они затем умножаются на 2³¹-1.

Это начальное значение также можно установить, вызвав функцию setseed:

```
SELECT setseed(значение);
```

TIME ZONE

SET TIME ZONE значение – альтернативное написание команды SET timezone TO значение. Синтаксис SET TIME ZONE позволяет указывать часовой пояс в специальном формате.

57.4.Замечания

Также изменить значение параметра можно с помощью функции `set_config`. Кроме того, выполнив `UPDATE` в системном представлении `pg_settings`, можно произвести то же действие, что выполняет `SET`.

57.5.Примеры

Установка пути поиска схем:

```
SET search_path TO my_schema, public;
```

Установка традиционного стиля даты `POSTGRES` с форматом ввода «день, месяц, год»:

```
SET datestyle TO postgres, dmy;
```

Установка часового пояса для Беркли, штат Калифорния:

```
SET TIME ZONE 'PST8PDT';
```

Установка часового пояса Италии:

```
SET TIME ZONE 'Europe/Rome';
```

57.6.Совместимость

`SET TIME ZONE` расширяет синтаксис, определенный в стандарте SQL. Стандарт допускает только числовые смещения часовых поясов, тогда как Синергия-БД позволяет задавать часовой пояс более гибко. Все другие функции `SET` являются расширениями Синергия-БД.

58. SET CONSTRAINTS

`SET CONSTRAINTS` – установить время проверки ограничений для текущей транзакции.

58.1.Синтаксис

```
SET CONSTRAINTS { ALL | имя [, ...] } { DEFERRED |  
IMMEDIATE }
```

58.2.Описание

`SET CONSTRAINTS` определяет, когда будут проверяться ограничения в текущей транзакции. Ограничения `IMMEDIATE` проверяются в конце каждого

оператора, а ограничения DEFERRED откладываются до фиксации транзакции. Режим IMMEDIATE или DEFERRED задается для каждого ограничения независимо.

При создании ограничение получает одну из следующих характеристик: DEFERRABLE INITIALLY DEFERRED (откладываемое, изначально отложенное), DEFERRABLE INITIALLY IMMEDIATE (откладываемое, изначально немедленное) или NOT DEFERRABLE (неоткладываемое). Третий вариант всегда подразумевает IMMEDIATE и на него команда SET CONSTRAINTS не влияет. Первые два варианта запускаются в каждой транзакции в указанном режиме, но их поведение можно изменить в рамках транзакции командой SET CONSTRAINTS.

SET CONSTRAINTS со списком имен ограничений меняет режим только этих ограничений (все они должны быть откладываемыми). Имя любого ограничения можно дополнить схемой. Если имя схемы не указано, в поисках первого подходящего имени будет просматриваться текущий путь поиска схем. SET CONSTRAINTS ALL меняет режим всех откладываемых ограничений.

Когда SET CONSTRAINTS меняет режим ограничения с DEFERRED на IMMEDIATE, новый режим начинает действовать в обратную сторону: все изменения данных, ожидающие проверки в конце транзакции, вместо этого проверяются в момент выполнения команды SET CONSTRAINTS. Если какое-либо ограничение нарушается, при выполнении SET CONSTRAINTS происходит ошибка (и режим проверки не меняется). Таким образом, с помощью SET CONSTRAINTS можно принудительно проверить ограничения в определенном месте транзакции.

В настоящее время это распространяется только на ограничения UNIQUE, PRIMARY KEY, REFERENCES (внешний ключ) и EXCLUDE. Ограничения NOT NULL и CHECK всегда проверяются немедленно в момент добавления или изменения строки (не в конце оператора). Ограничения уникальности и ограничения-исключения, объявленные без указания DEFERRABLE, так же проверяются немедленно.

Срабатывание триггеров, объявленных как «триггеры ограничений» так же зависит от этой команды – они срабатывают в момент, когда должно проверяться соответствующее ограничение.

58.3.Замечания

Так как Синергия-БД не требует, чтобы имена ограничений были уникальны в схеме (достаточно уникальности в таблице), возможно, что для заданного имени найдется несколько соответствующих ограничений. В этом случае SET CONSTRAINTS подействует на все эти ограничения. Для имен без указания схемы, её действие будет распространяться только на ограничение(я), найденное в первой из схем; другие схемы просматриваться не будут.

Эта команда меняет поведение ограничений только в текущей транзакции. При выполнении этой команды вне блока транзакции выдается предупреждение и больше ничего не происходит.

58.4.Совместимость

Эта команда реализует поведение, описанное в стандарте SQL, с одним исключением – в Синергия-БД она не влияет на проверку ограничений NOT NULL и CHECK. Кроме того, Синергия-БД проверяет неоткладываемые ограничения уникальности немедленно, а не в конце оператора, как предлагает стандарт.

59. SET ROLE

SET ROLE – установить идентификатор текущего пользователя в рамках сеанса.

59.1.Синтаксис

```
SET [ SESSION | LOCAL ] ROLE имя_роли  
SET [ SESSION | LOCAL ] ROLE NONE  
RESET ROLE
```

59.2.Описание

Эта команда меняет идентификатор текущего пользователя в активном сеансе SQL на имя_роли. Имя роли может быть записано в виде идентификатора или строковой константы. После SET ROLE, права доступа для команд SQL проверяются так, как если бы сеанс изначально был установлен с этим именем роли.

Указывая определенное имя_роли, текущий пользователь должен являться

членом этой роли. Если пользователь сеанса является суперпользователем, он может выбрать любую роль.

Указания `SESSION` и `LOCAL` действуют на эту команду так же, как и на обычную команду `SET`.

Формы `NONE` и `RESET` сбрасывают идентификатор текущего пользователя, так что активным становится идентификатор пользователя сеанса. Эти формы могут выполняться любыми пользователями.

59.3.Замечания

С помощью этой команды можно как добавить права, так и ограничить их. Если роль пользователя сеанса имеет атрибут `INHERITS`, она автоматически получает права всех ролей, на которые может переключиться (с помощью `SET ROLE`); в этом случае `SET ROLE` убирает все права, назначенные непосредственно пользователю сеанса и другим ролям, в которые он включен, и оставляет только права, назначенные указанной роли. Если же роль пользователя сеанса имеет атрибут `NOINHERITS`, `SET ROLE` убирает права, назначенные непосредственно пользователю сеанса, и вместо них назначает права, которые имеет указанная роль.

В частности, когда суперпользователь переключается на роль не суперпользователя (используя `SET ROLE`), он теряет свои права суперпользователя.

`SET ROLE` оказывает действие, сравнимое с `SET SESSION AUTHORIZATION`, но проверка прав выполняется по-другому. Также `SET SESSION AUTHORIZATION` определяет, какие роли разрешены для последующей `SET ROLE`, тогда как при смене ролей командой `SET ROLE` набор ролей, допустимых для последующей команды `SET ROLE` не меняется.

`SET ROLE` не обрабатывает сеансовые переменные, указанные в свойствах роли `ALTER ROLE`; они устанавливаются только при подключении.

`SET ROLE` нельзя использовать внутри функции с характеристикой `SECURITY DEFINER`.

59.4.Примеры

```
SELECT SESSION_USER, CURRENT_USER;
```

```

  session_user | current_user
-----+-----
  peter       | peter

```

```
SET ROLE 'paul';
```

```
SELECT SESSION_USER, CURRENT_USER;
```

```

  session_user | current_user
-----+-----
  peter       | paul

```

59.5.Совместимость

Синергия-БД принимает идентификаторы («имя_роли»), тогда как стандарт SQL требует, чтобы имя роли записывалось в виде строковой константы. Стандарт не позволяет выполнять эту команду в транзакции; в Синергия-БД такого ограничения нет, так как для него нет причины. Указания SESSION и LOCAL относятся к расширениям Синергия-БД, так же, как и синтаксис RESET.

60. SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION – установить идентификатор пользователя сеанса и идентификатор текущего пользователя в рамках сеанса.

60.1.Синтаксис

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION
имя_пользователя
```

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
```

```
RESET SESSION AUTHORIZATION
```

60.2.Описание

Эта команда меняет идентификатор пользователя сеанса и идентификатор

текущего пользователя в рамках активного сеанса SQL на имя_пользователя. Имя пользователя может быть записано в виде идентификатора или строковой константы. С помощью этой команды, можно, например, временно переключиться на непривилегированного пользователя, сохранив возможность затем стать суперпользователем.

Идентификатором пользователя сеанса изначально принимается имя пользователя, введенное клиентом (возможно, прошедшее проверку подлинности). Идентификатор текущего пользователя обычно совпадает с идентификатором пользователя сеанса, но может временно меняться в функциях с характеристикой SECURITY DEFINER и подобными механизмами; также его можно изменить командой SET ROLE. Идентификатор текущего пользователя принимается во внимание при проверке разрешений.

Идентификатор пользователя сеанса можно изменить, только если начальный пользователь сеанса (аутентифицированный пользователь) имеет права суперпользователя. В противном случае эта команда разрешается, только если в ней указывается имя аутентифицированного пользователя.

Указания SESSION и LOCAL действуют на эту команду так же, как и на обычную команду SET.

Формы DEFAULT и RESET сбрасывают идентификаторы текущего пользователя и пользователя сеанса, так что текущим становится пользователь, изначально проходивший проверку подлинности. Эти формы могут выполняться любым пользователем.

60.3.Замечания

SET SESSION AUTHORIZATION нельзя использовать в функции с характеристикой SECURITY DEFINER.

60.4.Примеры

```
SELECT SESSION_USER, CURRENT_USER;
```

```
session_user | current_user
```

```
-----+-----
peter      | peter
```

```
SET SESSION AUTHORIZATION 'paul';
```

```
SELECT SESSION_USER, CURRENT_USER;
```

```
session_user | current_user
-----+-----
paul         | paul
```

60.5. Совместимость

Стандарт SQL позволяет вместо строковой константы имя_пользователя указывать некоторые другие выражения, но на практике это не очень полезно. Синергия-БД допускает синтаксис идентификаторов («имя_пользователя»), а стандарт SQL – нет. Стандарт не позволяет выполнять эту команду в транзакции; в Синергия-БД такого ограничения нет, так как для него нет причины. Указания SESSION и LOCAL относятся к расширениям Синергия-БД, так же, как и синтаксис RESET.

Набор прав, требуемых для выполнения этой команды, согласно стандарту, определяется реализацией.

61. SET TRANSACTION

SET TRANSACTION – установить характеристики текущей транзакции.

61.1. Синтаксис

```
SET TRANSACTION режим_транзакции [, ...]
SET TRANSACTION SNAPSHOT id_снимка
SET SESSION CHARACTERISTICS AS TRANSACTION
режим_транзакции [, ...]
```

Где возможные варианты режим_транзакции:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ |
READ COMMITTED | READ UNCOMMITTED }
READ WRITE | READ ONLY
```

[NOT] DEFERRABLE

61.2.Описание

Команда SET TRANSACTION устанавливает характеристики текущей транзакции. На последующие транзакции она не влияет. SET SESSION CHARACTERISTICS устанавливает характеристики транзакции по умолчанию для последующих транзакций в рамках сеанса. Заданные по умолчанию характеристики затем можно переопределить для отдельных транзакций командой SET TRANSACTION.

К характеристикам транзакции относится уровень изоляции транзакции, режим доступа транзакции (чтение/запись или только чтение) и допустимость её откладывания. В дополнение к ним можно выбрать снимок, но только для текущей транзакции, не для сеанса по умолчанию.

Уровень изоляции транзакции определяет, какие данные может видеть транзакция, когда параллельно с ней выполняются другие транзакции:

- READ COMMITTED

Оператор видит только те строки, которые были зафиксированы до начала его выполнения. Этот уровень устанавливается по умолчанию;

- REPEATABLE READ

Все операторы текущей транзакции видят только те строки, которые были зафиксированы перед первым запросом на выборку или изменение данных, выполненным в этой транзакции;

- SERIALIZABLE

Все операторы текущей транзакции видят только те строки, которые были зафиксированы перед первым запросом на выборку или изменение данных, выполненным в этой транзакции. Если наложение операций чтения и записи параллельных сериализуемых транзакций может привести к ситуации, невозможной при последовательном их выполнении (когда одна транзакция выполняется за другой), произойдет откат одной из транзакций с ошибкой `serialization_failure` (сбой сериализации).

В стандарте SQL определен ещё один уровень, READ UNCOMMITTED. В Синергия-БД уровень READ UNCOMMITTED обрабатывается как READ

COMMITTED.

Уровень изоляции транзакции нельзя изменить после выполнения первого запроса на выборку или изменение данных (SELECT, INSERT, DELETE, UPDATE, FETCH или COPY) в текущей транзакции.

Режим доступа транзакции определяет, будет ли транзакция только читать данные или будет и читать, и писать. По умолчанию подразумевается чтение/запись. В транзакции без записи запрещаются следующие команды SQL: INSERT, UPDATE, DELETE и COPY FROM, если только целевая таблица не временная; любые команды CREATE, ALTER и DROP, а также COMMENT, GRANT, REVOKE, TRUNCATE; кроме того, запрещаются EXPLAIN ANALYZE и EXECUTE, если команда, которую они должны выполнить, относится к вышеперечисленным. Это высокоуровневое определение режима только для чтения, которое в принципе не исключает запись на диск.

Свойство DEFERRABLE оказывает влияние, только если транзакция находится также в режимах SERIALIZABLE и READ ONLY. Когда для транзакции установлены все три этих свойства, транзакция может быть заблокирована при первой попытке получить свой снимок данных, после чего она сможет выполняться без дополнительных усилий, обычных для режима SERIALIZABLE, и без риска привести к сбою сериализации или пострадать от него. Этот режим подходит для длительных операций, например для построения отчетов или резервного копирования.

Команда SET TRANSACTION SNAPSHOT позволяет выполнить новую транзакцию со снимком данных, который имеет уже существующая. Эта ранее созданная транзакция должна экспортировать этот снимок с помощью функции pg_export_snapshot. Эта функция возвращает идентификатор снимка, который и нужно передать команде SET TRANSACTION SNAPSHOT в качестве идентификатора импортируемого снимка. В данной команде этот идентификатор должен записываться в виде строковой константы, например '000003A1-1'. SET TRANSACTION SNAPSHOT можно выполнить только в начале транзакции, до первого запроса на выборку или изменение данных (SELECT, INSERT, DELETE,

UPDATE, FETCH или COPY) в текущей транзакции. Более того, для транзакции уже должен быть установлен уровень изоляции `SERIALIZABLE` или `REPEATABLE READ` (в противном случае, снимок будет сразу же потерян, так как на уровне `READ COMMITTED` для каждой команды делается новый снимок). Если импортирующая транзакция работает на уровне изоляции `SERIALIZABLE`, то транзакция, экспортирующая снимок, также должна работать на этом уровне. Кроме того, транзакции в режиме чтение/запись не могут импортировать снимок из транзакции в режиме «только чтение».

61.3.Замечания

Если команде `SET TRANSACTION` не предшествует `START TRANSACTION` или `BEGIN`, она выдает предупреждение и больше ничего не делает.

Без `SET TRANSACTION` можно обойтись, задав требуемые режимы_транзакции в операторах `BEGIN` или `START TRANSACTION`. Но для `SET TRANSACTION SNAPSHOT` такой возможности не предусмотрено.

Режимы транзакции для сеанса по умолчанию можно также задать в конфигурационных переменных `default_transaction_isolation`, `default_transaction_read_only` и `default_transaction_deferrable`. На практике, `SET SESSION CHARACTERISTICS` – это просто более многословная альтернатива изменению этих переменных командой `SET`. Это значит, что значения этих переменных по умолчанию можно задать в файле конфигурации, с помощью команды `ALTER DATABASE` и т.д.

61.4.Примеры

Чтобы начать новую транзакцию со снимком данных, который получила уже существующая транзакция, его нужно сначала экспортировать из первой транзакции. При этом будет получен идентификатор снимка, например:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_export_snapshot();
pg_export_snapshot
-----
```

00000003-0000001B-1

(1 row)

Затем этот идентификатор нужно передать команде SET TRANSACTION SNAPSHOT в начале новой транзакции:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;  
SET TRANSACTION SNAPSHOT '00000003-0000001B-1';
```

61.5.Совместимость

Эти команды определены в стандарте SQL, за исключением режима транзакции DEFERRABLE и формы SET TRANSACTION SNAPSHOT, которые являются расширениями Синергия-БД.

В стандарте уровнем изоляции по умолчанию является SERIALIZABLE. В Синергия-БД уровнем по умолчанию обычно считается READ COMMITTED, но его можно изменить, как описано выше.

В стандарте SQL есть ещё одна характеристика транзакции, которую нельзя задать этими командами: размер диагностической области. Эта специфическая концепция встраиваемого SQL, поэтому в сервере Синергия-БД она не реализована.

Стандарт SQL требует, чтобы последовательные режимы_транзакций разделялись запятыми, но по историческим причинам Синергия-БД позволяет опустить запятые.

62. SHOW

SHOW – показать значение параметра времени выполнения

62.1.Синтаксис

SHOW имя

SHOW ALL

62.2.Описание

SHOW выводит текущие значения параметров времени выполнения. Эти переменные можно установить, воспользовавшись оператором SET, отредактировав файл конфигурации postgresql.conf, передав в переменной окружения PGOPTIONS (при использовании psql или приложения на базе libpq) либо в

параметрах командной строки при запуске сервера postgres.

62.3.Параметры

ИМЯ

Имя параметра времени выполнения. Доступные параметры описаны на странице справки SET. Кроме того, есть несколько параметров, которые можно просмотреть, но нельзя изменить:

SERVER_VERSION

Показывает номер версии сервера.

SERVER_ENCODING

Показывает кодировку набора символов на стороне сервера. В настоящее время этот параметр можно узнать, но нельзя изменить, так как кодировка определяется в момент создания базы данных.

LC_COLLATE

Показывает параметр локали базы данных, определяющий правило сортировки (порядок текстовых строк). В настоящее время этот параметр можно узнать, но нельзя изменить, так как он определяется в момент создания базы данных.

LC_CTYPE

Показывает параметр локали базы данных, определяющий классификацию символов. В настоящее время этот параметр можно узнать, но нельзя изменить, так как он определяется в момент создания базы данных.

IS_SUPERUSER

Возвращает true, если текущая роль обладает правами суперпользователя.

ALL

Показать значения всех конфигурационных параметров с описаниями.

62.4.Замечания

Ту же информацию выдает функция `current_setting`. Кроме того, эту информацию можно получить через системное представление `pg_settings`.

62.5.Примеры

Просмотр текущего значения параметра `DateStyle`:

```
SHOW DateStyle;
```

```
DateStyle
-----
ISO, MDY
(1 row)
```

Просмотр текущего значения параметра `geqo`:

```
SHOW geqo;
geqo
-----
on
(1 row)
```

62.6.Совместимость

Команда `SHOW` является расширением Синергия-БД.

63. START TRANSACTION

`START TRANSACTION` – начать блок транзакции.

63.1.Синтаксис

```
START TRANSACTION [ режим_транзакции [, ...] ]
```

Где возможные варианты `режим_транзакции`:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ |
READ COMMITTED | READ UNCOMMITTED }
READ WRITE | READ ONLY
[ NOT ] DEFERRABLE
```

63.2.Описание

Эта команда начинает новый блок транзакции. Если указан уровень изоляции, режим чтения/записи или допустимость откладывания транзакции, новая транзакция получит эти характеристики, как при выполнении команды `SET TRANSACTION`. Данная команда равнозначна команде `BEGIN`.

63.3.Совместимость

Согласно стандарту, выполнять `START TRANSACTION`, чтобы начать блок транзакции, необязательно: блок неявно начинает любая команда `SQL`. Поведение

Синергия-БД можно представить как неявное выполнение COMMIT после каждой команды, которой не предшествует START TRANSACTION (или BEGIN), и поэтому такое поведение часто называется «автофиксацией». Другие реляционные СУБД тоже могут предлагать автофиксацию как удобную возможность.

Значение DEFERRABLE параметра режим_транзакции является языковым расширением Синергия-БД.

Стандарт SQL требует, чтобы последовательные режимы_транзакций разделялись запятыми, но по историческим причинам Синергия-БД позволяет опустить запятые.

64. TRUNCATE

TRUNCATE – опустошить таблицу или набор таблиц.

64.1. Синтаксис

```
TRUNCATE [ TABLE ] [ ONLY ] имя [ * ] [, ... ]  
        [ RESTART IDENTITY | CONTINUE IDENTITY ] [ CASCADE |  
RESTRICT ]
```

64.2. Описание

Команда TRUNCATE быстро удаляет все строки из набора таблиц. Она действует так же, как безусловная команда DELETE для каждой таблицы, но гораздо быстрее, так как она фактически не сканирует таблицы. Более того, она немедленно высвобождает дисковое пространство, так что выполнять операцию VACUUM после нее не требуется. Наиболее полезна она для больших таблиц.

64.3. Параметры

ИМЯ

Имя (возможно, дополненное схемой) таблицы, подлежащей опустошению. Если перед именем таблицы указано ONLY, очищается только заданная таблица. Без ONLY очищается и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно также добавить необязательное указание *, чтобы явно обозначить, что блокировка затрагивает и все дочерние таблицы.

RESTART IDENTITY

Автоматически перезапускать последовательности, связанные с колонками опустошаемой таблицы.

CONTINUE IDENTITY

Не изменять значения последовательностей. Это поведение по умолчанию.

CASCADE

Автоматически опустошать все таблицы, ссылающиеся по внешнему ключу на заданные таблицы, или на таблицы, затронутые в результате действия CASCADE.

RESTRICT

Отказать в опустошении любых таблиц, на которые по внешнему ключу ссылаются другие таблицы, не перечисленные в этой команде. Это поведение по умолчанию.

64.4.Замечания

Чтобы опустошить таблицу, необходимо иметь право TRUNCATE для этой таблицы.

Команда TRUNCATE запрашивает блокировку ACCESS EXCLUSIVE для каждой таблицы, которую она обрабатывает. Когда указано RESTART IDENTITY, все последовательности, которые должны быть перезапущены, так же блокируются исключительно. В случаях, когда требуется обеспечить параллельный доступ к таблице, следует использовать DELETE.

TRUNCATE нельзя использовать с таблицей, на которую по внешнему ключу ссылаются другие таблицы, если только и эти таблицы не опустошаются этой же командой. Проверка допустимости очистки в таких случаях потребовала бы сканирования таблицы, а главная идея данной команды в том, чтобы не делать этого. Для автоматической обработки всех зависимых таблиц можно использовать указание CASCADE – но будьте очень осторожны с ним, иначе вы можете потерять данные, которые не собирались удалять!

При выполнении TRUNCATE не срабатывают никакие триггеры ON DELETE, которые могут быть настроены для таблиц. Однако при этом срабатывают триггеры ON TRUNCATE. Если триггеры ON TRUNCATE определены для любых из этих

таблиц, то все триггеры BEFORE TRUNCATE срабатывают до того, как происходит опустошение, а все триггеры AFTER TRUNCATE срабатывают после того, как завершается опустошение последней таблицы и все последовательности сбрасываются. Триггеры срабатывают по порядку обработки таблиц (сначала для таблиц, перечисленных в команде, затем для тех, что затрагиваются каскадно).

Команда TRUNCATE не является безопасной с точки зрения многоверсионности. После опустошения таблицы, она будет выглядеть пустой для одновременно выполняемых транзакций, которые используют снимок, сделанный до того, как произошло опустошение.

TRUNCATE является надежной транзакционной операцией в отношении данных в таблицах: опустошение будет безопасно отменено, если окружающая транзакция не будет зафиксирована.

С указанием RESTART IDENTITY подразумеваемые операции ALTER SEQUENCE RESTART также выполняются транзакционно; то есть, они будут отменены, если окружающая транзакция не будет зафиксирована. Это отличается от обычного поведения ALTER SEQUENCE RESTART. Учтите, что если до того, как транзакция отменится, будут выполнены какие-либо дополнительные операции с последовательностями, эффект этих операций также будет отменен, но не их влияние на значение `currval()`; то есть после транзакции `currval()` продолжит возвращать последнее значение последовательности, полученное внутри прерванной транзакции, хотя сама последовательность уже может быть несогласованной с ним. Подобным образом обычно ведет себя `currval()` после сбоя транзакции.

TRUNCATE в настоящее время не поддерживается для сторонних таблиц. Из этого следует, что если у целевой таблицы есть дочерние таблицы, являющиеся сторонними, команда не будет выполнена.

64.5.Примеры

Опустошение таблиц `bigtable` и `fattable`:

```
TRUNCATE bigtable, fattable;
```

Та же операция и сброс всех связанных генераторов последовательностей:

```
TRUNCATE bigtable, fattable RESTART IDENTITY;
```

Опустошение таблицы othertable и каскадная обработка всех таблиц, ссылающихся на othertable по ограничениям внешнего ключа:

```
TRUNCATE othertable CASCADE;
```

64.6.Совместимость

Стандарт SQL:2008 включает команду TRUNCATE с синтаксисом TRUNCATE TABLE имя_таблицы. Предложения CONTINUE IDENTITY/RESTART IDENTITY также описаны в стандарте, но с небольшими отличиями, хотя их назначение похоже. Поведение этой команды при параллельных операциях, согласно стандарту, отчасти определяются реализацией, так что приведенные выше замечания при необходимости следует учитывать и сопоставлять с другими реализациями.

65. UNLISTEN

UNLISTEN – прекратить ожидание уведомления.

65.1.Синтаксис

```
UNLISTEN { канал | * }
```

65.2.Описание

UNLISTEN применяется для отмены существующей подписки на получение событий NOTIFY. UNLISTEN отменяет существующую подписку в текущем сеансе Синергия-БД на канал уведомлений с именем канал. Специальный знак * отменяет все подписки в текущем сеансе.

В описании NOTIFY использование LISTEN и NOTIFY рассматривается более подробно.

65.3.Параметры

канал

Имя канала уведомлений (любой идентификатор).

*

Отменяются все текущие подписки на уведомления для активного сеанса.

65.4.Замечания

Вы можете также попытаться отменить подписку на канал, на который не подписаны; предупреждений или ошибки при этом не будет.

UNLISTEN * автоматически выполняется в конце каждого сеанса.

Транзакция, выполнившая UNLISTEN, не может быть подготовлена для двухфазной фиксации.

65.5.Примеры

Подписка на получение события:

```
LISTEN virtual;
```

```
NOTIFY virtual;
```

```
Asynchronous notification «virtual» received from server  
process with PID 8448.
```

Сразу после выполнения UNLISTEN последующие сообщения NOTIFY игнорируются:

```
UNLISTEN virtual;
```

```
NOTIFY virtual;
```

```
-- событие NOTIFY не поступает
```

65.6.Совместимость

Команда UNLISTEN отсутствует в стандарте SQL.

66. UPDATE

UPDATE – изменить строки таблицы.

66.1.Синтаксис

```
[ WITH [ RECURSIVE ] запрос_WITH [, ...] ]  
UPDATE [ ONLY ] имя_таблицы [ * ] [ [ AS ] псевдоним ]  
    SET { имя_столбца = { выражение | DEFAULT } |  
        ( имя_столбца [, ...] ) = [ ROW ]  
    ( { выражение | DEFAULT } [, ...] ) |  
        ( имя_столбца [, ...] ) = ( вложенный_SELECT )  
    } [, ...]  
[ FROM список_FROM ]
```

```
[ WHERE условие | WHERE CURRENT OF имя_курсора ]
[ RETURNING * | выражение_результата [ [ AS ]
имя_результата ] [, ...] ]
```

66.2.Описание

UPDATE изменяет значения указанных колонок во всех строках, удовлетворяющих условию. В предложении SET должны указываться только те колонки, которые будут изменены; колонки, не изменяемые явно, сохраняют свои предыдущие значения.

Изменить строки в таблице, используя информацию из других таблиц в базе данных, можно двумя способами: применяя вложенные запросы или указав дополнительные таблицы в предложении FROM. Выбор предпочитаемого варианта зависит от конкретных обстоятельств.

Предложение RETURNING указывает, что команда UPDATE должна вычислить и вернуть значения для каждой фактически измененной строки. Вычислить в нем можно любое выражение с колонками целевой таблицы и/или колонками других таблиц, упомянутых во FROM. При этом в выражении будут использоваться новые (измененные) значения колонок таблицы. Список RETURNING имеет тот же синтаксис, что и список результатов SELECT.

Для выполнения этой команды необходимо иметь право UPDATE для таблицы, или как минимум для колонок, перечисленных в списке изменяемых. Также необходимо иметь право SELECT для всех колонок, значения которых считываются в выражениях или условии.

66.3.Параметры

запрос_WITH

Предложение WITH позволяет задать один или несколько подзапросов, на которые затем можно ссылаться по имени в запросе UPDATE.

имя_таблицы

Имя (возможно, дополненное схемой) таблицы, строки которой будут изменены. Если перед именем таблицы добавлено ONLY, соответствующие строки изменяются только в указанной таблице. Без ONLY строки будут также изменены во

всех таблицах, унаследованных от указанной. При желании, после имени таблицы можно указать *, чтобы явно обозначить, что операция затрагивает все дочерние таблицы.

псевдоним

Альтернативное имя целевой таблицы. Когда указывается это имя, оно полностью скрывает фактическое имя таблицы. Например, в запросе `UPDATE foo AS f` дополнительные компоненты оператора `UPDATE` должны обращаться к целевой таблице по имени `f`, а не `foo`.

имя_колонки

Имя колонки в таблице `имя_таблицы`. Имя колонки при необходимости может быть дополнено именем вложенного поля или индексом массива. Имя таблицы добавлять к имени целевой колонки не нужно – например, запись `UPDATE tab SET tab.col = 1` ошибочна.

выражение

Выражение, результат которого присваивается колонке. В этом выражении можно использовать предыдущие значения этой и других колонок таблицы.

DEFAULT

Присвоить колонке значение по умолчанию (это может быть `NULL`, если для колонки не определено некоторое выражение по умолчанию).

вложенный_SELECT

Подзапрос `SELECT`, выдающий столько выходных столбцов, сколько перечислено в предшествующем ему списке столбцов в скобках. При выполнении этого подзапроса должна быть получена максимум одна строка. Если он выдаёт одну строку, значения столбцов в нём присваиваются целевым столбцам; если же он не возвращает строку, целевым столбцам присваивается `NULL`. Этот подзапрос может обращаться к предыдущим значениям текущей изменяемой строки в таблице.

список_FROM

Список табличных выражений, позволяющий использовать в условии `WHERE` и выражениях присваивания колонки из других таблиц. Этот список подобен тому, что задается в предложении Предложение `FROM` оператора `SELECT`. Заметьте, что целевую таблицу нужно добавлять в `список_FROM` только при формировании

замкнутого соединения (в этом случае она должна фигурировать в списке `_FROM` под псевдонимом).

условие

Выражение, возвращающее значение типа `boolean`. Изменены будут только те строки, для которых это выражение возвращает `true`.

имя_курсора

Имя курсора, который будет использоваться в условии `WHERE CURRENT OF`. С таким условием будет изменена строка, выбранная из этого курсора последней. Курсор должен образовываться запросом, не применяющим группировку, к целевой таблице команды `UPDATE`. Заметьте, что `WHERE CURRENT OF` нельзя задать вместе с булевым условием. За дополнительными сведениями об использовании курсоров с `WHERE CURRENT OF` обратитесь к `DECLARE`.

выражение_результата

Выражение, которое будет вычисляться и возвращаться командой `UPDATE` после изменения каждой строки. В этом выражении можно использовать имена любых колонок таблицы `имя_таблицы` или таблиц, перечисленных в списке `FROM`. Чтобы получить все колонки, достаточно написать `*`.

имя_результата

Имя, назначаемое возвращаемой колонке.

66.4. Выводимая информация

В случае успешного завершения, `UPDATE` возвращает метку команды в виде

`UPDATE число`

Здесь `число` обозначает количество измененных строк, включая те подлежащие изменению строки, значения в которых не были изменены. Заметьте, что это число может быть меньше количества строк, удовлетворяющих условию, когда изменения отменяются триггером `BEFORE UPDATE`. Если число равно 0, данный запрос не изменил ни одной строки (это не считается ошибкой).

Если команда `UPDATE` содержит предложение `RETURNING`, её результат будет похож на результат оператора `SELECT` (с теми же колонками и значениями, что содержатся в списке `RETURNING`), полученный для строк, измененных этой

командой.

66.5.Замечания

Когда присутствует предложение FROM, целевая таблица по сути соединяется с таблицами, перечисленными в списке FROM, и каждая выходная строка соединения представляет операцию изменения для целевой таблицы. Применяя предложение FROM, необходимо обеспечить, чтобы соединение выдавало максимум одну выходную строку для каждой строки, которую нужно изменить. Другими словами, целевая строка не должна соединяться с более чем одной строкой из других таблиц. Если это условие нарушается, только одна из строк соединения будет использоваться для изменения целевой строки, но какая именно, предсказать нельзя.

Из-за этой неопределённости надёжнее ссылаться на другие таблицы только в подзапросах, хотя такие запросы часто хуже читаются и работают медленнее, чем соединение.

В секционированной таблице строка при изменении может перестать удовлетворять ограничению содержащей её секции. При этом если есть другая секция в дереве секционирования, ограничению которой эта строка удовлетворяет, то она переносится в данную секцию. Если такой секции нет, происходит ошибка. За кулисами перемещение строки выполняется посредством операций DELETE и INSERT.

Существует возможность того, что при выполнении другой параллельной операции UPDATE или DELETE с перемещаемой строкой возникнет ошибка сериализации. Например, предположим, что в сеансе 1 выполняется UPDATE для ключа секционирования, а тем временем в параллельном сеансе 2, в котором эта строка видима, выполняется операция UPDATE или DELETE с этой строкой. В этом случае UPDATE/DELETE в сессии 2 заметит перемещение строки и выдаст ошибку сериализации (которая всегда представляется кодом SQLSTATE '40001'). Получив такую ошибку, приложения могут попытаться повторить транзакцию. В обычном случае, когда таблица не секционирована или строка не перемещается, в сеансе 2 видна изменённая строка, и операция UPDATE/DELETE выполняется с новой

версией строки.

Заметьте, что строки могут перемещаться из локальных секций в секцию в сторонней таблице (если обёртка сторонних данных поддерживает перенаправление кортежей), но не из секции в сторонней таблице в другую секцию.

66.6.Примеры

Изменение слова Drama на Dramatic в колонке kind таблицы films:

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

Изменение значений температуры и сброс уровня осадков к значению по умолчанию в одной строке таблицы weather:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi =  
temp_lo+15, prcp = DEFAULT  
WHERE city = 'San Francisco' AND date = '2003-07-03';
```

Выполнение той же операции с получением измененных записей:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi =  
temp_lo+15, prcp = DEFAULT  
WHERE city = 'San Francisco' AND date = '2003-07-03'  
RETURNING temp_lo, temp_hi, prcp;
```

Такое же изменение с применением альтернативного синтаксиса со списком колонок:

```
UPDATE weather SET (temp_lo, temp_hi, prcp) =  
(temp_lo+1, temp_lo+15, DEFAULT)  
WHERE city = 'San Francisco' AND date = '2003-07-03';
```

Увеличение счетчика продаж для продавца, занимающегося компанией Acme Corporation, с применением предложения FROM:

```
UPDATE employees SET sales_count = sales_count + 1 FROM  
accounts  
WHERE accounts.name = 'Acme Corporation'  
AND employees.id = accounts.sales_person;
```

Выполнение той же операции, с вложенным запросом в предложении WHERE:

```
UPDATE employees SET sales_count = sales_count + 1 WHERE  
id =  
(SELECT sales_person FROM accounts WHERE name = 'Acme  
Corporation');
```

Изменение имени контакта в таблице счетов (это должно быть имя назначенного менеджера по продажам):

```
UPDATE accounts SET (contact_first_name,  
contact_last_name) =  
    (SELECT first_name, last_name FROM salesmen  
    WHERE salesmen.id = accounts.sales_id);
```

Подобный результат можно получить, применив соединение:

```
UPDATE accounts SET contact_first_name = first_name,  
                    contact_last_name = last_name  
    FROM salesmen WHERE salesmen.id = accounts.sales_id;
```

Однако, если `salesmen.id` – не уникальный ключ, второй запрос может давать непредсказуемые результаты, тогда как первый запрос гарантированно выдаст ошибку, если найдётся несколько записей с одним `id`. Кроме того, если соответствующая запись `accounts.sales_id` не найдётся, первый запрос запишет в поля имени `NULL`, а второй вовсе не изменит строку.

Обновление статистики в сводной таблице в соответствии с текущими данными:

```
UPDATE summary s SET (sum_x, sum_y, avg_x, avg_y) =  
    (SELECT sum(x), sum(y), avg(x), avg(y) FROM data d  
    WHERE d.group_id = s.group_id);
```

Попытка добавить новый продукт вместе с количеством. Если такая запись уже существует, вместо этого увеличить количество данного продукта в существующей записи. Чтобы реализовать этот подход, не откатывая всю транзакцию, можно использовать точки сохранения:

```
BEGIN;  
-- другие операции  
SAVEPOINT sp1;  
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');  
-- Предполагая, что здесь возникает ошибка из-за  
-- нарушения уникальности ключа,  
-- мы выполняем следующие команды:  
ROLLBACK TO sp1;  
UPDATE wines SET stock = stock + 24 WHERE winename =
```

```
'Chateau Lafite 2003';
```

```
-- Продолжение других операций и в завершение...
```

```
COMMIT;
```

Изменение колонки `kind` таблицы `films` в строке, на которой в данный момент находится курсор `c_films`:

```
UPDATE films SET kind = 'Dramatic' WHERE CURRENT OF  
c_films;
```

66.7. Совместимость

Эта команда соответствует стандарту SQL, за исключением предложений `FROM` и `RETURNING`, которые являются расширениями Синергия-БД, как и возможность применять `WITH` с `UPDATE`.

В некоторых других СУБД также поддерживается дополнительное предложение `FROM`, но предполагается, что целевая таблица должна ещё раз упоминаться в этом предложении. Синергия-БД воспринимает предложение `FROM` не так, поэтому будьте внимательны, портируя приложения, которые используют это расширение языка.

Согласно стандарту, исходным значением для вложенного списка имён столбцов в скобках может быть любое выражение, выдающее строку с нужным количеством столбцов. Синергия-БД принимает в качестве этого значения только конструктор строки или вложенный `SELECT`. Изменяемое значение отдельного столбца можно обозначать словом `DEFAULT` в конструкторе строки, но не внутри вложенного `SELECT`.

67. VACUUM

`VACUUM` – провести сборку мусора и, возможно, проанализировать базу данных.

67.1. Синтаксис

```
VACUUM [ ( параметр [, ...] ) ] [ таблица_и_столбцы  
[, ...] ]
```

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ ANALYZE ]  
[ таблица_и_столбцы [, ...] ]
```

Здесь допускается параметр:

FULL
FREEZE
VERBOSE
ANALYZE
DISABLE_PAGE_SKIPPING

и таблица_и_столбцы:

имя_таблицы [(имя_столбца [, ...])]

67.2.Описание

VACUUM высвобождает пространство, занимаемое «мертвыми» кортежами. При обычных операциях Синергия-БД кортежи, удаленные или устаревшие в результате обновления, физически не удаляются из таблицы; они сохраняются в ней, пока не будет выполнена команда VACUUM. Таким образом, периодически необходимо выполнять VACUUM, особенно для часто изменяемых таблиц.

Без параметра команда VACUUM обрабатывает все таблицы в текущей базе данных, которые может очистить текущий пользователь. Если в параметре передается имя таблицы, VACUUM обрабатывает только эту таблицу.

VACUUM ANALYZE выполняет очистку (VACUUM), а затем анализ (ANALYZE) всех указанных таблиц. Это удобная комбинация для регулярного обслуживания БД. За дополнительной информацией об анализе обратитесь к описанию ANALYZE.

Простая команда VACUUM (без FULL) только высвобождает пространство и делает его доступным для повторного использования. Эта форма команды может работать параллельно с обычными операциями чтения и записи таблицы, так она не требует исключительной блокировки. Однако освобожденное место не возвращается операционной системе (в большинстве случаев); оно просто остается доступным для размещения данных этой же таблицы. VACUUM FULL переписывает все содержимое таблицы в новый файл на диске, не содержащий ничего лишнего, что позволяет вернуть неиспользованное пространство операционной системе. Эта форма работает намного медленнее и запрашивает исключительную блокировку для каждой обрабатываемой таблицы.

Когда список параметров заключается в скобки, параметры могут быть записаны в любом порядке. Без скобок параметры должны указываться именно в том порядке, который показан выше.

67.3.Параметры

FULL

Выбирает режим «полной» очистки, который может освободить больше пространства, но выполняется гораздо дольше и запрашивает исключительную блокировку таблицы. Этот режим также требует дополнительное место на диске, так как он записывает новую копию таблицы и не освобождает старую до завершения операции. Обычно это следует использовать, только когда требуется высвободить значительный объем пространства, выделенного таблице.

FREEZE

Выбирает агрессивную «заморозку» кортежей. Добавление указания FREEZE равносильно выполнению команды VACUUM с параметрами vacuum_freeze_min_age и vacuum_freeze_table_age, равными нулю. Агрессивная заморозка всегда выполняется при перезаписи таблицы, поэтому в режиме FULL это указание избыточно.

VERBOSE

Выводит подробный отчет об очистке для каждой таблицы.

ANALYZE

Обновляет статистику, которую использует планировщик для выбора наиболее эффективного способа выполнения запроса.

DISABLE_PAGE_SKIPPING

Обычно VACUUM пропускает страницы, учитывая карту видимости. Страницы, на которых, судя по карте, все кортежи заморожены, можно пропускать всегда, а страницы, в которых все кортежи видны всем транзакциям, могут обрабатываться только при агрессивной очистке. Более того, за исключением агрессивной очистки, некоторые страницы можно пропускать, чтобы не ждать, пока другие сеансы закончат их использовать. Этот параметр отключает пропуск страниц и предназначен для использования только когда целостность карты видимости

вызывает подозрения, что возможно при аппаратных или программных сбоях, приводящих к разрушению БД.

имя_таблицы

Имя (возможно, дополненное схемой) определённой таблицы или материализованного представления, подлежащего очистке. Если указанная таблица является секционированной, очистке подвергаются все её конечные секции.

имя_столбца

Имя столбца, подлежащего анализу. По умолчанию анализируются все столбцы. Если указывается список столбцов, также должно присутствовать указание ANALYZE.

67.4.Выводимая информация

С указанием VERBOSE команда VACUUM выдает сообщения о процессе очистки, отмечая текущую обрабатываемую таблицу. Также она выводит различные статистические сведения о таблицах.

67.5.Замечания

Чтобы очистить таблицу, обычно нужно быть владельцем этой таблицы или суперпользователем. Однако владельцам баз данных также разрешено сжимать все таблицы в своих базах, за исключением общих каталогов. Ограничение в отношении общих каталогов означает, что настоящую глобальную команду VACUUM может выполнить только суперпользователь. VACUUM при обработке пропускает все таблицы, на очистку которых текущий пользователь не имеет прав.

VACUUM нельзя выполнять внутри блока транзакции.

Для таблиц с индексами GIN, VACUUM (в любой форме) также завершает все ожидающие операции добавления в индекс, перемещая записи индекса из очереди в соответствующие места в основной структуре индекса GIN.

Мы рекомендуем очищать активные производственные базы данных достаточно часто (как минимум, каждую ночь), чтобы избавляться от «мертвых» строк. После добавления или удаления большого числа строк может быть хорошей идеей выполнить команду VACUUM ANALYZE для каждой затронутой таблицы. При

этом результаты всех последних изменений будут отражены в системных каталогах, что позволит планировщику запросов Синергия-БД принимать более эффективные решения при планировании.

Режим FULL не рекомендуется для обычного применения, но в некоторых случаях он бывает полезен. Например, когда были удалены или изменены почти все строки таблицы, может возникнуть желание физически сжать ее, чтобы освободить место на диске и ускорить сканирование этой таблицы. Чаще всего VACUUM FULL сжимает таблицу более эффективно, чем обычный VACUUM.

VACUUM создает значительную нагрузку на подсистему ввода/вывода, что может отрицательно сказаться на производительности других активных сеансов. Поэтому иногда полезно использовать возможность задержки очистки с учетом её стоимости.

Синергия-БД включает средство «автоочистки», которое позволяет автоматизировать регулярную очистку.

67.6.Примеры

Очистка одной таблицы onek, проведение её анализа для оптимизатора и печать подробного отчёта о действиях операции очистки:

```
VACUUM (VERBOSE, ANALYZE) onek;
```

67.7.Совместимость

Оператор VACUUM отсутствует в стандарте SQL.

68. VALUES

VALUES – вычислить набор строк.

68.1.Синтаксис

```
VALUES ( выражение [, ...] ) [, ...]
```

```
    [ ORDER BY выражение_сортировки [ ASC | DESC | USING  
оператор ] [, ...] ]
```

```
    [ LIMIT { число | ALL } ]
```

```
    [ OFFSET начало [ ROW | ROWS ] ]
```

```
    [ FETCH { FIRST | NEXT } [ число ] { ROW | ROWS }
```


ONLY]

68.2.Описание

VALUES вычисляет значение строки или множество значений строк, заданное выражениями. Чаще всего эта команда используется для формирования «таблицы констант» в большой команде, но её можно использовать и отдельно.

Когда указывается больше, чем одна строка, все строки должны иметь одинаковое количество элементов. Типы данных результирующих колонок таблицы определяются в результате совмещения явных и неявных типов выражений, заданных для этих колонок, по тем же правилам, что и в UNION.

В составе других команд синтаксис допускает использование VALUES везде, где допускается SELECT. Так как грамматически она воспринимается как SELECT, с командой VALUES можно использовать предложения ORDER BY, LIMIT (или равнозначное FETCH FIRST) и OFFSET.

68.3.Параметры

выражение

Константа или выражение, которое вычисляется и вставляется в указанное место результирующей таблицы (множества строк). В списке VALUES, находящемся на верхнем уровне оператора INSERT, выражение может быть заменено словом DEFAULT, указывающим, что в целевую колонку должно быть вставлено значение этой колонки по умолчанию. Когда VALUES употребляется в других контекстах, указание DEFAULT использовать нельзя.

выражение_сортировки

Выражение или целочисленная константа, указывающая, как должны сортироваться строки результата. Это выражение может обращаться к колонкам результата VALUES по именам column1, column2 и т.д.

оператор

Оператор сортировки.

число

Максимальное число строк, которое должно быть возвращено.

начало

Число строк, которые должны быть пропущены, прежде чем начнется выдача строк.

68.4.Замечания

Следует избегать составления списков VALUES с очень большим количеством строк, так как при этом можно столкнуться с нехваткой памяти или снижением производительности. Применение VALUES в команде INSERT – особый случай (так как ожидаемые типы колонок становятся известны из целевой таблицы команды INSERT и их не надо вычислять, сканируя весь список VALUES), так что в этом контексте можно работать с гораздо более объемными списками, чем в других.

68.5.Примеры

Простейшая команда VALUES:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

Эта команда выдаст таблицу из двух колонок и трех строк. По сути она равнозначна запросу:

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

Более типично использование VALUES в составе большей команды SQL. Чаще всего она применяется в INSERT:

```
INSERT INTO films (code, title, did, date_prod, kind)
VALUES ('T_601', 'Yojimbo', 106, '1961-06-16',
'Drama');
```

В контексте INSERT список VALUES может содержать слово DEFAULT, указывающее, что в данном месте вместо некоторого значения должно использоваться значение колонки по умолчанию:

```
INSERT INTO films VALUES
('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82
minutes'),
```

```
('T_601', 'Yojimbo', 106, DEFAULT, 'Drama',  
DEFAULT);
```

VALUES также может применяться там, где можно написать вложенный SELECT, например в предложении FROM:

```
SELECT f.*  
FROM films f, (VALUES('MGM', 'Horror'), ('UA', 'Sci-  
Fi')) AS t (studio, kind)  
WHERE f.studio = t.studio AND f.kind = t.kind;  
  
UPDATE employees SET salary = salary * v.increase  
FROM (VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v  
(depno, target, increase)  
WHERE employees.depno = v.depno AND employees.sales >=  
v.target;
```

Заметьте, что когда VALUES используется в предложении FROM, предложение AS становится обязательным, так же, как и для SELECT. При этом не требуется указывать в AS имена всех колонок, но это рекомендуется делать. По умолчанию Синергия-БД дает колонкам VALUES имена column1, column2 и т.д., но в других СУБД имена могут быть другими.

Когда VALUES используется в команде INSERT, значения автоматически приводятся к типу данных соответствующей целевой колонки. Когда она используется в других контекстах, может потребоваться указать нужный тип данных. Если все записи представлены строковыми константами в кавычках, достаточно привести к нужному типу значения в первой строке, чтобы задать тип для всех:

```
SELECT * FROM machines  
WHERE ip_address IN (VALUES('192.168.0.1'::inet),  
('192.168.0.10'), ('192.168.1.43'));
```

Примечание. Для простых проверок тестов на включение (IN) лучше полагаться на форму IN со списком скаляров, списком скаляров, чем записывать запрос VALUES, как показано выше. Список скаляров проще записать и часто он обрабатывается он зачастую более эффективно.

68.6.Совместимость

VALUES соответствует стандарту SQL. Указания LIMIT и OFFSET являются расширениями Синергия-БД.

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

SQL	-	Structured Query Language (язык структурированных запросов)
WAL	-	Write-Ahead Log (журнал упреждающей записи)
БД	-	база данных
КСЗ	-	комплекс средств защиты
ОЗУ	-	оперативное запоминающее устройство
ОС	-	операционная система
ПРД	-	правила разграничения доступа
СУБД	-	система управления базами данных

[illegible]

Федеральное государственное унитарное предприятие
Российский федеральный ядерный центр
Всероссийский научно-исследовательский институт экспериментальной физики

УТВЕРЖДЕН
07623615.00082-02 34 01-ЛУ

КОМПЛЕКС ПРОГРАММ В ЗАЩИЩЕННОМ ИСПОЛНЕНИИ
«СИСТЕМА ПОЛНОГО ЖИЗНЕННОГО ЦИКЛА ИЗДЕЛИЙ
«ЦИФРОВОЕ ПРЕДПРИЯТИЕ»

**Программный модуль
«Система управления базами данных «Синергия-БД»**

Руководство оператора

Часть 5

07623615.00082-02 34 01-5

Листов 196

Име. № подл.	Подп. и дата	Взам. име. №	Име. № дубл.	Подп. и дата

СОДЕРЖАНИЕ

1. clusterdb	3
2. createdb	6
3. createuser	10
4. dropdb	15
5. dropuser	17
6. ecpg	20
7. pg_basebackup	23
8. pgbench	34
9. pg_config	58
10. pg_dump	62
11. pg_dumpall	80
12. pg_isready	89
13. pg_receivewal	92
14. pg_recvlogical	97
15. pg_restore	102
16. pg-wrapper	116
17. psql	118
18. reindexdb	179
19. vacuumdb	183
Перечень сокращений	188
Приложение 1 (справочное) Коды ошибок СУБД «Синергия-БД»	189

Документ описывает клиентские приложения и утилиты СУБД «Синергия-БД» (далее Синергия-БД). Некоторые из описанных приложений требуют особые привилегии. Основной отличительной особенностью этих приложений является возможность исполнения на любом компьютере, независимо от расположения сервера баз данных.

Имя пользователя и базы данных передаются из командной строки на сервер без изменения регистра – все пробельные и специальные символы необходимо экранировать с помощью кавычек. Имена таблиц и другие идентификаторы передаются регистр-независимо, за исключением отдельно описанных ситуаций, где может требоваться экранирование.

1. CLUSTERDB

`clusterdb` – кластеризовать базу данных СУБД «Синергия-БД».

1.1. Синтаксис

```
clusterdb [ опции подключения ... ] [ --verbose | -v ]  
[ --table | -t таблица ] ... [ dbname ]  
clusterdb [ опции подключения ... ] [ --verbose | -v ] --  
all | -a
```

1.2. Описание

`clusterdb` это приложение для повторной кластеризации таблиц базы данных Синергия-БД. Утилита находит ранее кластеризованные таблицы и проводит операцию на основании последнего использованного индекса. Затрагиваются лишь ранее кластеризованные таблицы.

`clusterdb` это обертка для SQL-команды `CLUSTER`. Оба пути кластеризации идентичны в результате.

1.3. Опции

`clusterdb` принимает аргументы командной строки, описанные ниже.

`-a`

--all

Кластеризовать все базы данных.

[-d] dbname

[--dbname=] dbname

Указывает имя базы данных для кластеризации. Если оно не указано, а опция -a (или --all) не используется, то имя базы берется из переменной окружения PGDATABASE. Если переменная окружения не установлена, то берется имя пользователя из параметров подключения в качестве имени целевой базы.

-e

--echo

Вывести команды к серверу, генерируемые при выполнении clusterdb.

-q

--quiet

Подавлять вывод сообщений о прогрессе выполнения.

-t таблица

--table=таблица

Кластеризовать таблицу. Возможно множественное использование опции -t.

-v

--verbose

Вывести подробную информацию во время процесса.

-V

--version

Вывести версию clusterdb и прервать дальнейшее выполнение.

-?

--help

Вывести справку по аргументам команды clusterdb.

clusterdb также принимает из командной строки параметры подключения:

-h host

--host=host

Указывает имя компьютера, на котором запущен сервер. Если значение начинается с наклонной черты, то оно интерпретируется как имя директории, где хранится файл Unix-сокета.

```
-p port  
--port=port
```

Указывает TCP-порт или расширение файла Unix-сокета, на котором сервер слушает подключения.

```
-U username  
--username=username
```

Имя пользователя, под которым производится подключение.

```
-w  
--no-password
```

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Опция может быть полезна в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

```
-W  
--password
```

Принудить `clusterdb` к запросу пароля перед подключением к серверу баз данных.

Опция несущественна, т.к. при включенной на сервере аутентификации по паролю, `clusterdb` запросит его автоматически. Однако `clusterdb` сделает лишнее подключение для определения необходимости ввода пароля. Поэтому во избежании лишнего запроса в этой ситуации полезно использовать флаг `-W`.

```
--maintenance-db=dbname
```

Указывает имя базы данных, в качестве опорной, для определения необходимости кластеризации других баз данных. По умолчанию используется `postgres`, а если она не существует, то используется `template1`.

1.4. Переменные окружения

```
PGDATABASE  
PGHOST  
PGPORT  
PGUSER
```

Параметры подключения по умолчанию.

Эта утилита, как и большинство других утилит Синергия-БД, также

использует переменные среды, поддерживаемые `libpq`.

1.5. Диагностика

В случае возникновения трудностей, обратитесь к `CLUSTER` и `psql`. Переменные окружения и параметры подключения по умолчанию `libpq` будут применены при запуске утилиты, это следует учитывать при диагностике.

1.6. Примеры

Для кластеризации базы данных `test`:

```
$ clusterdb test
```

Для кластеризации отдельной таблицы `foo` базы данных `xyzzu`:

```
$ clusterdb --table foo xyzzu
```

2. CREATEDB

`createdb` – создать базу данных Синергия-БД.

2.1. Синтаксис

```
createdb [ опции подключения ...] [ параметр ...] [dbname  
[описание]]
```

2.2. Описание

`createdb` создает базу данных Синергия-БД.

Чаще всего пользователь, выполняющий эту команду, назначается владельцем создаваемой базы данных. Однако можно указать владельца явным образом с помощью флага `-O`, если у текущего пользователя достаточно привилегий.

`createdb` это обертка для SQL-команды `CREATE DATABASE`. Оба пути создания базы идентичны.

2.3. Опции

`createdb` принимает аргументы, описанные ниже.

`dbname`

Указывает имя создаваемой базы. Имя должно быть уникальным в рамках

кластера Синергия-БД. По умолчанию в качестве имени базы данных берется имя текущего системного пользователя.

описание

Добавляет комментарий к создаваемой базе.

-D табличное пространство

--tablespace=табличное пространство

Указывает табличное пространство, используемое по умолчанию. Имя пространства обрабатывается аналогично идентификаторам, заключенными в двойные кавычки.

-e

--echo

Вывести команды к серверу, генерируемые при выполнении createdb.

-E кодировка

--encoding=кодировка

Указывает кодировку базы данных.

-l локаль

--locale=локаль

Указывает локаль базы данных. Имеет эффект одновременно установленных флагов --lc-collate и --lc-ctype.

--lc-collate=локаль

Устанавливает параметр LC_COLLATE для базы данных.

--lc-ctype=локаль

Устанавливает параметр LC_TYPE для базы данных.

-O владелец

--owner=владелец

Указывает пользователя в качестве владельца создаваемой базы. Имя пользователя обрабатывается аналогично идентификаторам, заключенным в двойные кавычки.

-T шаблон

--template=шаблон

Указывает шаблон, на основе которого будет создана база данных. Имя шаблона обрабатывается аналогично идентификаторам, заключенным в двойные

кавычки.

-V
--version

Вывести версию createdb и прервать дальнейшее исполнение.

-?
--help

Вывести помощь по команде createdb и прервать выполнение.

Флаги -D, -l, -E, -O, и -T по назначению соответствуют флагам SQL-команды CREATE DATABASE.

createdb также принимает из командной строки параметры подключения:

-h host
--host=host

Указывает имя компьютера, на котором запущен сервер. Если значение начинается с наклонной черты, то оно интерпретируется как имя директории, где хранится файл Unix-сокета.

-p port
--port=port

Указывает порт TCP или расширение файла Unix-сокета, на котором сервер слушает подключения.

-U username
--username=username

Имя пользователя, под которым производится подключение.

-w
--no-password

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass, попытка соединения не удастся. Опция может быть полезна в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

-W
--password

Принудить createdb к запросу пароля перед подключением к серверу баз данных.

Опция несущественна, т.к. при включенной на сервере аутентификации по

паролю, `createdb` запросит его автоматически. Однако `createdb` сделает лишнее подключение для определения необходимости ввода пароля. Поэтому во избежании лишнего запроса в этой ситуации полезно использовать флаг `-W`.

```
--maintenance-db=dbname
```

Указывает имя опорной базы данных, к которой при создании новой будет произведено подключение. Если имя не указано, то будет использоваться `postgres`. Если же `postgres` не существует, то будет использоваться `template1`.

2.4. Переменные окружения

`PGDATABASE`

Если установлена, то используется при создании базы данных, за исключением ситуации с явным указанием флага командной строки.

`PGHOST`

`PGPORT`

`PGUSER`

Параметры подключения по умолчанию. `PGUSER` указывает имя пользователя при создании базы данных, если не указано явно в командной строке или в переменной окружения `PGDATABASE`.

Эта утилита, как и большинство других утилит Синергия-БД, также использует переменные среды, поддерживаемые `libpq`.

2.5. Диагностика

В случае возникновения трудностей, обратитесь к `CREATE DATABASE` и `psql`. При диагностике нужно учитывать, что при запуске утилиты используются значения переменных окружения и параметров подключения по умолчанию `libpq`.

2.6. Примеры

Для создания базы данных `demo` на сервере, используемом по умолчанию:

```
$ createdb demo
```

Для создания базы `demo` на сервере `eden`, порт `5000`, с кодировкой `LATIN1`:

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
```

```
CREATE DATABASE demo ENCODING 'LATIN1';
```

3. CREATEUSER

`createuser` – завести новую учетную запись Синергия-БД.

3.1. Синтаксис

```
createuser [ опции подключения ... ] [ параметр ... ]  
[ username ]
```

3.2. Описание

`createuser` создает нового пользователя Синергия-БД, а если точнее - роль. Лишь суперпользователь и пользователи с привилегией `CREATEROLE` могут создавать новые роли, таким образом, `createuser` должна запускаться от их лица.

Чтобы создать дополнительного суперпользователя, необходимо подключиться под уже существующим, одной привилегии `CREATEROLE` не достаточно. Поскольку для суперпользователей игнорируются любые проверки в базе данных, стоит внимательно относиться к выдаче таких прав.

`createuser` это обертка для SQL-команды `CREATE ROLE`. Оба пути создания пользователей имеют одинаковый результат.

3.3. Опции

`createuser` принимает аргументы, описанные ниже.

`username`

Указывает Синергия-БД имя создаваемого пользователя. Имя роли должно быть уникальным в рамках кластера Синергия-БД.

`-c number`

`--connection-limit=number`

Устанавливает максимальное допустимое количество соединений для создаваемого пользователя. По умолчанию ограничение в количестве соединений отсутствует.

`-d`

`--createdb`

Разрешает новому пользователю создавать базы данных.

-D
--no-createdb

Запрещает новому пользователю создавать базы данных. Это поведение по умолчанию.

-e
--echo

Вывести команды к серверу, генерируемые при выполнении createuser.

-E
--encrypted

Шифровать пароль пользователя, хранимый в базе. Если флаг не указан, то для пароля используется поведение по умолчанию.

-g role
--role=role

Указывает роль, к которой будет добавлена текущая роль в качестве члена группы. Допускается множественное использование флага -g.

-i
--inherit

Создаваемая роль автоматически унаследует привилегии ролей, в группы которых она войдет. Это поведение по умолчанию.

-I
--no-inherit

Роль не будет наследовать привилегии ролей, в группы которых она войдет.

--interactive

Запросить имя для создаваемого пользователя, а также значения для флагов -d/-D, -r/-R, -s/-S, если они явно не указаны в командной строке.

-l
--login

Позволяет создаваемому пользователю открывать сессии. Это поведение по умолчанию.

-L
--no-login

Запрещает создаваемому пользователю открывать сессии. При этом роль может использоваться для управления привилегиями базы данных.

-N

--unencrypted

Не шифровать пароль, хранимый в базе. Если флаг не указан, то для пароля используется поведение по умолчанию.

-P

--pwprompt

Если флаг указан, то `createuser` запросит пароль для создаваемого пользователя. Если не планируется аутентификация по паролю, то пароль можно не устанавливать.

-r

--createrole

Разрешает новому пользователю создавать другие роли, что означает наделение привилегией `CREATEROLE`.

-R

--no-createrole

Запрещает пользователю создавать новые роли. Это поведение по умолчанию.

-s

--superuser

Создаваемая роль будет иметь права суперпользователя.

-S

--no-superuser

Новый пользователь не будет суперпользователем. Это поведение по умолчанию.

-V

--version

Вывести версию `createuser` и завершить выполнение.

--replication

Создаваемый пользователь будет наделен правом `REPLICATION`. Это рассмотрено подробнее в документации по `CREATE ROLE`.

--no-replication

Создаваемый пользователь не будет иметь привилегии `REPLICATION`. Это рассмотрено подробнее в документации по `CREATE ROLE`.

-?

--help

Вывести помощь по команде `createuser`.

`createuser` также принимает из командной строки параметры подключения:

-h host

--host=host

Указывает имя компьютера, на котором запущен сервер. Если значение начинается с наклонной черты, то оно интерпретируется как имя директории, где хранится файл Unix-сокета.

-p port

--port=port

Указывает TCP-порт или расширение файла Unix-сокета, на котором сервер слушает подключения.

-U username

--username=username

Имя пользователя, под которым производится текущее подключение к базе (это не имя создаваемого пользователя).

-w

--no-password

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Опция может быть полезна в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

-W

--password

Принудить `createuser` к запросу пароля при текущем подключении.

Опция несущественна, т.к. при включенной на сервере аутентификации по паролю, `createuser` запросит его автоматически. Однако `createuser` сделает лишнее подключение для определения необходимости ввода пароля. Поэтому во избежании лишнего запроса в этой ситуации полезно использовать флаг `-W`.

3.4. Переменные окружения

PGHOST

PGPORT

PGUSER

Параметры подключения по умолчанию.

Эта утилита, как и большинство других утилит Синергия-БД, также использует переменные среды, поддерживаемые libpq.

3.5. Диагностика

В случае возникновения трудностей, обратитесь к CREATE ROLE и psql. Переменные окружения и параметры подключения по умолчанию libpq будут применены при запуске утилиты, это следует учитывать при диагностике.

3.6. Примеры

Чтобы создать роль joe на сервере, используемом по умолчанию:

```
$ createuser joe
```

Чтобы создать роль joe на сервере, используемом по умолчанию, с запросом дополнительных параметров:

```
$ createuser --interactive joe
```

Назначить роль суперпользователем? (y/n) n

Разрешить новой роли создавать базы данных? (y/n) n

Разрешить новой роли создавать другие роли? (y/n) n

Чтобы создать того же самого joe на сервере eden, слушающего подключения на порту 5000, с явно заданными параметрами:

```
$ createuser -h eden -p 5000 -S -D -R -e joe
```

```
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE  
INHERIT LOGIN;
```

Чтобы создать роль joe с правами суперпользователя и предустановленным паролем:

```
$ createuser -P -s -e joe
```

Введите пароль для новой роли: хуззу

Подтвердите ввод пароля: хуззу

```
CREATE ROLE joe PASSWORD  
'md5b5f5ba1a423792b526f799ae4eb3d59e' SUPERUSER CREATEDB  
CREATEROLE INHERIT LOGIN;
```

В приведенном примере введенный пароль отображается лишь для отражения сути, на деле же он не выводится на экран. Как видно в выводе лога команд, пароль зашифрован. Если же указан флаг `--unencrypted`, то он отобразится в этом логе неизменным, а также, возможно, и в других логах сервера. По этой причине в этой ситуации стоит использовать флаг `-e` с особой осторожностью.

4. DROPDB

`dropdb` – удалить базу данных Синергия-БД.

4.1. Синтаксис

```
dropdb [ опции подключения ... ] [ параметр ... ] dbname
```

4.2. Описание

`dropdb` удаляет ранее созданную базу данных Синергия-БД, и должна выполняться от имени суперпользователя или её владельца.

`dropdb` это обертка для SQL-команды `DROP DATABASE`. Оба пути удаления баз данных идентичны.

4.3. Опции

Аргументы, которые принимает `dropdb`, описаны ниже.

`dbname`

Указывает имя удаляемой базы данных.

`-e`

`--echo`

Вывести команды к серверу, генерируемые при выполнении `dropdb`.

`-i`

`--interactive`

Выводит вопрос о подтверждении перед удалением.

`-V`

`--version`

Выводит версию `dropdb`.

`--if-exists`

Не считать ошибкой, если база данных не существует. В этом случае будет

выдано замечание.

-?
--help

Вывести справку по команде dropdb.

dropdb также принимает из командной строки параметры подключения:

-h host
--host=host

Указывает имя компьютера, на котором запущен сервер. Если значение начинается с наклонной черты, то оно интерпретируется как имя директории, где хранится файл Unix-сокета.

-p port
--port=port

Указывает TCP-порт или расширение файла Unix-сокета, на котором сервер слушает подключения.

-U username
--username=username

Имя пользователя, под которым производится подключение.

-w
--no-password

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass, попытка соединения не удастся. Опция может быть полезна в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

-W
--password

Принудить dropdb к запросу пароля перед подключением к серверу баз данных.

Опция несущественна, т.к. при включенной на сервере аутентификации по паролю, dropdb запросит его автоматически. Однако dropdb сделает лишнее подключение для определения необходимости ввода пароля. Поэтому во избежании лишнего запроса в этой ситуации полезно использовать флаг -W.

--maintenance-db=dbname

Указывает имя базы данных, к которой будет произведено подключение при удалении целевой. Если имя не указано, то будет использовано postgres. Если же postgres не существует, то будет использоваться template1.

4.4. Переменные окружения

PGHOST
PGPORT
PGUSER

Параметры подключения по умолчанию.

Эта утилита, как и большинство других утилит Синергия-БД, также использует переменные среды, поддерживаемые libpq.

4.5. Диагностика

В случае возникновения трудностей, обратитесь к DROP DATABASE и psql. При диагностике следует учесть, что при запуске утилиты также применяются переменные окружения и параметры подключения по умолчанию libpq.

4.6. Примеры

Для удаления базы данных demo на сервере, используемом по умолчанию:

```
$ dropdb demo
```

Для удаления базы данных demo на сервере eden, слушающим подключения на порту 5000, в интерактивном режиме и выводом запросов к серверу:

```
$ dropdb -p 5000 -h eden -i -e demo
```

База данных «demo» будет удалена навсегда.

Продолжить? (y/n) y

```
DROP DATABASE demo;
```

5. DROPUSER

dropuser – удалить учетную запись пользователя Синергия-БД.

5.1. Синтаксис

```
dropuser [ опции подключения ... ] [ параметр ... ]  
[ username ]
```

5.2. Описание

`dropuser` удаляет ранее созданного пользователя Синергия-БД. Лишь суперпользователь или пользователь с привилегией `CREATEROLE` могут удалять пользователей Синергия-БД. Необходимо быть суперпользователем, чтобы удалить учетную запись другого суперпользователя.

`dropuser` это обертка для SQL-команды `DROP ROLE`. Оба пути удаления пользователей идентичны.

5.3. Опции

`dropuser` принимает в качестве аргументов:

`username`

Указывает имя удаляемой роли Синергия-БД. Если передан флаг `-i/--interactive`, а имя не указано в параметрах команды, его необходимо будет ввести интерактивно.

`-e`
`--echo`

Вывести команды к серверу, генерируемые при выполнении `dropuser`.

`-i`
`--interactive`

Вывести подтверждение об удалении роли, и запросить её имя, если оно не указано в параметрах команды.

`-V`
`--version`

Вывести версию `dropuser`.

`--if-exists`

Перехватить ошибку, если пользователь не существует. В этом случае, вместо ошибки, будет выведено информационное сообщение.

`-?`
`--help`

Вывести справку по команде `dropuser`.

`dropuser` также принимает из командной строки параметры подключения:

`-h host`


```
--host=host
```

Указывает имя компьютера, на котором запущен сервер. Если значение начинается с наклонной черты, то оно интерпретируется как имя директории, где хранится файл Unix-сокета.

```
-p port  
--port=port
```

Указывает TCP-порт или расширение файла Unix-сокета, на котором сервер слушает подключения.

```
-U username  
--username=username
```

Имя пользователя, под которым производится текущее подключение к базе.

```
-w  
--no-password
```

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Опция может быть полезна в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

```
-W  
--password
```

Принудить `dropuser` к запросу пароля перед подключением к серверу баз данных.

Опция несущественна, т.к. при включенной на сервере аутентификации по паролю, `dropuser` запросит его автоматически. Однако `dropuser` сделает лишнее подключение для определения необходимости ввода пароля. Поэтому во избежании лишнего запроса в этой ситуации полезно использовать флаг `-W`.

5.4. Переменные окружения

```
PGHOST  
PGPORT  
PGUSER
```

Параметры подключения по умолчанию

Эта утилита, как и большинство других утилит Синергия-БД, также использует переменные среды, поддерживаемые `libpq`.

5.5. Диагностика

В случае возникновения трудностей, обратитесь к DROP ROLE и psql. При диагностике следует учесть, что при запуске утилиты также применяются переменные окружения и параметры подключения по умолчанию libpq.

5.6. Примеры

Чтобы удалить роль joe на сервере, используемом по умолчанию:

```
$ dropuser joe
```

Чтобы удалить роль joe на сервере eden, слушающем подключения на порту 5000, в интерактивном режиме и с выводом выполняемых команд:

```
$ dropuser -p 5000 -h eden -i -e joe
```

Роль «joe» будет удалена навсегда.

Продолжить? (y/n) y

```
DROP ROLE joe;
```

6. ECPG

ecpg – встроенный SQL C-препроцессор.

6.1. Синтаксис

```
ecpg [ параметр ... ] файл ...
```

6.2. Описание

ecpg это встроенный SQL препроцессор для программ, написанных на языке C. Он преобразует программы на C, содержащие SQL-выражения, заменяя их вызовами встроенных функций. Получаемые на выходе файлы можно затем скомпилировать и скомпоновать.

ecpg преобразует каждый файл, переданный в параметрах в соответствующий файл на C. Входным файлам лучше давать расширения .pgc. В этом случае оно будет автоматически заменяться на .c. Если же расширение входного файла отличается от .pgc, то к имени просто добавится .c. Имя выходного файла можно задать опцией -o.

Данный раздел не содержит описания встроенного SQL-языка.

6.3. Опции

Аргументы, которые принимает `esrg`, описаны ниже.

`-c`

Автоматически генерировать код, написанный на языке C, из кода SQL. Сейчас это справедливо для `EXEC SQL TYPE`.

`-C` режим

Установить режим совместимости. режим может принимать значение `INFORMIX` или `INFORMIX_SE`.

`-D` символ

Определить символ начала команд C-препроцессора.

`-i`

Также разбирать и системные включения.

`-I` каталог

Указать дополнительный путь включаемых файлов, используемый при выполнении `EXEC SQL INCLUDE`. По умолчанию используются `.` (текущая директория), `/usr/local/include`, директория, задаваемая при компиляции Синергия-БД (обычно - `/usr/local/pgsql/include`), и `/usr/include`, в порядке, как это перечислено.

`-o` имя_файла

Указывает `esrg` имя файла, в который будет осуществлен вывод.

`-r` параметр

Определяет поведение времени исполнения. Флаг может принимать следующие значения:

`no_indicator`

Использовать специальные символы для представления значений `null`. Исторически некоторые базы данных используют такой подход.

`prepare`

Сформировать подготовленные выражения. `libesrg` сформирует кеш подготовленных выражений и будет использовать их при необходимости повторно. В случае переполнения кеша, `libesrg` освободит память за счет вытеснения

наименее используемых выражений.

`questionmarks`

Разрешает использовать знак вопроса в качестве аргумента подстановки в целях совместимости. Ранее это было поведением по умолчанию.

`-t`

Включить автоматическую фиксацию транзакций. В этом режиме каждая SQL-команда будет автоматически фиксироваться, пока не будет явно включена в блок транзакции. В режиме по умолчанию команды фиксируются лишь при явном вызове `EXEC SQL COMMIT`.

`-v`

Вывести информацию о версии, а также путях поиска включаемых файлов.

`--version`

Вывести версию `espg`.

`-?`

`--help`

Вывести справку по команде `espg`.

6.4. Замечания

При компиляции полученных файлов, компилятор должен иметь возможность найти заголовочные файлы `ESPG` в директории включений Синергия-БД. Для этого можно использовать флаг `-I` во время компиляции, например, `-I/usr/local/pgsql/include`.

Программы на `C` со встроенным `SQL` необходимо компоновать с библиотекой `libespg`, например, используя флаг компоновщика `-L/usr/local/pgsql/lib -lespg`.

Имена директорий, подходящих для установки, можно найти в разделе `pgbench`.

6.5. Примеры

Если имеется исходный файл на `C` `prog1.pgc` со встроенным `SQL`, можно создать исполняемую программу, используя следующую последовательность

команд:

```
ecpg prog1.pgc  
cc -I/usr/local/pgsql/include -c prog1.c  
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

7. PG_BASEBACKUP

`pg_basebackup` – создать резервную копию кластера Синергия-БД.

7.1. Синтаксис

`pg_basebackup` [параметр ...]

7.2. Описание

`pg_basebackup` предназначен для создания резервных копий работающего кластера баз данных Синергия-БД. Копия создается без оказания влияния на другие клиентские сессии. Она может использоваться для обеих стратегий восстановления - на заданный момент в прошлом и в качестве отправной точки потоковой репликации или через отгрузку логов на резервные сервера.

`pg_basebackup` создает бинарную копию файлов кластера, контролируя режим создания копии автоматически. Резервные копии всегда создаются для кластера целиком и невозможно создать копию для какой-либо сущности базы отдельно. Для этой цели можно использовать, например, утилиту `pg_dump`.

Копия создается через обычное подключение к Синергия-БД, и при этом используется протокол репликации. Подключение должно осуществляться от лица суперпользователя или пользователя с правом `REPLICATION`, а в `pg_hba.conf` должно быть прописано подключение для репликации. Настройка сервера `max_wal_senders` должна позволять выполнить ещё одно подключение для резервирования.

Можно запустить одновременно несколько команд `pg_basebackup`, но с точки зрения производительности лучше делать всего одну копию одновременно, а затем копировать получаемый результат.

`pg_basebackup` может создавать копии не только для мастера, но также и

для резервного сервера. Для этого необходимо настроить возможность принимать подключения реплик на резервном сервере (флаги `max_wal_senders` и `hot_standby`, а также сконфигурировать аутентификацию по хосту). При этом на мастере необходимо включить `full_page_writes`.

Стоит обратить внимание на существующие ограничения при удаленном резервировании, где источником служит резервный сервер:

- файл истории создания резервных копий не создается на кластере;
- нет гарантии, что все необходимые файлы журналов WAL будут доступны на момент завершения создания резервной копии. Если планируется использовать резервную копию для восстановления из журнала, то для обеспечения гарантированной целостности необходимо самостоятельно добавлять их с помощью опции `-x`;
- если резервный сервер переводится в роль мастера во время удаленного резервирования, то операция создания копии не исполнится;
- все необходимые для резервной копии WAL-записи должны содержать полные страницы, что требует включения настройки `full_page_writes` на мастере и отказ в использовании таких утилит, как, например, `pg_compresslog` в качестве значения `archive_command`.

7.3. Опции

Описанные далее аргументы командной строки влияют на размещение и формат вывода.

`-D каталог`
`--pgdata=каталог`

Целевой каталог для записи данных. `pg_basebackup` создаст его и родительские, если необходимо. Каталог может быть создан заранее, но должен быть пустым, иначе возникнет ошибка.

Если резервирование работает в режиме `tar`, а имя каталога имеет значение `-` (тире), то `tar`-файл будет писаться в `stdout`.

Этот флаг является обязательным.

`-F формат`
`--format=формат`

Устанавливает формат вывода. формат может принимать следующие значения:

`p`
`plain`

Записывает выводимые данные в обычные файлы, сохраняя текущую схему размещения каталогов данных и табличных пространств. Если кластер не имеет дополнительных табличных пространств, то вся база будет помещена в заданный каталог. Иначе основной каталог хранения данных будет помещён в целевой каталог, а все остальные табличные пространства – в те же абсолютные пути, в которых они находятся на сервере.

Это формат по умолчанию.

`t`
`tar`

Записывает в целевой каталог файлы в формате tar. Основной каталог хранения данных будет писаться в файл `base.tar`, а табличные пространства – в файлы, именованные в соответствии с их OID.

Если имя целевого каталога задано как `-` (тире), то данные будут писаться в стандартный вывод, что позволяет, например, использовать `gzip`. Это возможно лишь когда не применяются дополнительные табличные пространства и не используется трансляция WAL.

`-r скорость_передачи`
`--max-rate=скорость_передачи`

Максимальная скорость передачи данных с сервера. Значение задаётся в Кб/с. Для установки значения в мегабайтах, можно использовать суффикс `M`. Также допустим суффикс `k`, но он не принципиален. Допустимые значения лежат в рамках между 32 Кб/с и 1024 Мб/с.

Служит для снижения влияния на производительность сервера со стороны работающего `pg_basebackup`.

Параметр всегда влияет на передачу данных каталога хранения. На передачу

данных файлов WAL влияние оказывается лишь при методе сбора `fetch`.

```
-R  
--write-recovery-conf
```

Записать минимальный файл `recovery.conf` в каталог вывода (или базовый архивный файл при использовании формата `tar`) для упрощения настройки ведомого сервера. В файл `recovery.conf` будут записаны параметры соединения и, если указан, слот репликации, который использует `pg_basebackup`, так что впоследствии при потоковой репликации будут использоваться те же параметры.

```
-T старый_каталог=новый_каталог  
--tablespace-mapping=старый_каталог=новый_каталог
```

Переместить табличное пространство из `старого_каталога` в `новый_каталог` в процессе копирования. Чтобы перемещение произошло, в параметре `старый_каталог` должен задаваться в точности путь табличного пространства, как он определён. Но не будет ошибкой, если табличного пространства, на которое указывает `старый_каталог`, в архиве не окажется. И `старый_каталог`, и `новый_каталог` должны задаваться абсолютными путями. Если в пути встречается символ `=`, его необходимо экранировать обратной косой чертой. Этот параметр можно добавить несколько раз для нескольких табличных пространств.

Если табличное пространство перемещается таким способом, символьные ссылки внутри основного каталога хранения данных также приводятся в соответствие с новым местоположением. Таким образом, для экземпляра сервера подготавливается новый каталог данных, в котором все табличные пространства оказываются в новом расположении.

```
--waldir=каталог_wal
```

Указывает размещение каталога хранения журнала предзаписи. Задаваемый в параметре `каталог_wal` путь должен быть абсолютным. Каталог с журналом предзаписи можно задать только при создании копии в простом режиме.

```
-X метод  
--wal-method=метод
```

Включает все необходимые файлы журналов предзаписи (файлы WAL) в

резервную копию. В том числе включаются все журналы предзаписи, сгенерированные в процессе создания резервной копии. Если только не выбран метод `none`, главный процесс БД может быть запущен непосредственно с восстановленным каталогом, без обращения к дополнительному архиву журналов; таким образом будет получена полностью самодостаточная резервная копия.

Для сбора журналов предзаписи поддерживаются следующие методы:

`n`
`none`

Не включать журнал предзаписи в резервную копию.

`f`
`fetch`

Файлы журнала предзаписи собираются в конце процесса копирования. Таким образом необходимо установить достаточно большое значение параметра `wal_keep_segments`, чтобы избежать преждевременного удаления файлов журнала. В случае удаления файлов до завершения процесса копирования возникнет ошибка, а копия будет непригодной к использованию.

Когда используется формат `tar`, файлы журнала предзаписи записываются в файл `base.tar`.

`s`
`stream`

Передавать журнал предзаписи в процессе создания резервной копии. При этом открывается второе соединение к серверу, по которому будет передаваться журнал предзаписи, одновременно с созданием резервной копии. Таким образом будут использоваться два подключения из разрешённых параметром `max_wal_senders`. И если клиент будет успевать получать журнал предзаписи, ведущему серверу не потребуется хранить дополнительные файлы журнала.

Когда используется формат `tar`, файлы журнала предзаписи сохраняются в отдельном файле с именем `pg_wal.tar` (если версия сервера ниже 10, файл будет называться `pg_xlog.tar`).

Это значение по умолчанию.

-z

--gzip

Включает gzip-сжатие выводимого tar-файла с уровнем компрессии по умолчанию. Сжатие поддерживается только для формата tar, при этом ко всем именам файлов tar добавляется суффикс .gz.

-Z *уровень*

--compress=*уровень*

Включает gzip-сжатие выводимого tar-файла и задаёт уровень сжатия от 0 (без сжатия) до 9 (максимальное сжатие). Сжатие поддерживается только для формата tar, при этом ко всем именам файлов tar добавляется суффикс .gz.

Описанные далее аргументы командной строки влияют на генерацию резервной копии и ход выполнения приложения.

-c *fast|spread*

--checkpoint=*fast|spread*

Устанавливает режим контрольных точек: *fast* (быстрый) или *spread* (протяжённый, по умолчанию).

-C

--create-slot

С данным параметром до начала копирования будет создан слот репликации с именем, заданным в --slot. Если такой слот уже существует, выдаётся ошибка.

-l *метка*

--label=*метка*

Устанавливает метку для созданной резервной копии. Если не указана, то по умолчанию будет использовано значение «pg_basebackup base backup».

-n

--no-clean

По умолчанию, когда программа pg_basebackup прерывается с ошибкой, она удаляет все каталоги, которые она могла создать, прежде чем обнаружила, что не может завершить задание (например, каталог данных и каталог журнала предзаписи). Данный ключ отключает эту очистку и тем самым полезен для отладки.

Заметьте, что каталоги табличных пространств не очищаются в любом случае.

-N

--no-sync

По умолчанию `pg_basebackup` ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром `pg_basebackup` завершается немедленно, то есть выполняется быстрее, но в случае неожиданного сбоя операционной системы резервная копия может оказаться испорченной. Вообще этот параметр предназначен прежде всего для тестирования, для производственной среды он не подходит.

```
-P  
--progress
```

Включает отчёт о прогрессе. Если этот режим включён, то во время создания копии будет передаваться примерный процент выполнения. Так как данные в базе могут меняться во время копирования, это значение будет лишь приближённым и может достигать не точно 100%. В частности, когда в копию включается журнал WAL, конечный размер невозможно предсказать заранее, и в этом случае ожидаемый конечный размер будет увеличиваться, превысив ориентировочный полный размер без WAL.

Если режим включён, то процесс копирования начнется с перечисления размеров всей базы, а затем продолжится отправкой непосредственно данных. Это может немного увеличить время операции, в частности, пройдет больше времени до начала передачи данных.

```
-S имя_слота  
--slot=имя_слота
```

Этот параметр может применяться только вместе с `-X stream`. Он устанавливает использование заданного слота репликации при потоковой передаче WAL. Если базовая копия предназначена для использования на ведомом сервере с потоковой репликацией, затем должен использоваться слот с тем же именем в `recovery.conf`. Тем самым гарантируется, что сервер не удалит никакие необходимые данные WAL после того, как базовая копия будет получена, и до того, как начнётся потоковая репликация.

В случае отсутствия ключа `-C` требуется, чтобы указанный слот репликации уже существовал.

Если этот ключ не указан и сервер поддерживает временные слоты репликации (они появились в версии 10), для трансляции WAL автоматически используется временный слот репликации.

`-v`
`--verbose`

Включает режим подробного вывода. Будет выводиться некоторая дополнительная информация при начале и завершении, а также имена обрабатываемых файлов, если включён отчёт о прогрессе.

`--no-slot`

Этот ключ предотвращает создание временного слота репликации во время резервного копирования, даже если это поддерживается сервером.

Временные слоты репликации создаются по умолчанию, если при трансляции журнала имя слота не задаётся в параметре `-S`.

Основное предназначение этого ключа в том, чтобы можно было сделать базовую резервную копию, когда на сервере не хватает свободных слотов репликации. Использование слотов репликации почти всегда предпочтительнее, так как при этом предотвращается удаление во время резервного копирования необходимых файлов WAL с сервера.

`--no-verify-checksums`

Отключает проверку контрольных сумм, если они включены на сервере, с которого делается резервная копия.

По умолчанию контрольные суммы проверяются и при выявлении их несоответствия выдаётся ненулевой код завершения. Однако базовая резервная копия не удаляется в этом случае, как и с ключом `--no-clean`.

Далее описаны параметры управления подключением.

`-d` строка подключения
`--dbname=строка подключения`

Указывает параметры подключения к серверу в формате строки подключения.

Опция названа `--dbname` по аналогии с другими клиентскими приложениями, но, т.к. `pg_basebackup` не подключается ни к одной базе индивидуально, её имя в строке подключения игнорируется.

```
-h host  
--host=host
```

Указывает имя компьютера, на котором запущен работающий сервер баз данных. Если значение начинается с косой черты, то оно трактуется как название директории, где хранится файл Unix-сокета. Значение по умолчанию берется из переменной окружения PGHOST, если она установлена. Также производится попытка подключения по Unix-сокету.

```
-p port  
--port=port
```

Указывает TCP-порт или расширение локального файла Unix-сокета, на котором сервер слушает подключения. По умолчанию берется значение переменной окружения PGPORT, если оно установлено, либо значение времени компиляции.

```
-s interval  
--status-interval=interval
```

Указывает интервал в секундах между отправкой пакетов статуса, отправляемых на сервер. Это позволяет упростить мониторинг прогресса. Чтобы выключить периодическое обновление статуса, необходимо установить значение в ноль. При этом обновление будет отправляться по запросу сервера для избежания отсоединения по истечению времени. Значение по умолчанию составляет 10 секунд.

```
-U username  
--username=username
```

Имя пользователя, под которым производится подключение.

```
-w  
--no-password
```

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass, попытка соединения не удастся. Опция может быть полезна в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

```
-W  
--password
```

Принуждает pg_basebackup к запросу пароля перед подключением к серверу баз данных.

Опция не существенна, т.к. при включенной на сервере аутентификации по паролю, `pg_basebackup` запросит его автоматически. Однако `pg_basebackup` сделает лишнее подключение для определения необходимости ввода пароля. Поэтому во избежании лишнего запроса в этой ситуации полезно использовать флаг `-W`.

Другие флаги:

`-V`

`--version`

Вывести версию `pg_basebackup`.

`-?`

`--help`

Вывести справку по команде `pg_basebackup`.

7.4. Переменные окружения

Как и большинство других утилит Синергия-БД, приложение также использует переменные окружения, поддерживаемые `libpq`.

7.5. Замечания

Прежде чем начнётся копирование, на сервере с копируемой базой необходимо выполнить контрольную точку. И если копирование запускается без ключа `--checkpoint=fast`, это может занять некоторое время, в течение которого `pg_basebackup` не будет проявлять никакой активности.

Резервная копия будет включать в себя все файлы каталога хранения данных и табличных пространств, а также конфигурационные файлы и прочие файлы, размещённые в каталоге данных, за исключением определённых временных файлов, принадлежащих Синергия-БД. Однако копируются лишь простые файлы и каталоги, кроме них, сохраняются только символические ссылки на табличные пространства. Символические ссылки, указывающие на определённые каталоги, известные Синергия-БД, копируются как пустые каталоги. Другие символические ссылки и файлы спецустройств игнорируются.

Если не указан параметр `--tablespace-mapping`, табличные пространства

в простом формате будут копироваться в тот же путь, который они имеют на сервере. Поэтому при наличии табличных пространств создать базовую копию в простом формате на том же сервере не удастся, так как копия будет направлена в те же каталоги, где располагаются исходные табличные пространства.

Когда применяется режим формата `tar`, пользователь должен позаботиться о том, чтобы все архивы `tar` были распакованы до запуска сервера Синергия-БД. Если имеются дополнительные табличные пространства, архивы `tar` для них должны быть распакованы в правильные каталоги. В таком случае для этих табличных пространств сервером будут созданы символические ссылки, согласно содержимому файла `tablespace_map`, включённого в архив `base.tar`.

`pg_basebackup` сохранит разрешения для группы в форматах `plain` и `tar`, если разрешения для группы установлены в исходном кластере.

7.6. Примеры

Создание резервной копии сервера `mydbserver` и сохранение её в локальном каталоге `/var/lib/sdb/sdb-11/data`:

```
$
```

```
pg_basebackup -h mydbserver -D
/var/lib/sdb/sdb-11/data
```

Создание резервной копии локального сервера в отдельных сжатых файлах `tar` для каждого табличного пространства и сохранение их в каталоге `backup` с индикатором прогресса в процессе выполнения:

```
$
```

```
pg_basebackup -D backup -Ft -z -P
```

Создание резервной копии локальной базы данных с одним табличным пространством и сжатие её с помощью `bzip2`:

```
$
```

```
pg_basebackup -D - -Ft -X fetch |
```

```
bzip2 > backup.tar.bz2
```

Эта команда прервётся с ошибкой, если в базе данных будет несколько табличных пространств.

Создание резервной копии локальной базы данных с перемещением табличного пространства /opt/ts в ./backup/ts:

```
$pg_basebackup -D backup/data -T  
/opt/ts=$(pwd) /backup/ts
```

8. PGBENCH

pgbench – запустить тест производительности Синергия-БД.

8.1. Синтаксис

```
pgbench-i [параметр...] [имя_бд]  
pgbench [параметр...] [имя_бд]
```

8.2. Описание

pgbench – это простая программа для запуска тестов производительности Синергия-БД. Она многократно выполняет одну последовательность команд, возможно в параллельных сеансах базы данных, а затем вычисляет среднюю скорость транзакций (число транзакций в секунду). По умолчанию pgbench тестирует сценарий, примерно соответствующий TPC-B, который состоит из пяти команд SELECT, UPDATE и INSERT в одной транзакции. Однако вы можете легко протестировать и другие сценарии, написав собственные скрипты транзакций.

Типичный вывод pgbench выглядит так:

```
transaction type: <builtin: TPC-B (sort of)>  
scaling factor: 10  
query mode: simple  
number of clients: 10  
number of threads: 1  
number of transactions per client: 1000  
number of transactions actually processed: 10000/10000
```



```
tps = 85.184871 (including connections establishing)
```

```
tps = 85.296346 (excluding connections establishing)
```

В первых шести строках выводятся значения некоторых самых важных параметров. В следующей строке показывается количество выполненных и запланированных транзакций (это будет произведение числа клиентов и числа транзакций для одного клиента); эти количества будут равны, если только выполнение не завершится досрочно. В режиме -Т выводится только число фактически выполненных транзакций. В последних двух строках показывается число транзакций в секунду, подсчитанное с учётом и без учёта времени установления подключения к серверу.

Для запускаемого по умолчанию теста типа ТРС-В требуется предварительно подготовить определённые таблицы. Чтобы создать и наполнить эти таблицы, следует запустить `pgbench` с ключом `-i` (инициализировать). Если вы применяете нестандартный скрипт, это не требуется, но тем не менее нужно подготовить конфигурацию, нужную вашему тесту. Запуск инициализации выглядит так:

```
pgbench -i [другие-параметры] имя_базы
```

где `имя_базы` – имя уже существующей базы, в которой будет проводиться тест. Чтобы указать, как подключиться к серверу баз данных, вы также можете добавить параметры `-h`, `-p` и/или `-U`.

Примечание. `pgbench -i` создаёт четыре таблицы `pgbench_accounts`, `pgbench_branches`, `pgbench_history` и `pgbench_tellers`, предварительно уничтожая существующие таблицы с этими именами. Если вы вдруг используете эти имена в своей базе данных, обязательно переключитесь на другую базу!

С «коэффициентом масштаба», по умолчанию равным 1, эти таблицы изначально содержат такое количество строк:

table	# of rows
pgbench_branches	1
pgbench_tellers	10
pgbench_accounts	100000
pgbench_history	0

Эти числа можно (и в большинстве случаев даже нужно) увеличить,

воспользовавшись параметром `-s` (коэффициент масштаба). При этом также может быть полезен ключ `-F` (фактор заполнения).

Подготовив требуемую конфигурацию, можно запустить тест производительности командой без `-i`, то есть:

```
pgbench [параметры] имя_базы
```

Практически во всех случаях, чтобы получить полезные результаты, необходимо передать какие-либо дополнительные параметры. Наиболее важные параметры: `-c` (число клиентов), `-t` (число транзакций), `-T` (длительность) и `-f` (файл со скриптом). Полный список параметров приведён ниже.

8.3. Параметры

Следующий список разделён на три подраздела: одни параметры используются при инициализации базы данных, другие при проведении тестирования, а третьи в обоих случаях.

8.3.1. Параметры инициализации

Аргументы командной строки для инициализации, которые принимает `pgbench` описаны ниже.

```
-i  
--initialize
```

Требуется для вызова режима инициализации.

```
-I этапы_инициализации  
--init-steps=этапы_инициализации
```

Выполнять только выбранные из всех обычных подготовительных этапов. В параметре `этапы_инициализации` отдельные символы для каждого этапа выбирают, какие этапы должны выполняться. Все этапы выполняются в определённом порядке. Список этапов по умолчанию: `dtgvp`. Полный перечень подготовительных этапов:

`d` (`Drop`, удалить)

Удалить все существующие таблицы `pgbench`.

`t` (`create Tables`, создать таблицы)

Создать таблицы, используемые стандартным сценарием `pgbench`, а именно: `pgbench_accounts`, `pgbench_branches`, `pgbench_history` и

`pgbench_tellers.`

`g` (Generate data, сгенерировать данные)

Сгенерировать данные и загрузить их в стандартные таблицы, заменив все уже существующие данные.

`v` (Vacuum, очистка)

Вызывать VACUUM для стандартных таблиц.

`p` (create Primary keys, создать первичные ключи)

Создать первичные ключи в стандартных таблицах.

`f` (create Foreign keys, создать внешние ключи)

Создать ограничения внешних ключей между стандартными таблицами.

Заметьте, что это действие по умолчанию не выполняется.

`-F` фактор_заполнения

`--fillfactor=` фактор_заполнения

Создать таблицы `pgbench_accounts`, `pgbench_tellers` и `pgbench_branches` с заданным фактором заполнения. Значение по умолчанию – 100.

`-n`

`--no-vacuum`

Не выполнять очистку во время инициализации. Этот параметр выключает этап инициализации `v`, даже если он был указан в `-I`.

`-q`

`--quiet`

Переключить вывод в немногословный режим, когда выводится только одно сообщение о прогрессе в 5 секунд. В режиме по умолчанию одно сообщение выводится на каждые 100000 строк, при этом за секунду обычно выводится довольно много строк (особенно на хорошем оборудовании).

`-s` коэффициент_масштаба

`--scale=` коэффициент_масштаба

Умножить число генерируемых строк на заданный коэффициент. Например, с ключом `-s 100` в таблицу `pgbench_accounts` будут записаны 10 000 000 строк. Значение по умолчанию – 1. При коэффициенте, равном 20 000 или больше, столбцы, содержащие идентификаторы счетов (столбцы `aid`), перейдут к большим целым числам (типу `bigint`), чтобы в них могли уместиться все возможные значения

идентификаторов.

```
--foreign-keys
```

Создать ограничения внешних ключей между стандартными таблицами. Этот ключ добавляет этап `f` к последовательности подготовительных этапов, если он отсутствует.

```
--index-tablespace=табл_пространство_индексов
```

Создать индексы в указанном табличном пространстве, а не в пространстве по умолчанию.

```
--tablespace=табличное_пространство
```

Создать таблицы в указанном табличном пространстве, а не в пространстве по умолчанию.

```
--unlogged-tables
```

Создать все таблицы как нежурналируемые, а не как постоянные таблицы.

8.3.2. Параметры тестирования производительности

`pgbench` принимает аргументы командной строки для тестирования производительности, описанные ниже.

```
-b имя_скрипта[@вес]
```

```
--builtin=имя_скрипта[@вес]
```

Добавляет указанный встроенный скрипт в список выполняемых скриптов. Необязательный целочисленный вес, задаваемый после `@`, позволяет изменить вероятность выбора скрипта. Если он не указан, подразумевается значение 1. В число встроенных скриптов входят `tpcb-like`, `simple-update` и `select-only`. Также принимаются однозначные начала их имён. Со специальным именем `list` программа выводит список встроенных скриптов и немедленно завершается.

```
-c клиенты
```

```
--client= клиенты
```

Число имитируемых клиентов, то есть число одновременных сеансов базы данных. Значение по умолчанию – 1.

```
-C
```

```
--connect
```

Устанавливать новое подключение для каждой транзакции вместо одного для каждого клиента. Это полезно для оценивания издержек подключений.

-d
--debug

Выводить отладочные сообщения.

-D имя_переменной = значение
--define= имя_переменной = значение

Определить переменную для пользовательского скрипта (см. ниже). Параметр -D может добавляться неоднократно.

-f имя_файла[@вес]
--file= имя_файла[@вес]

Добавить в список выполняемых скриптов скрипт транзакции из файла имя_файла. Изменить вероятность выбора этого скрипта позволяет необязательный целочисленный вес, задаваемый после @. Подробности описаны ниже.

-j потоки
--jobs= потоки

Число рабочих потоков в rgbench. Использовать нескольких потоков может быть полезно на многопроцессорных компьютерах. Клиенты распределяются по доступным потокам равномерно, насколько это возможно. Значение по умолчанию – 1.

-l
--log

Записать информацию о каждой транзакции в файл протокола. Подробности описаны ниже.

-L предел
--latency-limit= предел

Транзакции, продолжающиеся дольше указанного предела (в миллисекундах), подсчитываются и отмечаются отдельно, как опаздывающие.

В режиме ограничения скорости (--rate=...) транзакции, которые отстают от графика более чем на заданный предел (в мс) и поэтому никак не могут уложиться в отведённый интервал, не передаются серверу вовсе. Они подсчитываются и

отмечаются отдельно как пропущенные.

```
-М режим_запросов  
--protocol= режим_запросов
```

Протокол, выбираемый для передачи запросов на сервер:

simple: использовать протокол простых запросов.

extended: использовать протокол расширенных запросов.

prepared: использовать протокол расширенных запросов с подготовленными операторами.

По умолчанию выбирается протокол простых запросов.

```
-n  
--no-vacuum
```

Не производить очистку таблиц перед запуском теста. Этот параметр необходим, если вы применяете собственный сценарий, не затрагивающий стандартные таблицы `pgbench_accounts`, `pgbench_branches`, `pgbench_history` и `pgbench_tellers`.

```
-N  
--skip-some-updates
```

Запустить встроенный упрощённый скрипт `simple-update`. Краткий вариант записи `-b simple-update`.

```
-P сек  
--progress= сек
```

Выводить отчёт о прогрессе через заданное число секунд (сек). Выдаваемый отчёт включает время, прошедшее с момента запуска, скорость (в TPS) с момента предыдущего отчёта, а также среднее время ожидания транзакций и стандартное отклонение. В режиме ограничения скорости (`-R`) время ожидания вычисляется относительно назначенного времени запуска транзакции, а не фактического времени её начала, так что оно включает и среднее время отставания от графика.

```
-r  
--report-latencies
```

Выводить по завершении тестирования средняя время ожидания операторов (время выполнения с точки зрения клиента) для каждой команды. Подробности описаны ниже.

-R скорость передачи

--rate= скорость передачи

Выполнять транзакции, ориентируясь на заданную скорость, а не максимально быстро (по умолчанию). Скорость задаётся в транзакциях в секунду. Если заданная скорость превышает максимально возможную, это ограничение скорости не повлияет на результаты.

Для получения нужной скорости транзакции запускаются со случайными задержками, имеющими распределение Пуассона. При этом запланированное время запуска отсчитывается от начального времени, а не от завершения предыдущей транзакции. Это означает, что если какие-то транзакции отстанут от изначально рассчитанного времени завершения, всё же возможно, что последующие нагонят график.

В режиме ограничения скорости время ожидания транзакций, выводимое по итогам тестирования, вычисляется, исходя из запланированного времени запуска, так что в него входит время, которое очередная транзакция должна была ждать завершения предыдущей транзакции. Это время называется временем отклонения от графика, и его среднее и максимальное значения выводятся отдельно. Время ожидания транзакций с момента их фактического запуска, то есть время, потраченное на выполнение транзакций в базе данных, можно получить, если вычесть время отклонения от графика из времени ожидания транзакций.

Если ограничение --latency-limit задаётся вместе с --rate, транзакция может заведомо не вписываться в отведённое ей время, если предыдущая транзакция завершится слишком поздно, так как ожидаемое время окончания транзакции отсчитывается от времени запуска по графику. Такие транзакции не передаются серверу, а пропускаются и подсчитываются отдельно.

Большое значение отклонения от графика свидетельствует о том, что система не успевает выполнять транзакции с заданной скоростью и выбранным числом клиентов и потоков. Когда среднее время ожидания транзакции превышает запланированный интервал между транзакциями, каждая последующая транзакция будет отставать от графика, и чем дольше будет выполняться тестирование, тем

больше будет отставание. Когда это наблюдается, нужно уменьшить скорость транзакций.

```
-s коэффициент_масштаба  
--scale= коэффициент_масштаба
```

Показать заданный коэффициент масштаба в выводе `pgbench`. Для встроенных тестов это не требуется; корректный коэффициент масштаба будет получен в результате подсчёта строк в таблице `pgbench_branches`. Однако при использовании только нестандартных тестов (запускаемых с ключом `-f`) без этого параметра в качестве коэффициента масштаба будет выводиться 1.

```
-S  
--select-only
```

Запустить встроенный скрипт `select-only` (только выборка). Краткий вариант записи `-b select-only`.

```
-t транзакции  
--transactions= транзакции
```

Число транзакций, которые будут выполняться каждым клиентом (по умолчанию 10).

```
-T секунды  
--time= секунды
```

Выполнять тест с ограничением по времени (в секундах), а не по числу транзакций для каждого клиента. Параметры `-t` и `-T` являются взаимоисключающими.

```
-v  
--vacuum-all
```

Очищать все четыре стандартные таблицы перед запуском теста. Без параметров `-n` и `-v` `pgbench` будет очищать от старых записей таблицы `pgbench_tellers` и `pgbench_branches`, а также опустошать `pgbench_history`.

```
--aggregate-interval=секунды
```

Длительность интервала агрегации (в секундах). Может использоваться только с ключом `-l`. С данным параметром в протокол выводится сводка по интервалам, как описано ниже.

`--log-prefix=префикс`

Задать префикс имён файлов для файлов протоколов, создаваемых с ключом `--log`. Префикс по умолчанию – `pgbench_log`.

`--progress-timestamp`

При отображении прогресса (с параметром `-P`) выводить текущее время (в формате Unix), а не количество секунд от начала запуска. Время задаётся в секундах с точностью до миллисекунд. Это помогает сравнивать журналы, записываемые разными средствами.

`--random-seed= ЗАТРАВКА`

Установить затравку для генератора случайных чисел. Инициализирует генератор случайных чисел, который затем выдаёт последовательность начальных состояний отдельных генераторов для каждого потока. ЗАТРАВКА может принимать следующие значения: `time` (по умолчанию, затравка базируется на текущем времени), `rand` (задействовать надёжный генератор случайных чисел или выдать ошибку, если он отсутствует) или беззнаковое десятичное число. Генератор случайных чисел может вызываться явно из скрипта `pgbench` (функциями `random...`) или неявно (например, для планирования выполнения транзакций с ключом `--rate`). В случае установки значения явным образом оно выводится в терминале. Любое значение, допустимое в качестве ЗАТРАВКИ, можно также задать в переменной окружения `PGBENCH_RANDOM_SEED`. Чтобы заданная затравка применялась во всех возможных случаях использования, задайте этот параметр первым или установите переменную окружения.

Явное указание определённой затравки позволяет точно воспроизвести выполнение `pgbench` в части использования случайных чисел. Так как случайное состояние поддерживается внутри потока, это означает, что выполнение `pgbench` при одинаковых запусках повторится в точности, если один поток используется одним клиентом и отсутствуют внешние зависимости или зависимости от данных. Со статистической точки зрения точное воспроизведение выполнения нежелательно, так как это может скрыть вариативность производительности или показать завышенную скорость, например из-за попадания в одни и те же страницы данных.

Однако это может быть очень полезно для отладки, например, для повторения редкого сценария, приводящего к ошибке. Используйте данную возможность обдуманно.

`--sampling-rate=скорость передачи`

Частота выборки для записи данных в протокол, изменяя которую можно уменьшить объём протокола. При указании этого параметра в протокол выводится информация только о заданном проценте транзакций. Со значением 1.0 в нём будут отмечаться все транзакции, а с 0.05 только 5%.

Обработывая протокол, не забудьте учесть частоту выборки. Например, вычисляя скорость (TPS), вам нужно будет соответственно умножить содержащиеся в нём числа (то есть, с частотой выборки 0.01 вы получите только 1/100 фактической скорости).

8.3.3. Общие параметры

`pgbench` принимает общие аргументы командной строки, описанные ниже.

`-h` компьютер

`--host=` компьютер

Адрес сервера баз данных.

`-p` порт

`--port=` порт

Номер порта сервера баз данных.

`-U` имя_пользователя

`--username=` имя_пользователя

Имя пользователя для подключения.

`-V`

`--version`

Вывести версию `pgbench` и завершиться.

`-?`

`--help`

Вывести справку об аргументах командной строки `pgbench` и завершиться.

Примечание. Каково содержание «транзакции», которую выполняет `pgbench`? Программа `pgbench` выполняет тестовые скрипты, выбирая их случайным образом из заданного списка. В том числе это встроенные скрипты, заданные

аргументами `-b`, и пользовательские скрипты, заданные аргументами `-f`. Для каждого скрипта можно задать относительный вес после `@`, чтобы скорректировать вероятность его выбора. Вес по умолчанию – 1. Скрипты с весом 0 игнорируются.

Стандартный встроенный скрипт (также вызываемый с ключом `-b tpcb-like`) выдаёт семь команд в транзакции со случайно выбранными `aid`, `tid`, `bid` и `balance`. Его сценарий написан по мотивам теста производительности TPC-B, но это не собственно TPC-B, потому он называется так.

```
BEGIN;

UPDATE pgbench_accounts SET abalance = abalance + :delta
WHERE aid = :aid;

SELECT abalance FROM pgbench_accounts WHERE aid = :aid;

UPDATE pgbench_tellers SET tbalance = tbalance + :delta
WHERE tid = :tid;

UPDATE pgbench_branches SET bbalance = bbalance + :delta
WHERE bid = :bid;

INSERT INTO pgbench_history (tid, bid, aid, delta,
mtime) VALUES (:tid, :bid, :aid, :delta,
CURRENT_TIMESTAMP);

END;
```

При выборе встроенного скрипта `simple-update` (или указании `-N`) шаги 4 и 5 исключаются из транзакции. Это позволяет избежать конкуренции при обращении к этим таблицам, но тест становится ещё менее похожим на TPC-B.

При выборе встроенного теста `select-only` (или указании `-S`) выполняется только `SELECT`.

8.3.4. Пользовательские скрипты

Программа `pgbench` поддерживает запуск пользовательских сценариев оценки производительности, позволяя заменять стандартный скрипт транзакции (описанный выше) скриптом, считываемым из файла (с параметром `-f`). В этом случае «транзакцией» считается одно выполнение данного скрипта.

Файл скрипта содержит одну или несколько команд SQL, разделённых точкой с запятой. Пустые строки и строки, начинающиеся с `--`, игнорируются. В файлах скриптов также могут содержаться «метакоманды», которые обрабатывает сама

программа `pgbench`, как описано ниже.

Примечание. В версии СУБД «Синергия-БД 1», SQL-команды в файлах скриптов завершались символами перевода строки, и поэтому они не могли занимать несколько строк. Теперь для разделения последовательных команд SQL требуется добавлять точку с запятой (хотя без неё можно обойтись в конце SQL-команды, за которой идёт метакоманда). Если вам нужно создать файл скрипта, работающий и со старыми версиями `pgbench`, записывайте каждую команду SQL в отдельной строке и завершайте её точкой с запятой.

Для файлов скриптов реализован простой механизм подстановки переменных. Имя переменных должно состоять из букв (буквы могут быть не латинскими), цифр и подчёркиваний. Переменные можно задать в командной строке параметрами `-D`, описанными выше, или метакомандами, рассматриваемыми ниже. Помимо переменных, которые можно установить параметрами командной строки `-D`, есть несколько автоматически устанавливаемых переменных; они перечислены в таблице 8.1. Если значение этих переменных задаётся в параметре `-D`, оно переопределяет автоматическое значение. Когда значение переменной определено, его можно вставить в команду SQL, написав `:имя_переменной`. Каждый клиентский сеанс, если их несколько, получает собственный набор переменных.

Таблица 8.1 – Автоматические переменные

Переменная	Описание
<code>client_id</code>	уникальное число, идентифицирующее клиентский сеанс (начиная с нуля)
<code>default_seed</code>	затравка, используемая в хеш-функциях по умолчанию
<code>random_seed</code>	затравка генератора случайных чисел (в отсутствие переопределения с ключом <code>-D</code>)
<code>scale</code>	текущий коэффициент масштаба

Метакоманды в скрипте начинаются с обратной косой черты (`\`) и обычно продолжаются до конца строки, хотя их можно переносить на следующую строку последовательностью символов: обратная косая, возврат каретки. Аргументы метакоманд разделяются пробелами. Поддерживаемые метакоманды представлены ниже:

```
\if выражение
\elif выражение
\else
\endif
```

Эта группа команд реализует вкладываемые условные блоки, подобные `\if`

выражение в `psql`. В качестве условных задаются те же выражения, что и в `\set`, при этом истинным считается любое ненулевое значение.

```
\set имя_переменной выражение
```

Устанавливает для переменной `имя_переменной` значение, вычисленное из выражения. Выражение может содержать константу `NULL`, логические константы `TRUE` и `FALSE`, целочисленные константы (например, `5432`), константы с плавающей точкой (например, `3.14159`), ссылки на переменные `:имя_переменной`, операторы с обычными для SQL приоритетами и ассоциативностью, вызовы функций, общие условные SQL-выражения `CASE`, а также скобки.

Функции и большинство операторов возвращают `NULL` для аргументов `NULL`.

При проверке условия отличные от нуля числовые значения воспринимаются как `TRUE`, а числовые нулевые значения и `NULL` – как `FALSE`.

Если в конструкции `CASE` отсутствует заключительное `ELSE`, значением по умолчанию считается `NULL`.

8.4. Примеры:

```
\set ntellers 10 * :scale
\set aid (1021 * random(1, 100000 * :scale)) % \
        (100000 * :scale) + 1
\set divx CASE WHEN :x <> 0 THEN :y/:x ELSE NULL END
\sleep номер [ us | ms | s ]
```

Приостанавливает выполнение скрипта на заданное число микросекунд (`us`), миллисекунд (`ms`) или секунд (`s`). Когда единицы не указываются, подразумеваются секунды. Здесь число может быть целочисленной константой или ссылкой `:имя_переменной` на переменную с целочисленным значением.

Пример.

```
\sleep 10 ms
\setshell имя_переменной команда [ аргумент ... ]
```

Присваивает переменной `имя_переменной` результат команды оболочки

команда с указанными аргументами. Эта команда должна просто выдать целочисленное значение в стандартный вывод.

Здесь команда и каждый аргумент может быть либо текстовой константой, либо ссылкой на переменную: имя_переменной. Если вы хотите записать аргумент, начинающийся с двоеточия, добавьте перед аргументом дополнительное двоеточие.

Пример.

```
\setshell назначаемая_переменная команда
строковый_аргумент :переменная ::строка_начинающаяся_дво
еточием
\shell команда [ аргумент ... ]
```

Действует так же, как и \setshell, но не учитывает результат команды.

Пример.

```
\shell команда
строковый_аргумент :переменная ::строка_начинающаяся_дво
еточием
```

8.5. Встроенные операторы

Арифметические, битовые и логические операторы, а также операторы сравнения, перечисленные в таблице 8.2, встроены в rgbench и могут применяться в выражениях в \set.

Таблица 8.2 – Операторы rgbench, в порядке увеличения приоритета

Оператор	Описание	Пример	Результат
OR	логическое ИЛИ	5 or 0	TRUE
AND	логическое И	3 and 0	FALSE
NOT	логическое НЕ	not false	TRUE
IS [NOT] (NULL TRUE FALSE)	проверки значений	1 is null	FALSE
ISNULL NOTNULL	проверки на NULL	1 notnull	TRUE
=	равно	5 = 4	FALSE
<>	не равно	5 <> 4	TRUE
!=	не равно	5 != 5	FALSE

Окончание таблицы 8.2

Оператор	Описание	Пример	Результат
<	меньше	5 < 4	FALSE
<=	меньше или равно	5 <= 4	FALSE
>	больше	5 > 4	TRUE
>=	больше или равно	5 >= 4	TRUE
	целочисленное битовое ИЛИ	1 2	3
#	целочисленное битовое исключающее ИЛИ	1 # 3	2
&	целочисленное битовое И	1 & 3	1
~	целочисленное битовое НЕ	~ 1	-2
<<	целочисленный битовый сдвиг влево	1 << 2	4
>>	целочисленный битовый сдвиг вправо	8 >> 2	2
+	сложение	5 + 4	9
-	вычитание	3 - 2.0	1.0
*	умножение	5 * 4	20
/	деление (при целочисленном остаток отбрасывается)	5 / 3	1
%	остаток от деления	3 % 2	1
-	смена знака	- 2.0	-2.0

8.6. Встроенные функции

Функции, перечисленные в таблице 8.3, встроены в `rgbench` и могут применяться в выражениях в метакоманде `\set`.

Таблица 8.3 – Функции `rgbench`

Функция	Тип результата	Описание	Пример	Результат
<code>abs(a)</code>	то же, что и <code>a</code>	модуль числа (абсолютное значение)	<code>abs(-17)</code>	17
<code>debug(a)</code>	то же, что и <code>a</code>	выводит <code>a</code> в <code>stderr</code> и возвращает <code>a</code>	<code>debug(5432.1)</code>	5432.1
<code>double(i)</code>	<code>double</code>	приведение к типу с плавающей точкой	<code>double(5432)</code>	5432.0
<code>exp(x)</code>	<code>double</code>	экспонента	<code>exp(1.0)</code>	2.718281828459045
<code>greatest(a [, ...])</code>	<code>double</code> , если любой из аргументов (<code>a</code>) – <code>double</code> , <code>a</code> иначе целое число	наибольшее значение среди аргументов	<code>greatest(5, 4, 3, 2)</code>	5

Окончание таблицы 8.3

Функция	Тип результата	Описание	Пример	Результат
hash(a [, затравка])	integer	псевдоним для hash_murmur2()	hash(10, 5432)	- 5817877081768 721676
hash_fnv1a(a [, затравка])	integer	хеш FNV-1a	hash_fnv1a(10, 5432)	- 7793829335365 542153
hash_murmur2 (a [, затравка])	integer	хеш MurmurHash2	hash_murmur2(1 0, 5432)	- 5817877081768 721676
int(x)	integer	приведение к целочисленному типу	int(5.4 + 3.8)	9
least(a [, ...])	double, если любой из аргументов (a) – double, а иначе целое число	наименьшее значение среди аргументов	least(5, 4, 3, 2.1)	2.1
ln(x)	double	натуральный логарифм	ln(2.71828182845 9045)	1.0
mod(i, j)	integer	остаток от деления	mod(54, 32)	22
pi()	double	значение константы PI	pi()	3.1415926535 8979323846
pow(x, y), power(x, y)	double	возведение в степень	pow(2.0, 10), power(2.0, 10)	1024.0
random(lb, ub)	integer	случайное целое число с равномерным распределением в интервале [lb, ub]	random(1, 10)	целое между 1 и 10
random_expon ential(lb, ub, параметр)	integer	случайное целое число с экспоненциальным распределением в интервале [lb, ub], см. ниже	random_exponent ial(1, 10, 3.0)	целое между 1 и 10
random_gaussi an(lb, ub, параметр)	integer	целое число с распределением Гаусса в интервале [lb, ub], см. ниже	random_gaussian(1, 10, 2.5)	целое между 1 и 10
random_zipfia n(lb, ub, параметр)	integer	целое число с распределением Ципфа в интервале [lb, ub], см. ниже	random_zipfian(1 , 10, 1.5)	целое между 1 и 10
sqrt(x)	double	квадратный корень	sqrt(2.0)	1.414213562

Функция random выдаёт значения с равномерным распределением, то есть вероятности получения всех чисел в интервале равны. Функции random_exponential, random_gaussian и random_zipfian требуют указания дополнительного параметра типа double, определяющего точную форму

распределения.

Для экспоненциального распределения `parameter` управляет распределением, обрезаая быстро спадающее экспоненциальное распределение в точке `parameter`, а затем это распределение проецируется на целые числа между границами.

$$f(x) = \exp(-\text{parameter} * (x - \min) / (\max - \min + 1)) / (1 - \exp(-\text{parameter}))$$

значение i между $\min$ и $\max$ выдаётся с вероятностью: $f(i) - f(i + 1)$.

Интуиция подсказывает, что чем больше `parameter`, тем чаще будут выдаваться значения, близкие к $\min$, и тем реже значения, близкие к $\max$. Чем `parameter` ближе к 0, тем более плоским (более равномерным) будет распределение. В грубом приближении при таком распределении наиболее частый 1% значений в диапазоне рядом с $\min$ выдаётся `parameter%` времени. Значение `parameter` должно быть строго положительным.

Для распределения Гаусса по интервалу строится обычное нормальное распределение (классическая кривая Гаусса в форме колокола) и этот интервал обрезается в точке `-parameter` слева и `+parameter` справа. Вероятнее всего при таком распределении выдаются значения из середины интервала. Точнее говоря, если $\text{PHI}(x)$ – функция распределения нормальной случайной величины со средним значением μ , равным $(\max + \min) / 2.0$, и

$$f(x) = \text{PHI}(2.0 * \text{parameter} * (x - \mu) / (\max - \min + 1)) / (2.0 * \text{PHI}(\text{parameter}) - 1)$$

тогда значение i между $\min$ и $\max$ включительно выдаётся с вероятностью: $f(i + 0.5) - f(i - 0.5)$. Интуиция подсказывает, что чем больше `parameter`, тем чаще будут выдаваться значения в середине интервала, и тем реже значения у границ $\min$ и $\max$. Около 67% значений будут выдаваться из среднего интервала $1.0 / \text{parameter}$, то есть плюс/минус $0.5 / \text{parameter}$ от среднего значения, и 95% из среднего интервала $2.0 / \text{parameter}$, то есть плюс/минус $1.0 / \text{parameter}$ от среднего значения; например, если `parameter` равен 4.0, 67% значений выдаются из средней четверти ($1.0 / 4.0$) интервала (то есть от $3.0 / 8.0$ до $5.0 / 8.0$) и 95% из средней половины ($2.0 / 4.0$) интервала (из второй и третьей четвертей). Чтобы преобразование Бокса-Мюллера

было быстрым, `parameter` должен быть не меньше 2.0.

Функция `random_zipfian` генерирует приблизительное ограниченное распределение по закону Ципфа. Для параметра в интервале (0, 1) выбирается приблизительный алгоритм, реализованный по книге «Quickly Generating Billion-Record Synthetic Databases» («Быстрая генерация синтетических баз данных с миллиардами записей», Джим Грей и соавторы, SIGMOD 1994). Для параметра в интервале (1, 1000) выбирается метод отклонения, основанный на материале книги «Non-Uniform Random Variate Generation» («Генерация неравномерно распределённых случайных чисел» Люк Деврой, стр. 550-551, Springer 1986). При значении параметра, равном 1.0, распределение не определено. При значениях параметра, превышающих и близких к 1.0, и при небольшом интервале скорость функции будет низкой.

Параметр определяет, насколько неравномерно распределение. Чем больше параметр, тем чаще выдаются значения, близкие к началу интервала. Чем он ближе к 0, тем более плоским (равномерным) будет получающееся распределение. Это распределение таково, что при диапазоне, начинающемся с 1, отношение вероятности получить k к вероятности получения $k+1$ равняется $((k+1)/k)^{\text{параметр}}$. Например, `random_zipfian(1, ..., 2.5)` будет выдавать число 1 примерно в $(2/1)^{2.5} = 5.66$ раз чаще, чем число 2, а оно, в свою очередь, будет выдаваться примерно в $(3/2)^{2.5} = 2.76$ раза чаще, чем 3, и так далее.

Функции хеширования `hash`, `hash_murmur2` и `hash_fnv1a` принимают на вход аргумент и необязательный параметр с затравкой. Если значение затравки не задаётся, используется значение переменной: `default_seed`, которая инициализируется случайным числом (если не задаётся явно ключом командной строки `-D`). Функции хеширования могут использоваться для разброса распределения случайных функций, таких как `random_zipfian` или `random_exponential`. Например, следующий скрипт `rgbench` эмулирует возможную реальную нагрузку, типичную для социальных медиа- и блог-платформ, где

несколько пользователей генерируют лъвиную долю нагрузки:

```
\set r random_zipfian(0, 100000000, 1.07)
\set k abs(hash(:r)) % 1000000
```

В некоторых случаях требуются другие разнообразные распределения, не коррелирующие друг с другом, и тогда может быть полезно явное указание затравки:

```
\set k1 abs(hash(:r, :default_seed + 123)) % 1000000
\set k2 abs(hash(:r, :default_seed + 321)) % 1000000
```

В качестве примера взгляните на встроенное определение транзакции типа TPC-B:

```
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-5000, 5000)
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta
WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta
WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta
WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta,
mtime) VALUES (:tid, :bid, :aid, :delta,
CURRENT_TIMESTAMP);
END;
```

С таким скриптом транзакция на каждой итерации будет обращаться к разным, случайно выбираемым строкам. Этот пример показывает, почему важно, чтобы в каждом клиентском сеансе были собственные переменные – в противном случае они не будут независимо обращаться к разным строкам.

8.7. Протоколирование транзакций

С параметром `-l` (но без `--aggregate-interval`), `pgbench` записывает информацию о каждой транзакции в протокол. Этот файл протокола будет

называться префикс.nnn, где префикс по умолчанию – `pgbench_log`, а `nnn` – PID процесса `pgbench`. Префикс можно сменить, воспользовавшись ключом `--log-prefix`. Если параметр `-j` равен 2 или выше, будет создано несколько рабочих потоков, и каждый будет записывать отдельный протокол. Первый рабочий процесс будет использовать файл с тем же именем, что и в стандартном случае с одним потоком, а файлы остальных потоков будут называться префикс.nnn.mmm, где `mmm` – последовательный номер рабочего процесса, начиная с 1.

Протокол имеет следующий формат:

```
код_клиента число_транзакций длительность номер_скрипта время
я_эпохи время_мкс [отставание_от_графика]
```

Здесь `код_клиента` показывает, в сеансе какого клиента запускалась транзакция, `число_транзакций` отражает, сколько транзакций выполнялось в этом сеансе, `длительность` – общее время транзакций (в микросекундах), `номер_скрипта` показывает, какой файл скрипта использовался (это полезно при указании нескольких скриптов ключами `-f` и `-b`), а `время_эпохи/время_мкс` – отметка времени в формате Unix и смещение в микросекундах (из этих чисел можно получить время стандарта ISO 8601 с дробными секундами), показывающие, когда транзакция была завершена. Поле `отставание_от_графика` представляет разницу между запланированным временем запуска транзакции и фактическим временем запуска (в микросекундах). Оно выводится, только когда применяется параметр `--rate`. Когда одновременно применяются параметры `--rate` и `--latency-limit`, в поле `длительность` для пропущенных транзакций будет выводиться `skipped`.

Фрагмент протокола, полученного при выполнении с одним клиентом:

```
0 199 2241 0 1175850568 995598
0 200 2465 0 1175850568 998079
0 201 2513 0 1175850569 608
0 202 2038 0 1175850569 2663
```

Ещё один пример с `--rate=100` и `--latency-limit=5` (обратите внимание на дополнительный столбец `отставание_от_графика`):

```
0 81 4621 0 1412881037 912698 3005
0 82 6173 0 1412881037 914578 4304
```

07623615.00082-01 34 01-5

0 83 skipped 0 1412881037 914578 5217

0 83 skipped 0 1412881037 914578 5099

0 83 4722 0 1412881037 916203 3108

0 84 4142 0 1412881037 918023 2333

0 85 2465 0 1412881037 919759 740

В этом примере транзакция 82 опоздала, так как её длительность (6.173 мс) превысила ограничение в 5 мс. Следующие две транзакции были пропущены, так как было слишком поздно их начинать.

Когда проводится длительное тестирование с большим количеством транзакций, файлы протоколов могут быть очень объёмными. Чтобы в них записывалась только случайная выборка транзакций, можно запустить команду с параметром `--sampling-rate`.

8.8. Протоколирование с агрегированием

С параметром `--aggregate-interval` протоколы имеют несколько иной формат:

```
начало_интервала число_транзакций сумма_длительности сумма_
длительности_2 мин_длительность макс_длительность
[сумма_задержки сумма_задержки_2 мин_задержка макс_задержка
[пропущено_транзакций] ]
```

Здесь `начало_интервала` – начальное время интервала (в формате времени UNIX), `число_транзакций` – количество транзакций в данном интервале, `сумма_длительности` – суммарная длительность транзакций, `сумма_длительности_2` – сумма квадратов длительностей транзакций в интервале, `мин_длительность` – минимальная длительность в интервале, а `макс_задержка` – максимальная. Следующие поля, `сумма_задержки`, `сумма_задержки_2`, `мин_задержка` и `макс_задержка`, присутствуют, только если применяется параметр `--rate`. Они отражают время, на которое задержалась каждая транзакция, ожидая завершения предыдущей, то есть разницу между временем фактического запуска и запланированным временем. Самое последнее поле, `пропущено_транзакций`, тоже присутствует, только если применяется ключ `--latency-limit`. В нём выводится число транзакций, пропущенных из-за того, что их запуск задержался слишком надолго. Каждая транзакция учитывается в том интервале, в котором она была

зафиксирована.

Пример вывода:

```
1345828501 5601 1542744 483552416 61 2573
1345828503 7884 1979812 565806736 60 1479
1345828505 7208 1979422 567277552 59 1391
1345828507 7685 1980268 569784714 60 1398
1345828509 7073 1979779 573489941 236 1411
```

Заметьте, что простой протокол (без агрегирования) показывает, какой скрипт использовался для каждой транзакции, в отличие от протокола с агрегированием. Таким образом, если вам нужны подобные сведения, но в разрезе скриптов, вам придётся агрегировать данные самостоятельно.

8.9. Время ожидания по операторам

С параметром `-r` программа `pgbench` учитывает время выполнения каждого оператора каждым клиентом. Затем по завершении тестирования она выводит среднее по всем значениям как время ожидания каждого оператора.

Со стандартным скриптом вывод будет примерно таким:

```
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
latency average = 15.844 ms
latency stddev = 2.715 ms
tps = 618.764555 (including connections establishing)
tps = 622.977698 (excluding connections establishing)
statement latencies in milliseconds:
    0.002  \set aid random(1, 100000 * :scale)
    0.005  \set bid random(1, 1 * :scale)
```

07623615.00082-01 34 01-5

```

0.002  \set tid random(1, 10 * :scale)
0.001  \set delta random(-5000, 5000)
0.326  BEGIN;
0.603  UPDATE pgbench_accounts SET abalance =
abalance + :delta WHERE aid = :aid;
0.454  SELECT abalance FROM pgbench_accounts
WHERE aid = :aid;
5.528  UPDATE pgbench_tellers SET tbalance =
tbalance + :delta WHERE tid = :tid;
7.335  UPDATE pgbench_branches SET bbalance =
bbalance + :delta WHERE bid = :bid;
0.371  INSERT INTO pgbench_history (tid, bid,
aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta,
CURRENT_TIMESTAMP);
1.212  END;

```

Если задействуется несколько файлов скриптов, средние значения выводятся отдельно для каждого файла.

Учтите, что сбор дополнительных временных показателей влечёт некоторые издержки и приводит к снижению средней скорости и, как результат, падению TPS. На сколько именно снизится скорость, во многом зависит от платформы и оборудования. Хороший способ оценить, каковы эти издержки – сравнить средние значения TPS, получаемые с подсчётом времени операторов и без такого подсчёта.

8.10.Полезные советы

Используя `pgbench`, можно без особого труда получить абсолютно бессмысленные числа. Последуйте приведённым советам, чтобы получить полезные результаты.

Во-первых, никогда не доверяйте тестам, которые выполняются всего несколько секунд. Воспользуйтесь параметром `-t` и `-T` и установите время выполнения не меньше нескольких минут, чтобы избавиться от шума в средних значениях. В некоторых случаях для получения воспроизводимых результатов тестирование должно продолжаться несколько часов. Чтобы понять, были ли получены воспроизводимые значения, имеет смысл запустить тестирование

несколько раз.

Для стандартного сценария по типу ТРС-В начальный коэффициент масштаба (-s) должен быть не меньше числа клиентов, с каким вы намерены проводить тестирование (-c); в противном случае вы, по большому счёту, будете замерять время конкурентных изменений. Таблица `rgbench_branches` содержит всего -s строк, а каждая транзакция хочет изменить одну из них, так что если значение -c превышает -s, это несомненно приведёт к тому, что многие транзакции будут блокироваться другими.

Стандартный сценарий тестирования также довольно сильно зависит от того, сколько времени прошло с момента инициализации таблиц: накопление неактуальных строк и «мёртвого» пространства в таблицах влияет на результаты. Чтобы правильно оценить результаты, необходимо учитывать, сколько всего изменений было произведено и когда выполнялась очистка. Если же включена автоочистка, это может быть чревато непредсказуемыми изменениями оценок производительности.

Полезность результатов `rgbench` также может ограничиваться тем, что тестирование с большим числом клиентских сеансов само по себе нагружает систему. Этого можно избежать, запуская `rgbench` на другом компьютере, не на сервере баз данных, хотя при этом большое значение имеет скорость сети. Иногда, оценивая производительность одного сервера, полезно запускать даже несколько экземпляров `rgbench` параллельно, на отдельных клиентских компьютерах.

8.11.Безопасность

Если к базе данных, которая не приведена в соответствие шаблону безопасного использования схем, имеют доступ недоверенные пользователи, не запускайте `rgbench` в этой базе. Программа `rgbench` использует неполные имена и не настраивает для себя путь поиска.

9. PG_CONFIG

`pg_config` – вывести информацию об установленной версии Синергия-БД.

9.1. Синтаксис

```
pg_config [ параметр ...]
```

9.2. Описание

Приложение `pg_config` выводит информацию по конфигурационным параметрам развернутого кластера Синергия-БД. Это можно использовать, например, в программном обеспечении, которое взаимодействует с Синергия-БД для поиска необходимых файлов заголовков и библиотек.

9.3. Опции

Опции при использовании `pg_config` описаны ниже.

`--bindir`

Вывести путь расположения исполняемых файлов. Можно использовать, например, для поиска утилиты `psql`. Обычно там же находится и сама утилита `pg_config`.

`--docdir`

Вывести путь размещения файлов документации.

`--htmldir`

Вывести путь размещения файлов документации в формате HTML.

`--includedir`

Вывести путь размещения заголовочных C-файлов клиентских интерфейсов.

`--pkgincludedir`

Вывести путь размещения других заголовочных C-файлов.

`--includedir-server`

Вывести путь размещения заголовочных C-файлов для программирования серверной части.

`--libdir`

Вывести путь размещения кода объектов библиотек.

`--pkglibdir`

Вывести путь размещения динамически подгружаемых модулей, либо путь где сервер должен их искать. По этому пути также могут размещаться и другие

архитектурно-зависимые файлы.

`--localedir`

Вывести путь размещения файлов поддержки локалей. Если поддержка локалей не была сконфигурирована на этапе сборки Синергия-БД, то будет выведена пустая строка.

`--mandir`

Вывести путь размещения файлов руководства по использованию.

`--sharedir`

Вывести путь размещения архитектурно-независимых вспомогательных файлов.

`--sysconfdir`

Вывести путь размещения системных конфигурационных файлов.

`--pgxs`

Вывести путь размещения файлов сборки расширений.

`--configure`

Вывести список опций, использованных утилитой `configure` при сборке Синергия-БД. Это можно использовать, чтобы при последующей сборке сделать идентичную конфигурацию. Или для того, чтобы найти с какими опциями был собран используемый бинарный пакет. Стоит отметить, что бинарные пакеты нередко содержат патчи, специфичные для вендоров.

`--cc`

Вывести использованное при сборке Синергия-БД значение переменной `CC`. Оно отражает, какой C-компилятор применялся.

`--cppflags`

Вывести использованное при сборке Синергия-БД значение переменной `CPPFLAGS`. Оно отражает флаги C-компилятора, примененные для препроцессора. Обычно это флаги `-I`.

`--cflags`

Вывести использованное при сборке Синергия-БД значение переменной `CFLAGS`. Оно отражает флаги C-компилятора, примененные при сборке.

`--cflags_sl`

Вывести использованное при сборке Синергия-БД значение переменной `CFLAGS_SL`. Оно отражает дополнительные флаги С-компилятора для сборки разделяемых библиотек.

`--ldflags`

Вывести использованное при сборке Синергия-БД значение переменной `LDFLAGS`. Оно отражает флаги компоновщика.

`--ldflags_ex`

Вывести использованное при сборке Синергия-БД значение переменной `LDFLAGS_EX`. Оно отражает флаги компоновщика, использованные при сборке лишь исполняемых файлов.

`--ldflags_sl`

Вывести использованное при сборке Синергия-БД значение переменной `LDFLAGS_SL`. Оно отражает флаги компоновщика, использованные при сборке лишь разделяемых библиотек.

`--libs`

Вывести использованное при сборке Синергия-БД значение переменной `LIBS`. Обычно оно отражает флаги подключения внешних библиотек к Синергия-БД, переданные в опции `-l`.

`--version`

Вывести версию Синергия-БД.

`-?`

`--help`

Вывести справку по команде `pg_config`.

Если одновременно передано несколько опций, то выводимая информация будет следовать согласно порядку опций. Если опции не переданы, то будет выведена вся информация с подписями, к чему она относится.

9.4. Пример

Чтобы узнать подробнее о существующей собранной конфигурации

Синергия-БД, можно выполнить команду:

```
eval ./configure `pg_config --configure`
```

Вывод `pg_config --configure` содержит символы экранирования, поэтому значения аргументов, содержащие пробелы, представлены корректно. Таким образом, применение `eval` необходимо для получения верного результата.

10. PG_DUMP

`pg_dump` – извлечь базу данных Синергия-БД в скрипт или файл архива.

10.1.Синтаксис

```
pg_dump [ опции подключения ... ] [ параметр ... ]  
[ dbname ]
```

10.2.Описание

`pg_dump` это приложение для резервирования баз Синергия-БД. Оно создает согласованные копии, в том числе и на работающих базах данных. `pg_dump` не блокирует других пользователей базы, ни на чтение, ни на запись.

Приложение выводит данные либо в скрипты, либо в архивные форматы файлов. Скрипты представляют собой текстовые файлы, содержащие SQL-команды, необходимые для воссоздания базы данных до состояния на момент создания скрипта. Для восстановления из скрипта его содержимое можно передать утилите `psql`. Скрипты можно использовать для восстановления на других машинах, в том числе с иной архитектурой. Также скрипты с некоторыми изменениями можно использовать в других базах данных SQL.

Для восстановления из архивных форматов файлов используется утилита `pg_restore`. Эти форматы позволяют указывать `pg_restore` какие объекты базы данных восстановить, а также позволяют изменить порядок следования восстанавливаемых объектов. Архивные форматы файлов спроектированы так, чтобы их можно было переносить на другие платформы с другой архитектурой.

Применение архивных форматов в сочетании утилит `pg_restore` и `pg_dump` позволяет организовывать эффективный механизм архивации и переноса

данных. `pg_dump` можно использовать для резервирования всей базы данных, а затем при применении `pg_restore` выбрать нужные объекты для восстановления. Наиболее гибкие форматы резервных файлов это «custom» (`-Fc`) и «directory» (`-Fd`). Они позволяют выбрать и изменить порядок объектов, поддерживают восстановление в несколько потоков, а также сжимаются по умолчанию. При этом формат «directory» единственный, позволяющий выгружать данные в несколько потоков.

Во время работы `pg_dump` следует обращать внимание на предупреждения, которые печатаются в стандартный поток ошибок, особенно ввиду рассмотренных далее ограничений.

10.3.Опции

Опции командной строки для управления содержимым и форматом вывода описаны ниже.

`dbname`

Указывает имя базы данных, из которой будут выгружаться данные. Если имя не задано, то используется значение переменной окружения `PGDATABASE`. Если и переменная не задана, то в качестве имени базы будет взято имя пользователя, под которым осуществляется подключение.

`-a`

`--data-only`

Выводить только данные, но не схемы объектов (DDL). Будут копироваться данные таблиц, большие объекты, значения последовательностей.

Флаг похож на `--section=data`, но по историческим причинам не равнозначен ему.

`-b`

`--blobs`

Включить большие объекты в копию. Это поведение по умолчанию, за исключением ситуаций, когда используются флаги `--schema`, `--table`, или `--schema-only`, таким образом применение `-b` целесообразно для копий выборочных объектов.

`-B`
`--no-blobs`

Исключить из выгрузки большие объекты.

Когда задаётся и `-b`, и `-B`, большие объекты при выгрузке данных будут выводиться (см. описание ключа `-b`).

`-c`
`--clean`

Включить в выходной файл команды удаления (DROP) объектов базы данных перед командами создания (CREATE) этих объектов. Если дополнительно не указать флаг `--if-exists`, то при восстановлении в базу данных, где некоторые объекты отсутствуют, попытка удаления несуществующего объекта будет приводить к ошибке, которую можно игнорировать.

Опция полезна только для вывода файлов в формате `plain`. Для архивных форматов, эту опцию можно указывать при вызове `pg_restore`.

`-C`
`--create`

Сформировать вначале вывода команду создания базы данных и дальнейшим подключением к ней. В этом случае неважно какая база указана в параметрах подключения перед выполнением скрипта. Также, если указана опция `--clean`, то скрипт сначала удалит, а затем пересоздаст базу данных перед подключением к ней.

С ключом `--create` в выходной файл также включается комментарий к базе данных (если он задан) и все назначения переменных конфигурации, связанные с базой данных, то есть все команды `ALTER DATABASE ... SET ...` и `ALTER ROLE ... IN DATABASE ... SET ...`, ссылающиеся на эту базу данных. Также выгружаются права доступа к самой базе данных, если не добавлен ключ `--no-acl`.

Опция полезна только для вывода файлов в формате `plain`. Для архивных форматов, эту опцию можно указывать при вызове `pg_restore`.

`-E` кодировка
`--encoding=кодировка`

Создать копию в заданной кодировке. По умолчанию копия создается в

кодировке, используемой базой данных. Другой способ достичь того же результата это установить желаемую кодировку в переменную окружения `PGCLIENTENCODING`.

```
-f файл  
--file=файл
```

Отправить вывод в указанный файл. Параметр можно не указывать, если используется формат с выводом в файл. В этом случае будет использован стандартный вывод. Однако для формата с выводом в директорию параметр является обязательным и должен указывать на путь к директории. В таком случае директория будет создана командой `pg_dump` и не должна существовать заранее.

```
-F format  
--format=format
```

Указывает формат вывода копии. `format` может принимать следующие значения:

```
p  
plain
```

Сформировать текстовый SQL скрипт. Это поведение по умолчанию.

```
c  
custom
```

Вывести копию в архивном формате, пригодном для дальнейшего использования утилитой `pg_restore`. Наравне с `directory` является наиболее гибким форматом, позволяющим вручную выбирать и сортировать восстанавливаемые объекты. Вывод в этом формате по умолчанию сжимается.

```
d  
directory
```

Выгрузить в формате директории. Этот формат пригоден для дальнейшего использования утилитой `pg_restore`. При этом будет создана директория, в которой для каждой таблицы и большого объекта будут созданы отдельные файлы, а также файл оглавления в машинно-читаемом формате, понятном для `pg_restore`. С полученной резервной копией можно работать штатными средствами Unix, например, несжатую копию можно сжать посредством `gzip`. Этот формат по умолчанию сжимается, а также поддерживает работу в несколько потоков.

t
tar

Вывести копию в формате tar, для дальнейшего использования утилитой pg_restore. Этот формат совместим с форматом вывода в директорию. При распаковке полученной копии на выходе получается структура формата вывода в директорию. При этом не поддерживается сжатие и есть ограничение на размер отдельной таблицы в 8 Гб. Также нет возможности регулировать порядок восстановления таблиц из копии.

-i
--ignore-version

Устаревший флаг и сейчас игнорируется.

-j njobs
--jobs=njobs

Запустить выгрузку в несколько потоков, которые смогут параллельно обрабатывать njobs таблиц. Это уменьшает время работы, но увеличивает нагрузку на сервер. Опцию можно использовать только с форматом вывода directory, т.к. это единственный формат, позволяющий нескольким процессам одновременно записывать данные.

pg_dump откроет njobs + 1 соединений с базой данных. Таким образом необходимо обеспечить достаточное значение параметра max_connections.

Если во время выгрузки в несколько потоков, параллельно работающие сессии будут запрашивать эксклюзивные блокировки на объекты базы данных, то pg_dump может завершиться аварийно. Дело в том, что головной процесс pg_dump вначале запрашивает разделяемые блокировки на объекты, которые позже будут выгружать рабочие процессы. Это делается для того, чтобы никто не смог удалить объекты на время работы pg_dump. Если же другая сессия запросит эксклюзивную блокировку на объект, то запрос на блокировку будет поставлен в очередь, до тех пор пока разделяемая блокировка головного процесса pg_dump не будет снята. В последующем, любая попытка доступа к этому объекту будет вставать в очередь, вслед за эксклюзивной блокировкой. В том числе в очередь попадет и рабочий

процесс `pg_dump`. Если не принять меры предосторожности, то получим классическую взаимоблокировку. Для предупреждения подобных конфликтов, рабочий процесс `pg_dump` ещё раз запрашивает разделяемую блокировку на объект с опцией `NOWAIT`. И если он не смог получить блокировку, значит кто-то ещё запросил эксклюзивную блокировку объекта. А это значит, что нет возможности продолжить выгрузку, поэтому `pg_dump` прерывает дальнейшую работу.

Для получения целостной резервной копии серверу баз данных необходимо поддерживать функциональность синхронизированных снимков. Это позволяет разным клиентам работать с одной и той же версией данных, несмотря на использование разных подключений. `pg_dump -j` использует множественные подключения. Первое подключение осуществляется головным процессом, а последующие – рабочими процессами. Без функциональности синхронизируемых снимков нет гарантии того, что каждое подключение увидит одни и те же данные, что может привести к несогласованности данных резервной копии.

Если необходимо выполнить выгрузку в несколько потоков на сервере, необходимо быть уверенным, что база данных не будет изменяться с момента подключения головного процесса и до момента, когда последний рабочий процесс подключится к базе данных. Для этого, проще всего перед запуском `pg_dump` остановить все процессы, модифицирующие данные (DML и DDL).

```
-n схема  
--schema=схема
```

Выгрузить только схемы, соответствующие шаблону `схема`; вместе с этими схемами будут выгружены и все содержащиеся в них объекты. Когда этот параметр отсутствует, выгружаются все несистемные схемы в целевой базе данных. Чтобы выгрузить несколько схем, ключ `-n` можно указать несколько раз. Кроме того, параметр `схема` интерпретируется как шаблон по тем же правилам, что и с командами `psql \d`, так что несколько схем можно выбрать и шаблоном со знаками подстановки. Используя знаки подстановки, при необходимости заключайте шаблон в кавычки, чтобы эти знаки не разворачивала оболочка системы.

Примечания:

1. При использовании `-n`, `pg_dump` не выгружает объекты других схем, от которых выгружаемая схема может зависеть. Таким образом не гарантируется, что выгруженная схема будет успешно восстановлена на чистой базе данных.

2. Не принадлежащие схемам объекты (например, большие бинарные объекты), не выгружаются с опцией `-n`. Однако можно указать `--blobs` чтобы они попали в выгрузку.

`-N` схема

`--exclude-schema=схема`

Не выгружать схемы, соответствующие шаблону `схема.Шаблон`. Шаблон интерпретируется по тем же правилам, что и для опции `-n`. Опцию `-N` можно использовать в команде несколько раз для исключения схем, соответствующих нескольким шаблонам.

При одновременном использовании опций `-n` и `-N` будут выгружаться схемы, соответствующие шаблону опции `-n` и не противоречащие шаблону опции `-N`.

`-o`

`--oids`

Выгружать идентификаторы объектов (OIDs) вместе с данными таблиц. Используйте эту опцию, если в приложении используются есть ссылки на OID, например во внешних ключах. В противном случае, опцию лучше не использовать.

`-O`

`--no-owner`

Не формировать команды, устанавливающие владельца объектов базы данных. По умолчанию `pg_dump` генерирует команды `ALTER OWNER` или `SET SESSION AUTHORIZATION` для назначения владельцев объектов базы. Эти команды завершатся неудачно, если скрипт будет запущен не суперпользователем или не владельцем объектов. Чтобы создать скрипт, который можно выполнить при восстановлении от лица произвольного пользователя и назначить его в качестве владельца объектов восстанавливаемой базы, необходимо указать флаг `-O`.

Опция полезна только для вывода файлов в формате `plain`. Для архивных форматов, эту опцию можно указывать при вызове `pg_restore`.

```
-R  
--no-reconnect
```

Параметр является устаревшим, но в целях совместимости ещё работает.

```
-s  
--schema-only
```

Выгружать только определения объектов (схемы), без данных.

Действие опции противоположно действию опции `--data-only`. Это похоже на указание `--section=pre-data` `--section=post-data`, но по историческим причинам не равнозначно ему.

Чтобы не выгружать данные отдельных таблиц, используйте опцию `--exclude-table-data`.

```
-S username  
--superuser=username
```

Указать суперпользователя, который будет использоваться для отключения триггеров. Опция имеет значение только вместе с `--disable-triggers`. Обычно её лучше не использовать, а запускать полученный скрипт от имени суперпользователя.

```
-t таблица  
--table=таблица
```

Выгрузить только таблицы, соответствующие шаблону `таблица`. В этом контексте под «таблицей» подразумеваются также представления, материализованные представления, последовательности и сторонние таблицы. Чтобы выбрать несколько таблиц, ключ `-t` можно указать несколько раз. Кроме того, параметр `таблица` интерпретируется как шаблон по тем же правилам, что и с командами `psql \d`, так что несколько таблиц можно выбрать и с шаблоном со знаками подстановки. Используя знаки подстановки, при необходимости заключайте шаблон в кавычки, чтобы эти знаки не разворачивала оболочка системы.

Параметры `-n` и `-N` не действуют в присутствии параметра `-t`, так как отобранные им таблицы всё равно будут выгружены, а не табличные объекты выгружаться не будут.

Примечание. При использовании `-t`, `pg_dump` не выгружает прочие объекты,

от которых выгружаемые таблицы могут зависеть. Таким образом не гарантируется, что выгруженные таблицы будут успешно восстановлены на чистой базе данных.

```
-T таблица  
--exclude-table=таблица
```

Не выгружать таблицы, соответствующие шаблону таблица. Шаблон интерпретируется по тем же правилам, что и для опции `-t`. Опцию `-T` можно использовать в команде несколько раз для исключения таблиц, соответствующих нескольким шаблонам.

При одновременном использовании опций `-t` и `-T` будут выгружаться таблицы, соответствующие шаблону опции `-t` и не противоречащие шаблону опции `-T`.

```
-v  
--verbose
```

Включить подробный режим. `pg_dump` будет выводить в стандартный поток ошибок подробные комментарии к объектам, включая время начала и окончания выгрузки, а также сообщения о прогрессе выполнения.

```
-V  
--version
```

Вывести версию `pg_dump`.

```
-x  
--no-privileges  
--no-acl
```

Не выгружать права доступа (команды `GRANT/REVOKE`).

```
-Z 0..9  
--compress=0..9
```

Установить уровень сжатия данных. Ноль означает, что сжатие выключено. Для архивного формата `custom` будут сжиматься файлы отдельных таблиц. По умолчанию применяется умеренный уровень сжатия. Если указать отличный от нулевого уровень сжатия для формата `plain`, то сжиматься будет весь выходной файл, как это было бы при передаче файла команде `gzip`. Но по умолчанию для `plain` сжатие данных отключено. Формат `tar` в настоящий момент не поддерживает сжатие.

`--binary-upgrade`

Предназначена для утилит обновления сервера. Использование для иных целей не рекомендуется и не поддерживается. Поведение опции может быть изменено в последующих версиях без предварительного уведомления.

`--column-inserts`

`--attribute-inserts`

Выгружать данные таблиц в виде команд `INSERT` с явным указанием столбцов (`INSERT INTO table (column, ...) VALUES ...`). В этом случае, скорость восстановления значительно снизится. Полезно в тех случаях, когда загружать данные нужно не в Синергия-БД. Также, поскольку для каждой строки генерируется отдельная команда, сбой при последующей загрузке приведет к потере конкретной строки, а не всей таблицы.

`--disable-dollar-quoting`

Выключает экранирование тел функций с помощью знака доллара и принуждает к экранированию посредством стандартного синтаксиса SQL.

`--disable-triggers`

Используется при выгрузке одних данных (с опцией `--data-only`). Заставляет `pg_dump` включать в вывод команды для временного выключения триггеров при восстановлении в целевой базе данных. Применяется в ситуациях, когда существуют проверки ссылочной целостности или другие триггеры, которые необходимо выключить на время восстановления.

В настоящее время команды, генерируемые опцией `--disable-triggers`, должны исполняться от имени суперпользователя. Таким образом, необходимо также передавать флаг `-S`, либо при восстановлении выполнять скрипт от имени суперпользователя.

Опция полезна только для вывода файлов в формате `plain`. Для архивных форматов, эту опцию можно указывать при вызове `pg_restore`.

`--enable-row-security`

Этот параметр имеет смысл только при выгрузке содержимого таблицы, для которой включена защита строк. По умолчанию `pg_dump` устанавливает для

`row_security` значение `off`, чтобы убедиться, что выгружаются все данные из таблицы. Если пользователь не имеет достаточных прав для обхода защиты строк, выдаётся ошибка. Этот параметр указывает `pg_dump` включить `row_security`, что позволит пользователю выгрузить часть содержимого таблицы, к которой он имеет доступ.

Заметьте, что в настоящее время для использования этого параметра обычно желательно, чтобы данные были выгружены в формате `INSERT`, так как команда `COPY FROM` в процессе восстановления не поддерживает защиту строк.

`--exclude-table-data=таблица`

Не выгружать содержимое таблиц, соответствующих шаблону `таблица`. Шаблон таблицы интерпретируется по тем же правилам, что и в опции `-t`. Опцию `--exclude-table-data` можно использовать в команде несколько раз для исключения таблиц, соответствующих нескольким шаблонам. Полезно, когда нужно получить определение таблицы, без содержимого.

Чтобы не выгружать содержимое всех таблиц базы, используйте опцию `--schema-only`.

`--if-exists`

При очистке целевой базы использовать условные команды (добавлять фразу `IF EXISTS`). Применяется только с опцией `--clean`.

`--inserts`

Выгружать данные таблиц в виде команд `INSERT` вместо `COPY`. В этом случае, скорость восстановления значительно снизится. Полезно в тех случаях, когда загружать данные нужно не в Синергия-БД. Также, поскольку для каждой строки генерируется отдельная команда, сбой при последующей загрузке приведет к потере конкретной строки, а не всей таблицы. Обратите внимание, что последующая загрузка может завершиться неудачно, если у таблицы изменен порядок столбцов. Можно использовать опцию `--column-inserts`, для которой порядок столбцов не важен, но она работает ещё медленнее.

`--load-via-partition-root`

При выгрузке данных для секции таблицы выводить команды `COPY` или

INSERT, ссылающиеся на корневую таблицу в иерархии секционирования, а не на эту секцию. В результате при загрузке данных подходящая секция будет выбираться заново для каждой строки. Это может быть полезно при перезагрузке данных, когда на целевом сервере строки не всегда попадают в те же секции, в которых они находились на исходном. Это возможно, например, когда столбец разбиения имеет текстовый тип и в двух системах по-разному определено правило сортировки, по которому упорядочивается этот столбец.

При восстановлении из архива с этим параметром лучше не использовать распараллеливание, так как `pg_restore` не будет иметь представления, в какие секции попадут данные. Это может повлечь снижение производительности из-за конфликтов блокировки между параллельными заданиями или даже сбой из-за того, что ограничения внешнего ключа вступят в силу прежде, чем будут загружены все нужные данные.

`--lock-wait-timeout=время ожидания`

Опция позволяет отказаться от бесконечного ожидания при получении разделяемых блокировок таблиц вначале работы. Вместо этого, если было превышено время ожидания при получении блокировки, команда аварийно завершается. Время можно задать в любом формате, который поддерживает команда `SET statement_timeout`.

`--no-comments`

Не выгружать комментарии.

`--no-publications`

Не выгружать публикации.

`--no-security-labels`

Не выгружать метки безопасности.

`--no-subscriptions`

Не выгружать подписки.

`--no-sync`

По умолчанию `pg_dump` ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром `pg_dump` завершается немедленно, то есть

выполняется быстрее, но в случае неожиданного сбоя операционной системы выгруженные данные могут оказаться испорченными. Вообще этот параметр предназначен прежде всего для тестирования, для производственной среды он не подходит.

`--no-tablespaces`

Не формировать команды для указания табличных пространств. Все объекты будут создаваться в табличном пространстве по умолчанию.

Опция полезна только для вывода файлов в формате `plain`. Для архивных форматов, эту опцию можно указывать при вызове `pg_restore`.

`--no-unlogged-table-data`

Не выгружать данные нежурналируемых таблиц. Опция не влияет на выгрузку определения таблиц, она только подавляет вывод содержимого таблиц. При запуске на резервном сервере, содержимое нежурналируемых таблиц никогда не выгружается.

`--quote-all-identifiers`

Принудительно экранировать все идентификаторы. Может быть полезно при миграции на новую версию сервера, для избежания конфликтов с новыми резервными словами.

`--section=имя секции`

Выгружать лишь указанную секцию. Имя секции может принимать значения `pre-data`, `data` или `post-data`. Для выгрузки нескольких секций, опцию можно использовать несколько раз в одной команде. По умолчанию резервируются все секции.

Секция `data` содержит непосредственно данные таблиц, больших объектов и значения последовательностей. Секция `post-data` содержит определения индексов, триггеров, правил и ограничений (кроме ограничений проверки, созданных без `NOT VALID`). Секция `pre-data` включает определения остальных элементов.

`--serializable-deferrable`

Использовать при выгрузке транзакцию с уровнем изоляции

serializable, для гарантирования целостного снимка данных. Правда для этого нужно выждать момент, когда в потоке транзакций нет аномалий, и поэтому нет риска, что выгрузка завершится неудачно и риска отката других транзакций с ошибкой `serialization_failure`.

Опция не особо полезна в случаях, когда требуется восстановление после сбоя. Она полезна для создания копии базы данных, на которой формируются отчеты и выполняются другие операция чтения, в то время как на основной базе продолжается обычная работа. Без этой опции, выгрузка может содержать не целостное состояние базы данных. Например, если используется пакетная обработка, статус пакета может отражаться как завершенный, в то время как в выгрузке будут не все элементы пакета.

Опция не будет влиять на результат, если во время запуска `pg_dump` нет активных транзакций на чтение-запись. Если же активные транзакции чтения-записи есть, то начало выгрузки может быть отложено на неопределенный период времени. После того выгрузка началась производительность с опцией или без нее будет одинаковой.

```
--snapshot=имя_снимка
```

Использовать заданный синхронный снимок при выгрузке данных из базы.

В случае с параллельной выгрузкой будет использоваться имя снимка, определённое этим параметром; новый снимок не будет сделан.

```
--strict-names
```

Требует, чтобы каждому указанию схемы (`-n/--schema`) и таблицы (`-t/--table`) соответствовала минимум одна схема/таблица в выгружаемой базе данных. Заметьте, что если не находится вообще ни одной схемы/таблицы для заданных шаблонов, `pg_dump` выдаёт ошибку и без ключа `--strict-names`.

Этот параметр не действует на ключи `-N/--exclude-schema`, `-T/--exclude-table` или `--exclude-table-data`. Если не находятся объекты, соответствующие шаблонам исключения, это не считается ошибкой.

```
--use-set-session-authorization
```

Выводить команды стандарта SQL `SET SESSION AUTHORIZATION` вместо

ALTER OWNER, для назначения владельцев объектов базы данных. Это больше соответствует стандартам, но в зависимости от истории объектов попавших в выгрузку, существует вероятность неверного восстановления. Кроме того, использование SET SESSION AUTHORIZATION требует прав суперпользователя при восстановлении, в то время, как ALTER OWNER требует меньших привилегий.

-?

--help

Вывести справку по команде pg_dump.

Далее описаны параметры управления подключением:

-d dbname

--dbname=dbname

Указывает имя базы данных для подключения. Эквивалентно указанию dbname в качестве первого аргумента, не являющегося параметром в командной строке.

Если параметр содержит символ =, или начинается с допустимого префикса URI, например, postgresql:// или postgres://, то он трактуется аналогично строке подключения conninfo.

-h host

--host=host

Указывает имя компьютера, на котором запущен работающий сервер баз данных. Если значение начинается с косой черты, то оно трактуется как название директории, где хранится файл Unix-сокета. Значение по умолчанию берется из переменной окружения PGHOST, если она установлена. Также производится попытка подключения по Unix-сокету.

-p port

--port=port

Указывает TCP-порт или расширение локального файла Unix-сокета, на котором сервер слушает подключения. По умолчанию берется значение переменной окружения PGPORT, если оно установлено, либо значение времени компиляции.

-U username

--username=username

Имя пользователя, под которым производится подключение.

```
-w  
--no-password
```

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Опция может быть полезна в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

```
-W  
--password
```

Принудить `pg_dump` к запросу пароля перед подключением к серверу баз данных.

Опция несущественна, т.к. при включенной на сервере аутентификации по паролю, `pg_dump` запросит его автоматически. Однако `pg_dump` сделает лишнее подключение для определения необходимости ввода пароля. Поэтому во избежании лишнего запроса в этой ситуации полезно использовать флаг `-W`.

```
--role=имя роли
```

Указывает роль, под которой будет выполняться выгрузка. `pg_dump` выполнит команду `SET ROLE имя роли` после установления соединения. Это полезно, когда пользователь, под которым авторизовались (указанный в `-U`), не имеет достаточных привилегий для выгрузки, но может переключиться на роль, с достаточными полномочиями. Если в организации действует политика, запрещающая подключение напрямую под суперпользователем, то с этой опцией можно выполнить выгрузку, не нарушая политику.

10.4.Переменные окружения

```
PGDATABASE  
PGHOST  
PGOPTIONS  
PGPORT  
PGUSER
```

Параметры подключения по умолчанию.

Эта утилита, как и большинство других утилит Синергия-БД, также

использует переменные среды, поддерживаемые `libpq`.

10.5.Диагностика

`pg_dump` на низком уровне выполняет команды `SELECT`. Если есть проблемы с работой `pg_dump`, убедитесь, что в базе данных можно выполнить `SELECT`, например из `psql`. Также следует учитывать, что будут применяться любые параметры подключения по умолчанию и значения переменных окружения используемых в `libpq`.

Обычно действия `pg_dump` в базе данных собираются коллектором статистики. Если это нежелательно, то можно установить параметр `track_counts` в `false` в переменной окружения `PGOPTIONS` или в команде `ALTER USER`.

10.6.Замечания

Если в базу данных кластера `template1` устанавливались дополнительные объекты, то следует убедиться, что выгрузка `pg_dump` загружается в пустую базу данных. Иначе существует вероятность возникновения ошибок дублирования создаваемых объектов. Чтобы создать пустую базу данных, копируйте её из шаблона `template0`, вместо `template1`, например:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

Если выгружаются только данные с одновременным использованием `--disable-triggers`, `pg_dump` сформирует команды для выключения табличных триггеров перед вставкой данных, а после них - команды, включающие триггеры обратно. Если восстановление будет прервано в середине процесса, системный каталог может остаться в неверном состоянии.

Сформированный `pg_dump` файл не содержит данных статистики, используемых планировщиком. Поэтому после восстановления следует выполнить `ANALYZE` для достижения оптимальной производительности. Также не выгружаются команды `ALTER DATABASE ... SET`. Эти установки, вместе с данными о ролях и прочими настройками, выгружаются с помощью `pg_dumpall`.

Так как `pg_dump` применяется для переноса данных в новые версии

Синергия-БД, предполагается, что вывод `pg_dump` можно загрузить на сервер Синергия-БД более новой версии, чем версия `pg_dump`. `pg_dump` может также выгружать данные серверов Синергия-БД более старых версий, чем его собственная.

Выгружая подписки на логическую репликацию, `pg_dump` будет выдавать команды `CREATE SUBSCRIPTION` с указанием `connect = false`, так что при восстановлении подписки не будут устанавливаться удалённые подключения для создания слота репликации или для начального копирования таблиц. Таким образом, выгруженные данные могут быть восстановлены без сетевого доступа к удалённым серверам. Вновь активировать подписки должным образом – задача пользователя. Если задействованные серверы поменялись, возможно, придётся скорректировать строки подключения. Также может быть уместно опустошить целевые таблицы перед началом полного копирования таблиц.

10.7.Примеры

Чтобы выгрузить базу данных `mydb` в файл SQL-скрипта:

```
$ pg_dump mydb > db.sql
```

Для восстановления из выше полученного скрипта в чистую базу `newdb`:

```
$ psql -d newdb -f db.sql
```

Восстановление архива в ту же базу данных, из которой он был выгружен, с предварительным удалением текущего содержимого этой базы данных:

```
$pg_restore -d postgres --clean --create db.dump
```

Чтобы сформировать выгрузку в формате `custom`:

```
$ pg_dump -Fc mydb > db.dump
```

Чтобы сформировать выгрузку в формате `directory`:

```
$ pg_dump -Fd mydb -f dumpdir
```

Чтобы сформировать выгрузку в формате `directory` в 5 параллельных потоков:

```
$ pg_dump -Fd mydb -j 5 -f dumpdir
```

Для восстановления в чистую базу данных `newdb`:

```
$ pg_restore -d newdb db.dump
```

Чтобы выгрузить отдельную таблицу mytab:

```
$ pg_dump -t mytab mydb > db.sql
```

Чтобы выгрузить все таблицы, имена которых начинаются на emp и принадлежат схеме detroit, исключая таблицу employee_log:

```
$ pg_dump -t 'detroit.emp*' -T detroit.employee_log  
mydb > db.sql
```

Чтобы выгрузить все схемы, имена которых начинаются на east или west и заканчиваются на gsm, исключая те, имена которых содержат test:

```
$ pg_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb >  
db.sql
```

То же самое, но с использованием регулярного выражения:

```
$ pg_dump -n '(east|west)*gsm' -N '*test*' mydb > db.sql
```

Чтобы выгрузить все объекты базы данных, исключая таблицы, чьи имена начинаются с ts_:

```
$ pg_dump -T 'ts_*' mydb > db.sql
```

Чтобы указать имя в верхнем или смешанном регистре в опции -t и ей подобных, его необходимо экранировать двойными кавычками, иначе оно автоматически приводится к нижнему регистру. Однако стоит помнить, что двойные кавычки являются спецсимволом для командной оболочки, потому их также необходимо экранировать. Так, чтобы выгрузить таблицу с именем в смешанном регистре, нужно что-то подобное:

```
$ pg_dump -t «\»MixedCaseName\»« mydb > mytab.sql
```

11. PG_DUMPALL

pg_dumpall – выгрузить кластер баз данных Синергия-БД в формате скрипта.

11.1.Синтаксис

```
pg_dumpall [ опции подключения ... ] [ параметр ... ]
```

11.2.Описание

Утилита pg_dumpall предназначена для записи («выгрузки») всех баз данных кластера Синергия-БД в один файл в формате скрипта. Этот файл содержит

команды SQL, так что передав его на вход `psql`, можно восстановить все базы данных. Чтобы его сформировать, `pg_dumpall` вызывает для каждой базы данных в кластере `pg_dump` и дополнительно выгружает глобальные объекты, общие для всех баз данных. Утилита `pg_dump` не сохраняет эти объекты. В настоящее время, к таким объектам относятся группы, пользователи и табличные пространства, а также такие свойства, как права доступа, которые применяются к базам данных в целом.

Так как утилита `pg_dumpall` читает таблицы из всех баз данных, для получения полного содержимого баз запускать ее, как правило, нужно от имени суперпользователя. Чтобы впоследствии выполнить сохраненный скрипт, также необходимо иметь права суперпользователя, позволяющие создавать пользователей, группы и собственно базы данных.

Генерируемый SQL-скрипт записывается в стандартное устройство вывода. Чтобы перенаправить его в файл, воспользуйтесь параметром `[-f|file]` или операторами оболочки.

Утилите `pg_dumpall` требуется подключаться к серверу Синергия-БД несколько раз (к каждой базе по отдельности). Если вы проходите проверку подлинности по паролю, вам придется каждый раз вводить пароль. Чтобы избежать этого, удобно иметь файл `~/ .pgpass`.

11.3.Параметры

Опции командной строки для управления содержимым и форматом вывода описаны ниже.

`-a`
`--data-only`

Выгружать только данные, без схемы (определений данных).

`-c`
`--clean`

Добавить команды SQL для удаления (DROP) баз данных перед командами, создающими их. В дополнение к ним добавляются команды DROP для ролей и табличных пространств.

`-E` кодировка

```
--encoding=кодировка
```

Создать копию в заданной кодировке. По умолчанию копия создаётся в кодировке базы данных. Другой способ достичь того же результата – задать желаемую кодировку в переменной окружения PGCLIENTENCODING.

```
-f имя_файла
--file=имя_файла
```

Направить вывод в указанный файл. Если этот параметр опущен, скрипт записывается в стандартный вывод.

```
-g
--globals-only
```

Выгружать только глобальные объекты (роли и табличные пространства), без баз данных.

```
-O
--oids
```

Выгружать идентификаторы объектов (OIDs) вместе с данными таблиц. Используйте эту опцию, если в приложении используются есть ссылки на OID, например во внешних ключах. В противном случае, опцию лучше не использовать.

```
-O
--no-owner
```

Не генерировать команды, устанавливающие владение объектами, как в исходной базе данных. По умолчанию, pg_dumpall генерирует команды ALTER OWNER или SET SESSION AUTHORIZATION, восстанавливающие исходных владельцев для создаваемых элементов схемы. Однако выполнить эти команды сможет только суперпользователь (или пользователь, владеющий всеми объектами, создаваемыми скриптом). Чтобы получить скрипт, который сможет восстановить любой пользователь (но при этом он станет владельцем всех объектов), используется -O.

```
-r
--roles-only
```

Выгружать только роли, без баз данных и табличных пространств.

`-s`
`--schema-only`

Выгружать только определения объектов (схемы), без данных.

`-S username`
`--superuser=username`

Указать суперпользователя, который будет использоваться для отключения триггеров. Опция имеет значение только вместе с `--disable-triggers`. Обычно её лучше не использовать, а запускать полученный скрипт от имени суперпользователя.

`-t`
`--tablespaces-only`

Выгружать только табличные пространства, без баз данных и ролей.

`-v`
`--verbose`

Включить режим подробных сообщений. В этом режиме `pg_dumpall` записывает в выходной файл время начала/завершения выгрузки, а в стандартный канал ошибок – сообщения о процессе. При этом подробные сообщения будет также выводить `pg_dump`.

`-V`
`--version`

Сообщить версию `pg_dumpall` и завершиться.

`-x`
`--no-privileges`
`--no-acl`

Не выгружать права доступа (команды `GRANT/REVOKE`).

`--binary-upgrade`

Предназначена для утилит обновления сервера. Использование для иных целей не рекомендуется и не поддерживается. Поведение опции может быть изменено в последующих версиях без предварительного уведомления.

`--column-inserts`
`--attribute-inserts`

Выгружать данные в виде команд `INSERT` с явно задаваемыми именами колонок (`INSERT INTO таблица (колонка, ...) VALUES ...`). При этом

восстановление будет очень медленным; в основном это применяется для выгрузки данных, которые затем будут загружаться не в Синергия-БД.

`--disable-dollar-quoting`

Выключает экранирование тел функций с помощью знака доллара и принуждает к экранированию посредством стандартного синтаксиса SQL.

`--disable-triggers`

Этот параметр действует только при выгрузке одних данных. С ним `pg_dumpall` добавляет команды, отключающие триггеры в целевых таблицах на время загрузки данных. Используйте его, если в ваших таблицах определены проверки ссылочной целостности или другие триггеры, которые вы не хотели бы выполнять в процессе загрузки данных.

В настоящее время команды, генерируемые опцией `--disable-triggers`, должны исполняться от имени суперпользователя. Таким образом, необходимо также передавать флаг `-S`, либо при восстановлении выполнять скрипт от имени суперпользователя.

`--if-exists`

Использовать условные команды (т.е. добавлять предложение `IF EXISTS`) при очистке базы данных и других объектов. Этот параметр принимается, только если также указан параметр `--clean`.

`--inserts`

Выгружать данные в виде команд `INSERT`, а не `COPY`. При этом восстановление значительно замедлится; в основном это применяется для выгрузки данных, которые затем будут загружаться не в Синергия-БД. Заметьте, что восстановление может вовсе не выполниться при изменении порядка колонок в таблицах. В этом смысле параметр `--column-inserts` безопаснее, но восстановление будет ещё медленнее.

`--load-via-partition-root`

При выгрузке данных для секции таблицы выводить команды `COPY` или `INSERT`, ссылающиеся на корневую таблицу в иерархии секционирования, а не на эту секцию. В результате при загрузке данных подходящая секция будет выбираться

заново для каждой строки. Это может быть полезно при перезагрузке данных, когда на целевом сервере строки не всегда попадают в те же секции, в которых они находились на исходном. Это возможно, например, когда столбец разбиения имеет текстовый тип и в двух системах по-разному определено правило сортировки, по которому упорядочивается этот столбец.

`--lock-wait-timeout=время_ожидания`

Не ждать бесконечно получения разделяемых блокировок таблиц в начале процедуры выгрузки. Вместо этого, выдать ошибку, если не удастся заблокировать таблицы за указанное время_ожидания. Это время можно задать в любом из форматов, принимаемых командой `SET statement_timeout`.

`--no-comments`

Не выгружать комментарии.

`--no-publications`

Не выгружать публикации.

`--no-role-passwords`

Не выгружать пароли ролей. При восстановлении все роли получают пароль NULL и не смогут пройти проверку подлинности, пока им не будут назначены пароли. Так как значения паролей не нужны, когда используется это указание, информация о ролях считывается из системного представления `pg_roles`, а не из `pg_authid`. Таким образом, данный вариант может быть также полезен, если доступ к `pg_authid` ограничен политикой безопасности.

`--no-security-labels`

Не выгружать метки безопасности.

`--no-subscriptions`

Не выгружать подписки.

`--no-sync`

По умолчанию `pg_dumpall` ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром `pg_dumpall` завершается немедленно, то есть выполняется быстрее, но в случае неожиданного сбоя операционной системы выгруженные данные могут оказаться испорченными. Вообще этот параметр

предназначен прежде всего для тестирования, для производственной среды он не подходит.

`--no-tablespaces`

Не выводить команды, создающие или выбирающие табличные пространства для объектов. С этим параметром все объекты будут созданы в пространстве по умолчанию, установленном во время восстановления.

`--no-unlogged-table-data`

Не выгружать содержимое нежурналируемых таблиц. Этот параметр не влияет на то, как выгружаются определения этих таблиц (схема); он отключает только выгрузку данных из них.

`--quote-all-identifiers`

Принудительно экранировать все идентификаторы. Может быть полезно при миграции на новую версию сервера, для избежания конфликтов с новыми резервными словами.

`--use-set-session-authorization`

Выводить команды `SET SESSION AUTHORIZATION`, соответствующие стандарту, вместо `ALTER OWNER`, для назначения владельцев объектов. В результате выгруженный скрипт будет более стандартизированным, но может не восстановиться корректно, в зависимости от истории объектов.

`-?`

`--help`

Показать справку по аргументам командной строки `pg_dumpall` и завершиться.

Далее описаны параметры управления подключением.

`-d` строка подключения

`--dbname=строка подключения`

Указывает параметры подключения к серверу в формате строки подключения.

Этот параметр называется `--dbname` для согласованности с другими клиентскими приложениями, но так как `pg_dumpall` подключается не к одной базе данных, имя базы в строке подключения игнорируется. Чтобы указать имя базы данных, через подключение к которой будут выгружаться глобальные объекты и

находиться другие выгружаемые базы, воспользуйтесь параметром `-l`.

```
-h host  
--host=host
```

Указывает имя компьютера, на котором запущен сервер баз данных. Если значение начинается с косой черты, оно интерпретируется как имя каталога с доменным сокетом Unix. Значение по умолчанию берется из переменной окружения `PGHOST`, если она установлена. В противном случае выполняется подключение к доменному сокету.

```
-l dbname  
--database=dbname
```

Задаёт имя базы данных, через подключение к которой будут выгружаться глобальные объекты и находиться другие выгружаемые базы. По умолчанию используется `postgres`, а в случае её отсутствия – `template1`.

```
-p port  
--port=port
```

Указывает TCP-порт или расширение локального файла Unix-сокета, на котором сервер слушает подключения. По умолчанию берется значение переменной окружения `PGPORT`, если оно установлено, либо значение времени компиляции.

```
-U username  
--username=username
```

Имя пользователя, под которым производится подключение.

```
-w  
--no-password
```

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Опция может быть полезна в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

```
-W  
--password
```

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как `pg_dumpall` запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако, чтобы

понять это, `pg_dumpall` лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Заметьте, что пароль будет запрашиваться повторно для выгрузки каждой базы данных. Поэтому обычно лучше настроить файл `~/.pgpass`, и не вводить пароль каждый раз вручную.

```
--role=имя роли
```

Задаёт имя роли, которая будет осуществлять выгрузку. Получив это имя, `pg_dumpall` выполнит `SET ROLE имя_роли` после подключения к базе данных. Это полезно, когда проходящий проверку пользователь (указанный в `-U`) не имеет прав, необходимых для `pg_dumpall`, но может переключиться на роль, наделённую этими правами. В некоторых окружениях правила запрещают подключаться к серверу непосредственно суперпользователю, и этот параметр позволяет выполнить выгрузку, не нарушая их.

11.4.Переменные окружения

```
PGHOST  
PGOPTIONS  
PGPORT  
PGUSER
```

Параметры подключения по умолчанию.

Эта утилита, как и большинство других утилит Синергия-БД, также использует переменные среды, поддерживаемые `libpq`.

11.5.Замечания

Так как `pg_dumpall` внутри себя вызывает `pg_dump`, часть диагностических сообщений будет относиться к `pg_dump`.

Ключ `--clean` может быть полезен, даже если вы намереваетесь восстановить копию из скрипта в новом кластере. С `--clean` скрипт сможет удалить и пересоздать встроенные базы данных `postgres` и `template1`, так что они получат свойства, которые имели одноимённые базы в исходном кластере (например, локаль и кодировку). Без данного ключа эти базы сохраняют свои свойства

уровня базы данных, а также всё существующее содержимое.

После восстановления имеет смысл запустить ANALYZE для каждой базы данных, чтобы оптимизатор получил актуальную статистику. Также можно запустить анализ для всех баз данных, выполнив команду `vacuumdb -a -z`.

Не следует ожидать, что скрипт выгрузки будет выполняться абсолютно без ошибок. В частности, так как он будет содержать CREATE ROLE для каждой существующей в исходном кластере роли, при попытке создать суперпользователя определённо произойдёт ошибка «role already exists» (роль уже существует), если только целевой кластер был инициализирован не с другим начальным именем суперпользователя. Эта ошибка не критична и её следует просто игнорировать. С ключом `--clean` весьма вероятны другие незначительные сообщения об ошибках, связанные с несуществующими объектами; их число можно минимизировать, добавив ключ `--if-exists`.

При использовании `pg_dumpall` требуется, чтобы все необходимые каталоги табличных пространств существовали до восстановления; в противном случае, создание баз данных в нестандартном размещении завершится ошибкой.

11.6.Примеры

Чтобы выгрузить все базы данных, выполните:

```
$ pg_dumpall > db.out
```

Загрузить базы данных из этого файла можно так:

```
$ psql -f db.out postgres
```

К какой базе данных вы подключаетесь, здесь не важно, так как скрипт, созданный утилитой `pg_dumpall`, будет содержать все команды, требующиеся для создания сохраненных баз данных и подключения к ним.

12. PG_ISREADY

`pg_isready` – проверить соединение с сервером Синергия-БД.

12.1.Синтаксис

```
pg_isready [ опции подключения ... ] [ параметр ... ]
```

12.2.Описание

Утилита `pg_isready` предназначена для проверки соединения с сервером баз данных Синергия-БД. Результат проверки передается в коде завершения.

12.3.Параметры

```
-d dbname  
--dbname=dbname
```

Задаёт имя базы данных, к которой нужно подключиться.

Если этот параметр содержит знак `=` или начинается с допустимого префикса URI (`postgresql://` или `postgres://`), он воспринимается как строка `conninfo`.

```
-h hostname  
--host=hostname
```

Указывает имя компьютера, на котором запущен сервер. Если значение начинается с косой черты, оно интерпретируется как имя каталога с доменным сокетом Unix.

```
-p port  
--port=port
```

Задаёт TCP-порт или файл локального доменного сокета Unix, через который сервер принимает подключения. Значение по умолчанию определяется переменной среды `PGPORT`, а если она не установлена, выбирается порт, указанный при компиляции, обычно 5432.

```
-q  
--quiet
```

Не выводить сообщение о состоянии. Это полезно в скриптах.

```
-t секунды  
--timeout=секунды
```

Максимальное время ожидания (в секундах) при попытке подключения, по истечении которого констатируется, что сервер не отвечает. Значение по умолчанию – 3 секунды.

```
-U username  
--username=username
```


Подключиться к базе данных с заданным именем пользователя вместо подразумеваемого по умолчанию.

-V
--version

Сообщить версию `pg_isready` и завершиться.

-?
--help

Показать справку по аргументам командной строки `pg_isready` и завершиться.

12.4. Код завершения

Утилита `pg_isready` возвращает в оболочку 0, если сервер принимает подключения, 1, если он сбрасывает подключения (например, во время загрузки), 2, если при попытке подключения не получен ответ, и 3, если попытки подключения не было (например, из-за некорректных параметров).

12.5. Переменные окружения

Как и большинство других утилит Синергия-БД, `pg_isready` также использует переменные среды, поддерживаемые `libpq`.

12.6. Замечания

Чтобы получить состояние сервера, передавать имя пользователя, пароль и имя базы данных не требуется; но если передать некорректные значения, сервер выведет в журнал сообщение о неудачной попытке подключения.

12.7. Примеры

Обычное использование:

```
$ pg_isready
/tmp:5432 - accepting connections
$ echo $?
0
```

Запуск с параметрами подключения, во время загрузки кластера Синергия-БД:

```
$ pg_isready -h localhost -p 5433
```

```
localhost:5433 - rejecting connections
```

```
$ echo $?
```

```
1
```

Запуск с параметрами подключения, в случае, когда кластер Синергия-БД недоступен:

```
$ pg_isready -h someremotehost
```

```
someremotehost:5432 - no response
```

```
$ echo $?
```

```
2
```

13. PG_RECEIVEWAL

`pg_receivewal` – получает журналы транзакций с сервера Синергия-БД.

13.1.Синтаксис

```
pg_receivewal [ параметр ...]
```

13.2.Описание

Утилита `pg_receivewal` предназначена для приёма журнала предзаписи от работающего кластера Синергия-БД. Журнал предзаписи передаётся по протоколу потоковой репликации и записывается в локальный каталог. Затем этот каталог можно использовать в качестве архива для восстановления состояния на момент времени.

`pg_receivewal` принимает журнал предзаписи в реальном времени по мере того, как он генерируется на сервере, и не ждёт завершения сегментов, как это делает `archive_command`. Поэтому `pg_receivewal` можно использовать, не устанавливая `archive_timeout`.

В отличие от приёмника WAL, работающего на ведомом сервере Синергия-БД, `pg_receivewal` по умолчанию сохраняет на диск данные WAL, только когда файл WAL закрывается. Чтобы сохранять данные в реальном времени, нужно указать параметр `--synchronous`.

Журнал предзаписи передаётся через обычное подключение к Синергия-БД, с использованием протокола репликации. Подключение должен устанавливать

суперпользователь или пользователь с правом `REPLICATION`, а в `pg_hba.conf` должно разрешаться подключение для репликации. Кроме того, параметр `max_wal_senders` на сервере должен быть достаточно большим, чтобы можно было создать ещё один сеанс для передачи потока.

Если подключение разорвалось или его с самого начала не удаётся установить из-за некритической ошибки, `pg_receivewal` будет бесконечно повторять попытки подключения и восстановит передачу, как только сможет. Чтобы отменить это поведение, воспользуйтесь параметром `-n`.

В отсутствие критичных ошибок `pg_receivewal` будет выполняться до прерывания сигналом SIGINT (**Control+C**).

13.3.Параметры

`-D каталог`
`--directory=каталог`

Каталог, в который будут записываться данные.

Этот параметр является обязательным.

`-E lsn`
`--endpos=lsn`

Автоматически прекратить репликацию и завершить работу с кодом выхода 0 (без ошибки) при достижении заданного LSN.

Если будет получена запись с LSN, равным заданному `lsn`, она будет обработана.

`--if-not-exists`

Не выдавать ошибку, когда указан параметр `--create-slot` и слот с заданным именем уже существует.

`-n`
`--no-loop`

Не повторять цикл при ошибках подключения, а сразу завершать работу, возвращая ошибку.

`--no-sync`

С этим ключом `pg_receivewal` не будет принудительно сбрасывать данные

WAL на диск. Этот вариант быстрее, но при последующем сбое операционной системы сегменты WAL могут оказаться испорченными. Обычно этот ключ полезен при тестировании, но при создании архива WAL в производственной среде использовать его не следует.

Этот ключ несовместим с `--synchronous`.

```
-s интервал  
--status-interval=интервал
```

Указывает интервал в секундах между отправками серверу пакетов состояния. Это позволяет упростить мониторинг прогресса. Чтобы выключить периодическое обновление состояния, необходимо установить значение в ноль. При этом обновление будет отправляться по запросу сервера для избежания отсоединения по истечению времени. Значение по умолчанию составляет 10 секунд.

```
-S имя_слота  
--slot=имя_слота
```

Указывает `pg_receivewal` использовать существующий слот репликации. Когда задан этот параметр, `pg_receivewal` будет сообщать серверу текущую позицию сохранения, отмечая, какой сегмент был сохранён на диске, чтобы сервер мог удалить этот сегмент, если он больше не нужен.

Когда клиент репликации `pg_receivewal` настроен на сервере как синхронный ведомый сервер, для используемого слота репликации серверу будет передаваться позиция сохранённых данных, но только когда файл WAL закрывается. Таким образом, в такой конфигурации транзакции на ведущем сервере будут ожидать завершения продолжительное время и по сути будут работать неудовлетворительно. Чтобы эта конфигурация работала корректно, нужно дополнительно указать параметр `--synchronous`.

```
--synchronous
```

Сохранять данные WAL на диск сразу после того, как они были получены. Также передавать пакет состояния сразу после сохранения, вне зависимости от `--status-interval`.

Этот параметр следует указывать, если клиент репликации `pg_receivewal`

настроен на сервере как синхронный ведомый, чтобы обеспечить своевременную передачу ответа серверу.

```
-v  
--verbose
```

Включает режим подробных сообщений.

```
-Z уровень  
--compress=уровень
```

Включает gzip-сжатие журналов предзаписи и задаёт уровень сжатия от 0 (без сжатия) до 9 (максимальное сжатие). При этом ко всем именам файлов tar добавляется суффикс `.gz`.

Далее описаны параметры управления подключением.

```
-d строка подключения  
--dbname=строка подключения
```

Указывает параметры подключения к серверу в формате строки подключения.

Параметр называется `--dbname` для согласованности с другими клиентскими приложениями, но так как `pg_receivexlog` не подключается к какой-либо конкретной базе, это имя в строке подключения игнорируется.

```
-h host  
--host=host
```

Указывает имя компьютера, на котором запущен сервер. Если значение начинается с косой черты, оно интерпретируется как имя каталога с доменным сокетом Unix. Значение по умолчанию берется из переменной окружения `PGHOST`, если она установлена. В противном случае выполняется подключение к доменному сокету.

```
-p port  
--port=port
```

Указывает TCP-порт или расширение локального файла Unix-сокета, на котором сервер слушает подключения. По умолчанию берется значение переменной окружения `PGPORT`, если оно установлено, либо значение времени компиляции.

```
-U username  
--username=username
```

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Опция может быть полезна в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как `pg_receivexlog` запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако, чтобы понять это, `pg_receivexlog` лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--create-slot`

Создать слот физической репликации с именем, заданным аргументом `--slot`, и завершиться.

`--drop-slot`

Удалить слот репликации с именем, заданным аргументом `--slot`, и завершиться.

Другие флаги:

`-V`
`--version`

Сообщить версию `pg_receivexlog` и завершиться.

`-?`
`--help`

Показать справку по аргументам командной строки `pg_receivexlog` и завершиться.

13.4. Код завершения

`pg_receivewal` завершится с кодом 0 при прерывании сигналом SIGINT.

Это штатный способ его завершения, поэтому получение этого сигнала не считается ошибкой. При критических ошибках или получении других сигналов код выхода будет ненулевым.

13.5. Переменные окружения

Как и большинство других утилит Синергия-БД, приложение также использует переменные окружения, поддерживаемые `libpq`.

13.6. Замечания

Применяя `pg_receivexlog` вместо `archive_command` в качестве основного способа резервного копирования WAL, настоятельно рекомендуется использовать слоты репликации. В противном случае сервер вполне может переписать или удалить файлы журнала транзакций, прежде чем они будут скопированы, так как он не получает никакой информации, через `archive_command` или слоты репликации, о том, как проходит архивация потока WAL. Учтите, однако, что при использовании слота репликации может заполниться все место на диске, если принимающая сторона не будет успевать принимать данные WAL.

`pg_receivewal` сохранит разрешения для группы в полученных файлах WAL, если такие разрешения установлены в исходном кластере.

13.7. Примеры

Следующая команда принимает журнал предзаписи с сервера `mydbserver` и сохраняет его в локальном каталоге `/var/lib/sdb/sdb-11/archive`:

```
$pg_receivewal -h mydbserver -D /var/lib/sdb/sdb-11/archive
```

14. PG_RECVLOGICAL

`pg_recvlogical` – управляет потоками логического декодирования Синергия-БД.

14.1. Синтаксис

```
pg_recvlogical [ параметр ...]
```

14.2.Описание

Утилита `pg_recvlogical` управляет слотами логического декодирования и принимает данные из таких слотов репликации.

Она создает соединение в режиме репликации, так что на него распространяются те же ограничения, что и с `pg_`, плюс ограничения логической репликации.

`pg_recvlogical` не предоставляет возможностей, соответствующих режимам `peek` и `get` в SQL-интерфейсе логического декодирования. Она передаёт серверу подтверждения воспроизведения данных по мере их получения и при штатном выходе. Чтобы просмотреть данные, ожидающие передачи через слот, не принимая их, воспользуйтесь функцией `pg_logical_slot_peek_changes`.

14.3.Параметры

Для выбора действия необходимо указать минимум один из параметров, описанных ниже.

`--create-slot`

Создать новый слот логической репликации с именем, заданным аргументом `--slot`, используя модуль вывода, заданный аргументом `--plugin`, для базы данных, указанной в `--dbname`.

`--drop-slot`

Удалить слот репликации с именем, заданным аргументом `--slot`, и завершиться.

`--start`

Начать прием потока изменений из слота логической репликации с именем, заданным аргументом `--slot`, и продолжать до сигнала прерывания. Если передача потока прерывается на другой стороне из-за выключения или остановки сервера, цикл подключения и передачи повторяется (если не добавлен параметр `--no-loop`).

Формат потока определяется модулем вывода, выбранным при создании слота.

Для получения потока подключаться нужно к той же базе, для которой

создавался слот.

Параметры `--create-slot` и `--start` исключают друг друга. Действие `--drop-slot` несовместимо с любыми другими действиями.

Следующие параметры командной строки управляют расположением и форматом выводимых данных, а также другим поведением репликации:

```
-E lsn  
--endpos=lsn
```

В режиме `--start` автоматически закончить репликацию и выйти с кодом обычного завершения 0, когда при приёме данных достигается указанный LSN. Если этот ключ указывается не в режиме `--start`, выдаётся ошибка.

Если встречается запись с LSN, в точности равным `lsn`, эта запись будет выведена.

С указанием `--endpos` границы транзакций не отслеживаются, так что вывод программы может оказаться обрезанным посередине транзакции. Частично полученная транзакция не будет считаться принятой и будет воспроизведена заново при следующем чтении из этого слота. Отдельные сообщения не обрезаются никогда.

```
-f имя_файла  
--file=имя_файла
```

Записывать полученные и декодированные данные транзакций в указанный файл. Для вывода в `stdout` укажите `-` (минус).

```
-F секунды  
--fsync-interval=секунды
```

Устанавливает, как часто `pg_recvlogical` будет вызывать `fsync()`, чтобы гарантировать, что выходной файл надёжно сохранен на диске.

Сервер время от времени дает клиенту команду сохранить данные и сообщить сохраненную позицию, но этот параметр позволяет выполнять сохранение чаще.

При значении, равном 0, функция `fsync()` вообще не вызывается, но серверу сообщается новая позиция. Это может привести к потере данных в случае сбоя.

```
-I lsn  
--startpos=lsn
```

В режиме `--start` репликация начнется с данного LSN. В других режимах игнорируется.

`--if-not-exists`

Не выдавать ошибку, когда указан параметр `--create-slot` и слот с заданным именем уже существует.

`-n`

`--no-loop`

Когда подключение к серверу потеряно, не повторять цикл, просто завершить работу.

`-o имя [=значение]`

`--option=имя [=значение]`

Передаёт параметр `имя_параметра` модулю вывода, при этом может быть передано и его значение. Набор параметров и их действия зависят от выбранного модуля вывода.

`-P модуль`

`--plugin=модуль`

Использовать указанный модуль вывода логического декодирования при создании слота. Этот параметр не действует, если слот уже существует.

`-s секунды`

`--status-interval=секунды`

Этот параметр действует так же, как одноименный параметр `pg_receivexlog`.

`-S slot_name`

`--slot=slot_name`

Этот параметр задает имя слота логической репликации, который будет использоваться в режиме `--start`, создаваться в режиме `--create-slot` или удаляться в режиме `--drop-slot`.

`-v`

`--verbose`

Включает режим подробных сообщений.

Далее описаны параметры управления подключением.

`-d database`

```
--dbname=database
```

Этот параметр выбирает базу данных для подключения. Как именно она используется, можно узнать в описании соответствующих действий. Это может быть строка подключения `libpq`; за дополнительными сведениями обратитесь к `.`. По умолчанию её имя определяется именем пользователя.

```
-h имя_компьютера-или-ip  
--host=имя_компьютера-или-ip
```

Указывает имя компьютера, на котором запущен сервер. Если значение начинается с косой черты, оно интерпретируется как имя каталога с доменным сокетом Unix. Значение по умолчанию берется из переменной окружения `PGHOST`, если она установлена. В противном случае выполняется подключение к доменному сокету.

```
-p port  
--port=port
```

Указывает TCP-порт или расширение локального файла Unix-сокета, на котором сервер слушает подключения. По умолчанию берется значение переменной окружения `PGPORT`, если оно установлено, либо значение времени компиляции.

```
-U user  
--username=user
```

Имя пользователя для подключения. По умолчанию это имя текущего пользователя операционной системы.

```
-w  
--no-password
```

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Опция может быть полезна в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

```
-W  
--password
```

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как `pg_recvlogical` запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако,

чтобы понять это, `pg_recvlogical` лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Также есть следующие дополнительные параметры:

`-V`
`--version`

Сообщить версию `pg_recvlogical` и завершиться.

`-?`
`--help`

Показать справку по аргументам командной строки `pg_recvlogical` и завершиться.

14.4.Переменные окружения

Как и большинство других утилит Синергия-БД, приложение также использует переменные окружения, поддерживаемые `libpq`.

14.5.Замечания

`pg_recvlogical` сохранит разрешения для группы в полученных файлах WAL, если такие разрешения установлены в исходном кластере.

15. PG_RESTORE

`pg_restore` – восстановить базу данных Синергия-БД из файла архива, созданного командой `pg_dump`.

15.1.Синтаксис

`pg_restore` [опции подключения ...] [параметр ...]
[имя_файла]

15.2.Описание

Утилита `pg_restore` предназначена для восстановления базы данных Синергия-БД из архива, созданного командой `pg_dump` в любом из не текстовых форматов. Она выполняет команды, необходимые для восстановления того состояния базы данных, в котором база была сохранена. При наличии файлов

архивов, `pg_restore` может восстанавливать данные избирательно или даже переупорядочить объекты перед восстановлением. Заметьте, что разработанный для файлов архива формат не привязан к архитектуре.

Утилита `pg_restore` может работать в двух режимах. Если указывается имя базы данных, `pg_restore` подключается к этой базе данных и восстанавливает содержимое архива непосредственно в нее. В противном случае, создается SQL-скрипт с командами, необходимыми для пересоздания базы данных, который затем выводится в файл или в стандартное устройство вывода. Сформированный скрипт будет в точности соответствовать выводу `pg_dump` в простом текстовом формате. Поэтому некоторые из параметров, управляющих выводом, аналогичны параметрам `pg_dump`.

Разумеется, `pg_restore` может восстановить информацию, только если она присутствует в файле архива, и только в существующем виде. Например, если архив был создан с указанием «выгрузить данные в виде команд `INSERT`», `pg_restore` не сможет загрузить эти данные, используя операторы `COPY`.

15.3.Параметры

Утилита `pg_restore` принимает аргументы командной строки, описанные ниже.

имя_файла

Указывает расположение восстанавливаемого файла архива (или каталога, для архива в формате каталога). По умолчанию используется устройство стандартного ввода.

-a

--data-only

Восстанавливать только данные, без схемы (определений данных). При этом восстанавливаются данные таблиц, большие объекты и значения последовательностей, имеющиеся в архиве.

Флаг похож на `--section=data`, но по историческим причинам не равнозначен ему.

`-c`
`--clean`

Удалить (DROP) объекты базы данных, прежде чем пересоздавать их. Без дополнительного указания `--if-exists` при этом могут выдаваться безвредные сообщения об ошибках, если таких объектов не окажется в целевой базе данных.

`-C`
`--create`

Создать базу данных, прежде чем восстанавливать данные. Если также указан параметр `--clean`, удалить и пересоздать целевую базу данных перед подключением к ней.

С ключом `--create` программа `pg_restore` также восстанавливает комментарий к базе данных (если он задан) и все назначения переменных конфигурации, связанные с базой данных, то есть все команды `ALTER DATABASE ... SET ...` и `ALTER ROLE ... IN DATABASE ... SET ...`, ссылающиеся на эту базу данных. Также восстанавливаются права доступа к самой базе данных, если не добавлен ключ `--no-acl`.

С этим параметром база, заданная параметром `-d`, применяется только для подключения и выполнения начальных команд `DROP DATABASE` и `CREATE DATABASE`. Все данные восстанавливаются в базу данных, имя которой записано в архиве.

`-d dbname`
`--dbname=dbname`

Подключиться к базе данных `имя_базы` и восстановить данные непосредственно в нее.

`-e`
`--exit-on-error`

Завершать работу в случае возникновения ошибки при выполнении команд SQL в базе данных. По умолчанию процесс восстановления продолжается, а по его окончании выдается число ошибок.

`-f имя_файла`
`--file=имя_файла`

Задаёт файл для вывода сгенерированного скрипта или списка объектов, полученного с параметром `-l`. По умолчанию используется стандартный вывод.

```
-F format  
--format=format
```

Задаёт формат архива. Указывать формат необязательно, так как `pg_restore` определяет формат автоматически. Но если формат задается, допускается один из этих вариантов:

```
c  
custom
```

Архив сохранен в настраиваемом формате `pg_dump`.

```
d  
directory
```

Архив сохранен в каталоге.

```
t  
tar
```

Архив сохранен в формате `tar`.

```
-I индекс  
--index=индекс
```

Восстановить определение только заданного индекса. Добавив дополнительные ключи `-I`, можно указать несколько индексов.

```
-j число-заданий  
--jobs=число-заданий
```

Запустить наиболее длительные операции `pg_restore` (в частности, загрузку данных, создание индексов или ограничений) в нескольких параллельных заданиях. Это позволяет кардинально сократить время восстановления большой базы данных, когда сервер работает на многопроцессорном компьютере.

Каждое задание выполняется в отдельном задании или потоке, в зависимости от операционной системы, и использует отдельное подключение к серверу.

Оптимальное значение этого параметра зависит от аппаратной конфигурации сервера, клиента и сети. В частности, имеет значение количество процессорных ядер и устройство дискового хранилища. В качестве начального значения можно выбрать число ядер на сервере, но и при увеличении этого значения во многих случаях

восстановление будет быстрее. Конечно, при слишком больших значениях производительность упадет из-за перегрузки.

Этот параметр поддерживается только с архивом в специальном формате или в каталоге. Входные данные должны поступать из обычного файла или каталога (а не, например из канала ввода). Данный параметр игнорируется, когда генерируется скрипт (нет прямого подключения к базе данных). Кроме того, несколько заданий не могут выполняться в сочетании с параметром `--single-transaction`.

```
-l  
--list
```

Вывести содержимое архива. Вывод этой операции можно подать на вход этой же команде с параметром `-L`. Учтите, что когда вместе с `-l` применяются параметры фильтрации (например, `-n` или `-t`), они сокращают список выводимых объектов.

```
-L файл-список  
--use-list=файл-список
```

Восстановить из архива только элементы, перечисленные в файле-списке, и в том порядке, в каком они идут в этом файле. Заметьте, что когда вместе с `-L` применяются параметры фильтрации (например, `-n` или `-t`), они дополнительно ограничивают восстанавливаемые объекты.

Данный файл-список обычно представляет собой отредактированный результат предыдущей операции `-l`. Строки в нем могут быть переставлены или удалены, а также могут быть закомментированы точкой с запятой (;), добавленной в начале строки.

```
-n схема  
--schema=схема
```

Восстановить только объекты в указанной схеме. Добавив дополнительные ключи `-n`, можно указать несколько схем. Этот параметр можно сочетать с `-t`, чтобы восстановить только определенную таблицу.

```
-N схема  
--exclude-schema=схема
```

Не восстанавливать объекты в указанной схеме. Добавив дополнительные ключи `-N`, можно исключить несколько схем.

Когда и с ключом `-n`, и с ключом `-N` передаётся имя одной схемы, ключ `-N` выигрывает и схема исключается.

```
-O
--no-owner
```

Не генерировать команды, устанавливающие владение объектами, как в исходной базе данных. По умолчанию, `pg_restore` генерирует команды `ALTER OWNER` или `SET SESSION AUTHORIZATION`, восстанавливающие исходных владельцев создаваемых элементов схемы. Однако эти команды можно будет выполнить, только если к базе данных первоначально подключается суперпользователь (или пользователь, владеющими всеми объектами в скрипте). Чтобы получить скрипт, который сможет восстановить любой подключающийся пользователь (но при этом он станет владельцем всех созданных объектов), используется `-O`.

```
-P имя-функции(тип-аргумента[, ...])
--function=имя-функции(тип-аргумента[, ...])
```

Восстановить только указанную функцию. При этом важно записать имя функции и аргументы в точности так, как они фигурируют в оглавлении файла архива. Добавив дополнительные ключи `-P`, можно указать несколько функций.

```
-R
--no-reconnect
```

Параметр является устаревшим, но в целях совместимости ещё работает.

```
-s
--schema-only
```

Восстановить только схему (определения данных), без данных, в объеме, в котором элементы схемы представлены в архиве.

Действие опции противоположно действию опции `--data-only`. Это похоже на указание `--section=pre-data` `--section=post-data`, но по историческим причинам не равнозначно ему.

```
-S username
--superuser=username
```

Задаёт имя суперпользователя, полномочия которого будут использоваться для

отключения триггеров. Этот параметр применяется только с параметром `--disable-triggers`.

```
-t таблица  
--table=таблица
```

Восстановить определение и/или данные только указанной таблицы. В этом контексте под «таблицей» подразумеваются также представления, материализованные представления, последовательности и сторонние таблицы. Чтобы выбрать несколько таблиц, ключ `-t` можно указать несколько раз. Этот параметр можно скомбинировать с `-n`, чтобы выбрать таблицу(ы) в определённой схеме.

Примечания:

1. Когда указывается `-t`, `pg_restore` не пытается восстанавливать объекты базы данных, от которых могут зависеть выбранные таблицы. Таким образом, в этом случае не гарантируется, что выгруженные таблицы будут успешно восстановлены в чистой базе данных.

2. Этот флаг действует не совсем так, как флаг `-t` утилиты `pg_dump`. В настоящее время `pg_restore` не поддерживает выбор объектов по маске, а также не позволяет указать имя схемы с `-t`. И хотя `pg_dump` с флагом `-t` также выгружает подчинённые объекты (например, индексы) выбранных таблиц, `pg_restore` с флагом `-t` такие подчинённые объекты не обрабатывает.

```
-T триггер  
--trigger=триггер
```

Восстановить только указанный триггер. Добавив дополнительные ключи `-T`, можно указать несколько триггеров.

```
-v  
--verbose
```

Включает режим подробных сообщений.

```
-V  
--version
```

Сообщить версию `pg_restore` и завершиться.

```
-x  
--no-privileges  
--no-acl
```

Не восстанавливать права доступа (не выполнять команды `GRANT/REVOKE`).

```
-1
```

`--single-transaction`

Произвести восстановление в одной транзакции (то есть, завернуть выполняемые команды в BEGIN/COMMIT). При этом гарантируется, что либо все команды будут выполнены успешно, либо не будет никаких изменений. Этот режим подразумевает `--exit-on-error`.

`--disable-triggers`

Этот параметр действует только при выгрузке одних данных. С ним `pg_restore` выполняет команды, отключающие триггеры в целевых таблицах на время загрузки данных. Используйте его, если в ваших таблицах определены проверки ссылочной целостности или другие триггеры, которые вы не хотели бы выполнять в процессе загрузки данных.

В настоящее время команды, генерируемые с `--disable-triggers`, должны выполняться суперпользователем. Поэтому необходимо также задать имя суперпользователя в параметре `-S` или, что предпочтительнее, запускать `pg_restore` от имени суперпользователя Синергия-БД.

`--enable-row-security`

Этот параметр имеет смысл только при восстановлении содержимого таблицы, для которой включена защита строк. По умолчанию `pg_restore` устанавливает для `row_security` значение `off` для уверенности, что в таблице восстановлены все данные. Если у пользователя недостаточно прав для обхода защиты строк, выдаётся ошибка. Этот параметр указывает `pg_restore` установить в `row_security` значение `on`, чтобы пользователь мог попытаться восстановить содержимое таблицы с включённой защитой строк. Однако и при этом возможна ошибка, если пользователь не будет иметь права добавлять в эту таблицу выгруженные строки данных.

Заметьте, что в настоящее время для этого требуется, чтобы выгрузка выполнялась в режиме INSERT, так как COPY FROM не поддерживает защиту строк.

`--if-exists`

При очистке целевой базы использовать условные команды (добавлять фразу IF EXISTS). Применяется только с опцией `--clean`.

`--no-comments`

Не выводить команды, восстанавливающие комментарии, даже если они содержатся в архиве.

`--no-data-for-failed-tables`

По умолчанию данные восстанавливаются даже при ошибке команды создания таблицы (например, когда она уже существует). С этим параметром данные в таком случае не восстанавливаются. Это поведение полезно, если в целевой таблице уже содержатся нужные данные. Например, вспомогательные таблицы для расширений Синергия-БД (в частности, PostGIS) могут быть уже заполнены; этот параметр позволяет предотвратить дублирование или загрузку устаревших данных в эти таблицы.

Этот параметр действует только при восстановлении непосредственно в базу данных (не при генерации SQL-скрипта).

`--no-publications`

Не выводить команды, восстанавливающие публикации, даже если они содержатся в архиве.

`--no-security-labels`

Не выводить команды, восстанавливающие метки безопасности, даже если они содержатся в архиве.

`--no-subscriptions`

Не выводить команды, восстанавливающие подписки, даже если они содержатся в архиве.

`--no-tablespaces`

Не формировать команды для указания табличных пространств. Все объекты будут создаваться в табличном пространстве по умолчанию.

`--section=имя секции`

Восстановить только указанный раздел. В качестве имени раздела можно задать `pre-data`, `data` или `post-data`. Указав этот параметр неоднократно, можно выбрать несколько разделов. По умолчанию восстанавливаются все разделы.

Раздел «`data`» содержит собственно данные таблиц и определения больших

объектов. В разделе «post-data» содержатся определения индексов, триггеров, правил и ограничений (кроме отдельно проверяемых). Раздел «pre-data» содержит все остальные определения.

```
--strict-names
```

Требует, чтобы каждому указанию схемы (`-n/--schema`) и таблицы (`-t/--table`) соответствовала минимум одна схема/таблица в файле резервной копии.

```
--use-set-session-authorization
```

Выводить команды `SET SESSION AUTHORIZATION`, соответствующие стандарту, вместо `ALTER OWNER`, для назначения владельцев объектов. В результате выгруженный скрипт будет более стандартизированным, но может не восстановиться корректно, в зависимости от истории объектов.

```
-?
```

```
--help
```

Показать справку по аргументам командной строки `pg_restore` и завершиться.

`pg_restore` также принимает в качестве параметров соединения следующие аргументы командной строки:

```
-h host
```

```
--host=host
```

Указывает имя компьютера, на котором запущен сервер. Если значение начинается с косой черты, оно интерпретируется как имя каталога с доменным сокетом Unix. Значение по умолчанию берется из переменной окружения `PGHOST`, если она установлена. В противном случае выполняется подключение к доменному сокету.

```
-p port
```

```
--port=port
```

Указывает TCP-порт или расширение локального файла Unix-сокета, на котором сервер слушает подключения. По умолчанию берется значение переменной окружения `PGPORT`, если оно установлено, либо значение времени компиляции.

```
-U username  
--username=username
```

Имя пользователя, под которым производится подключение.

```
-w  
--no-password
```

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Опция может быть полезна в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

```
-W  
--password
```

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как `pg_restore` запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако, чтобы понять это, `pg_restore` лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

```
--role=имя роли
```

Задаёт имя роли, которая будет осуществлять восстановление. Получив это имя, `pg_restore` выполнит `SET ROLE имя_роли` после подключения к базе данных. Это полезно, когда проходящий проверку пользователь (указанный в `-U`) не имеет прав, необходимых для `pg_restore`, но может переключиться на роль, наделённую этими правами. В некоторых окружениях правила запрещают подключаться к серверу непосредственно суперпользователю, и этот параметр позволяет выполнить восстановление, не нарушая их.

15.4. Переменные окружения

```
PGHOST  
PGOPTIONS  
PGPORT  
PGUSER
```

Параметры подключения по умолчанию.

Как и большинство других утилит Синергия-БД, эта утилита также использует

переменные среды, поддерживаемые `libpq`. Однако она не учитывает `PGDATABASE`, когда имя базы не указано.

15.5.Диагностика

Когда с параметром `-d` устанавливается прямое подключение к базе данных, `pg_restore` выполняет обычные операторы SQL. При этом применяются все свойства подключения по умолчанию и переменные окружения, которые использует клиентская библиотека `libpq`. Если вы сталкиваетесь с проблемами при запуске `pg_restore`, убедитесь в том, что вы можете получить информацию из базы данных, используя, например `psql`.

15.6.Замечания

Если в вашей инсталляции база данных `template1` содержит какие-либо дополнения, важно убедиться в том, что вывод `pg_restore` загружается в действительно пустую базу; иначе вы, скорее всего, получите ошибки из-за дублирования определений создаваемых объектов. Чтобы получить пустую базу данных без дополнительных объектов, выберите в качестве шаблона `template0`, а не `template1`, например так:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

Ограничения `pg_restore` описаны ниже:

1. При восстановлении данных в уже существующие таблицы с параметром `--disable-triggers`, `pg_restore` выполняет команды, отключающие триггеры в пользовательских таблицах до добавления данных, а затем, после добавления данных, выполняет команды, снова включающие эти триггеры. Если восстановление прервется в середине, системные каталоги могут оказаться в некорректном состоянии.
2. Утилита `pg_restore` не способна восстанавливать большие объекты избирательно; например, только для определенной таблицы. Если архив содержит большие объекты, они будут восстановлены все, либо не будут восстановлены никакие (если они были исключены параметрами `-L`, `-t`

и т.п.).

Также обратитесь к документации `pg_dump`, чтобы узнать о связанных ограничениях `pg_dump`.

После восстановления имеет смысл запустить `ANALYZE` для каждой восстановленной таблицы, чтобы оптимизатор получил актуальную статистику; за дополнительными сведениями обратитесь к и .

15.7.Примеры

Предположим, что мы выгрузили базу данных `mydb` в файл специального формата:

```
$ pg_dump -Fc mydb > db.dump
```

Мы можем удалить базу данных и восстановить её из копии:

```
$ dropdb mydb
```

```
$ pg_restore -C -d postgres db.dump
```

В аргументе `-d` можно указать любую базу данных, существующую в кластере; `pg_restore` использует её, только чтобы выполнить команду `CREATE DATABASE` для базы `mydb`. С параметром `-C` данные всегда восстанавливаются в базу, имя которой записано в файле архива.

Восстановить данные в новую базу `newdb` можно так:

```
$ createdb -T template0 newdb
```

```
$ pg_restore -d newdb db.dump
```

Обратите внимание, мы не используем параметр `-C`, а вместо этого подключаемся непосредственно к базе, в которую хотим восстановить данные. Также заметьте, что мы создаем базу данных из шаблона `template0`, а не `template1`, чтобы изначально она была гарантированно пустой.

Чтобы переупорядочить элементы базы данных, сначала необходимо получить оглавление архива:

```
$ pg_restore -l db.dump > db.list
```

Файл оглавления содержит заголовок и по одной строке для каждого элемента,

например:

```
; Archive created at Mon Sep 14 13:55:39 2009
;      dbname: DBDEMOS
;      TOC Entries: 81
;      Compression: 9
;      Dump Version: 1.10-0
;      Format: CUSTOM
;      Integer: 4 bytes
;      Offset: 8 bytes
;      Dumped from database version: 8.3.5
;      Dumped by pg_dump version: 8.3.8
;
;
; Selected TOC Entries:
;
3; 2615 2200 SCHEMA - public pasha
1861; 0 0 COMMENT - SCHEMA public pasha
1862; 0 0 ACL - public pasha
317; 1247 17715 TYPE public composite pasha
319; 1247 25899 DOMAIN public domain0 pasha
```

С точки с запятой начинаются комментарии, а число в начале строки обозначает внутренний идентификатор, назначаемый каждому элементу в архиве.

Строки в этом файле можно закомментировать, удалить или переместить.

Например, список:

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

можно передать утилите `pg_restore`, чтобы восстановить только элементы 10 и 6 в указанном порядке:

```
$ pg_restore -L db.list db.dump
```

16. PG-WRAPPER

pg-wrapper – управление символическими ссылками Синергия-БД.

16.1.Синтаксис

```
pg-wrapperlinks { update | remove }
```

16.2.Описание

Программа pg-wrapper представляет собой скрипт оболочки, включённый в дистрибутив Синергия-БД и предназначенный для управления символическими ссылками на программы Синергия-БД и подключения страниц man по SQL в системах Linux. Так как Синергия-БД устанавливается в /opt/sdb/sdb-11, это необходимо, чтобы клиентские и серверные программы находились в стандартном системном пути (PATH), а также находились новые страницы man по SQL. Эти действия не производятся автоматически во время установки (исключение составляет пакет быстрой установки postgresql-sdb-11) во избежание возможных конфликтов с другими продуктами на базе PostgreSQL, которые могут уже быть установлены.

Скрипт pg-wrapper входит в состав пакета postgresql-sdb-11-client. Установив этот пакет, вы сможете найти pg-wrapper в каталоге *каталог-инсталляции/bin*, где *каталог-инсталляции* обозначает путь /opt/sdb/sdb-11.

Скрипт pg-wrapper должен запускать пользователь root.

16.3.Параметры

pg-wrapper принимает аргументы командной строки, описанные ниже.

```
links {update | remove}
```

Управление символическими ссылками Синергия-БД и страницами man по SQL:

- update – создать или заменить символические ссылки для серверных или

клиентских программ, распространяемых в составе Синергия-БД, а также подключить страницы man по SQL в файле конфигурации страниц man.

- `remove` – удалить символические ссылки и поддержку страниц man по SQL для текущей версии Синергия-БД.

16.4.Замечания

Если вы устанавливаете Синергия-БД из пакета `postgresql-sdb-11`, `pg-wrapper` вызывается автоматически.

Если вы устанавливаете отдельные пакеты Синергия-БД, вы можете запустить этот скрипт вручную, чтобы создать символические ссылки на предоставляемые клиентские и серверные программы, а также подключить страницы man по SQL в файле конфигурации страниц man.

При параллельной установке с другими продуктами на базе PostgreSQL поведение `pg-wrapper` зависит от того, поддерживается ли механизм `update-alternatives` вашей системой и уже установленным продуктом на базе PostgreSQL:

- Если поддерживается механизм `update-alternatives`, `pg-wrapper` добавляет символические ссылки на программы Синергия-БД в систему альтернатив, в соответствии с их приоритетом, а также подключает страницы man по SQL в файл конфигурации страницы man. Информацию об изменении приоритетов в системе альтернатив вы найдёте в руководстве man `update-alternatives` в вашей системе.

- Если в системе уже установлены страницы man по SQL от другого продукта, от новой инсталляции будут выводиться только новые страницы, а все совпадающие страницы man – от старой. Чтобы новая документация SQL выводилась для всех страниц, вам потребуется изменить системную конфигурацию, например, поменять значение `MANPATH`.

- Если механизм `update-alternatives` не поддерживается, `pg-wrapper` изменяет системную конфигурацию, только если отсутствуют конфликты с любыми другими установленными программами или страницами man. В случае таких конфликтов `pg-wrapper` не создаёт и не изменяет ссылки

программ и пропускает интеграцию страниц map по SQL. В этом случае вы можете либо продолжать использовать ранее установленные версии программ и страниц map по SQL, либо изменить параметры PATH и MANPATH вручную.

17. PSQL

psql – интерактивный терминал Синергия-БД.

17.1. Синтаксис

```
psql [ параметр ... ] [dbname [username]]
```

17.2. Описание

psql - это терминальный клиент для работы с Синергия-БД. Позволяет интерактивно вводить запросы, отправлять их в Синергия-БД и смотреть результаты. Ввод может быть не только интерактивным, но и из файла. Кроме того, предоставляется ряд мета-команд и различные возможности подобные тем, что имеются у командных оболочек, для облегчения написания скриптов и автоматизации широкого спектра задач.

17.3. Параметры

```
-a  
--echo-all
```

Отправляет на стандартный вывод все непустые входные строки по мере их чтения. Это не относится к строкам, считанным в интерактивном режиме. Эквивалентно установке переменной ECHO в значение all.

```
-A  
--no-align
```

Переключен на невыровненный режим вывода. По умолчанию, наоборот, используется выровненный режим вывода.

```
-b  
--echo-errors
```

Посылает все команды SQL с ошибками на стандартный вывод. Равнозначно присвоению переменной ECHO значения errors.

```
-с команда
```

--command=команда

Передаёт `psql` команду для выполнения. Этот ключ можно повторять и комбинировать в любом порядке с ключом `-f`. Когда указывается `-с` или `-f`, `psql` не читает команды со стандартного ввода; вместо этого она завершается сразу после обработки всех ключей `-с` и `-f` по порядку.

Заданная команда должна быть либо командной строкой, которая полностью интерпретируется сервером (т. е. не использует специфические функции `psql`), либо одиночной командой с обратной косой чертой. Таким образом, используя `-с`, нельзя смешивать метакоманды SQL и `psql`. Но это можно сделать, передав несколько ключей `-с` или передав строку в `psql` через канал:

```
psql -с '\x' -с 'SELECT * FROM foo;'
```

или

```
echo '\x \ SELECT * FROM foo;' | psql
```

`\` – разделитель метакоманд.

Каждая строка SQL-команд, заданная ключом `-с`, передаётся на сервер как один запрос. Поэтому сервер выполняет её в одной транзакции, даже когда эта строка содержит несколько команд SQL, если только в ней не содержатся явные команды `BEGIN/COMMIT`, разделяющие её на несколько транзакций. Кроме того, `psql` печатает результат только последней SQL-команды в строке. Это отличается от поведения, когда та же строка считывается из файла или подаётся на стандартный ввод `psql`, так как в последнем случае `psql` передаёт каждую команду SQL отдельно.

Из-за такого поведения указание нескольких SQL-команд в одной строке `-с` часто приводит к неожиданным результатам. Поэтому лучше использовать несколько ключей `-с` или подавать команды на стандартный ввод `psql`, применяя либо `echo`, как показано выше, либо создаваемый прямо в оболочке текст, например:

```
psql <<EOF
\x
SELECT * FROM foo;
EOF
-d dbname
```

```
--dbname=dbname
```

Указывает имя базы данных для подключения. Эквивалентно указанию `dbname` в качестве первого аргумента, не являющегося параметром в командной строке.

Если этот параметр содержит знак `=` или начинается с допустимого URI префикса (`postgresql://` или `postgres://`), он рассматривается как строка `conninfo`.

```
-e  
--echo-queries
```

Посылает все команды SQL, отправленные на сервер, ещё и на стандартный вывод. Эквивалентно установке переменной `ECHO` в значение `queries`.

```
-E  
--echo-hidden
```

Отображает фактические запросы, генерируемые `\d` и другими командами, начинающимися с `\`. Это можно использовать для изучения внутренних операций в `psql`. Эквивалентно установке переменной `ECHO_HIDDEN` значения `on`.

```
-f имя_файла  
--file=имя_файла
```

Использует файл `filename` в качестве источника команд вместо чтения команд в интерактивном режиме. После обработки файла работа `psql` завершается. Это во многом эквивалентно мета-команде `\i`.

Если `filename` это `-` (дефис), то читается стандартный ввод.

Использование этой опции немного отличается от `psql < filename`. В основном, оба варианта будут делать то, что вы ожидаете, но с `-f` доступны некоторые полезные свойства, такие как сообщения об ошибках с номерами строк. Также есть небольшая вероятность, что с этой опцией уменьшаются накладные расходы на запуск. С другой стороны, вариант с перенаправлением ввода из командного интерпретатора (в теории) гарантирует получение точно такого же вывода, какой вы получили бы, если ввели все вручную.

```
-F separator  
--field-separator=separator
```

Использование `separator` в качестве разделителя полей при невыровненном режиме вывода. Эквивалентно `\pset fieldsep` или `\f`.

```
-h hostname  
--host=hostname
```

Указывает имя хоста машины, на которой запущен сервер. Если значение начинается с косой черты, то оно используется в качестве директории для доменного сокета Unix.

```
-H  
--html
```

Включает табличный вывод в формате HTML. Эквивалентно `\pset format html` или команде `\H`.

```
-l  
--list
```

Выводит список всех доступных баз данных и завершает работу. Другие опции, не связанные с соединением, игнорируются. Это похоже на мета-команду `\list`.

```
-L имя_файла  
--log-file=имя_файла
```

В дополнение к обычному выводу, записывает вывод результатов всех запросов в файл `filename`.

```
-n  
--no-readline
```

Отключает использование Readline для редактирования командной строки и использования истории команд. Может быть полезно для выключения расширенных действий клавиши табуляции при вырезании и вставке.

```
-o имя_файла  
--output=имя_файла
```

Записывает вывод результатов всех запросов в файл `filename`. Эквивалентно команде `\o`.

```
-p port  
--port=port
```

Задаёт порт TCP или локальный доменный сокет Unix, на котором сервер

прослушивает соединения. По умолчанию используется значение переменной среды `PGPORT`, если оно не установлено, то используется значение, указанное во время компиляции, обычно 5432.

```
-P assignment  
--pset=assignment
```

Задаёт параметры печати, в стиле команды `\pset`. Обратите внимание, что имя параметра и значение разделяются знаком равенства, а не пробелом. Например, чтобы установить формат вывода в LaTeX, нужно написать `-P format=latex`.

```
-q  
--quiet
```

Указывает, что `psql` должен работать без вывода дополнительных сообщений. По умолчанию, выводятся приветствия и различные информационные сообщения. Этого не произойдет с использованием данной опции. Полезно вместе с опцией `-c`. Это эквивалентно установке переменной `QUIET` значения `on`.

```
-R separator  
--record-separator=separator
```

Использует `separator` как разделитель записей при невыровненном режиме вывода. Эквивалентно команде `\pset recordsep`.

```
-s  
--single-step
```

Запуск в пошаговом режиме. Это означает, что пользователь будет подтверждать выполнение каждой команды, отправляемой на сервер, с возможностью отменить выполнение. Используется для отладки скриптов.

```
-S  
--single-line
```

Запуск в однострочном режиме, при котором символ новой строки завершает SQL команды, также как это делает точка с запятой.

Примечание. Этот режим предназначен для тех, кто на нем настаивает, но вы не обязаны рекомендовать его использовать. В частности, если смешать на одной строке команды SQL и мета-команды, то порядок их выполнения может быть не всегда очевиден для неопытного пользователя.

```
-t  
--tuples-only
```


Отключает вывод имен столбцов и результирующей строки с количеством выбранных записей. Эквивалентно команде `\t`.

```
-T table_options  
--table-attr=table_options
```

Задаёт атрибуты, которые будут вставлены в тег HTML `table`. Смотри `\pset` для деталей.

```
-U username  
--username=username
```

Подключение к базе данных под пользователем `username` вместо используемого по умолчанию.

```
-v assignment  
--set=assignment  
--variable=assignment
```

Выполняет присвоение значения переменной, как мета-команда `\set`. Обратите внимание на то, что необходимо разделить имя переменной и значение (при наличии) знаком равенства в командной строке. Чтобы сбросить переменную, опустите знак равенства. Чтобы установить пустое значение, поставьте знак равенства, но опустите значение. Присваивания выполняются на очень ранней стадии запуска, поэтому если переменные зарезервированы для внутренних целей, то позже они могут быть перезаписаны.

```
-V  
--version
```

Выводит версию `psql` и завершает работу.

```
-w  
--no-password
```

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Опция может быть полезна в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

Обратите внимание, что эта опция действует на протяжении всей сессии и, таким образом, влияет на мета-команду `\connect`, также как и на первую попытку соединения.

`-W`
`--password`

Принудительно запрашивает пароль перед подключением к базе данных.

Эта опция не существенна, так как `psql` автоматически запрашивает пароль, если сервер требует аутентификацию по паролю. Однако, `psql` использует дополнительную попытку соединения для выяснения того, что серверу требуется пароль. В некоторых случаях стоит вводить `-W`, чтобы избежать лишней попытки соединения.

Обратите внимание, что эта опция действует на протяжении всей сессии и, таким образом, влияет на мета-команду `\connect`, также как и на первую попытку соединения.

`-x`
`--expanded`

Включает режим развернутого вывода таблицы. Эквивалентно команде `\x`.

`-X,`
`--no-psqlrc`

Не читать стартовые файлы (ни общесистемный файл `psqlrc`, ни пользовательский файл `~/.psqlrc`).

`-z`
`--field-separator-zero`

Установить нулевой байт в качестве разделителя полей для невыровненного режима вывода.

`-0`
`--record-separator-zero`

Установить нулевой байт в качестве разделителя записей для невыровненного режима вывода. Это полезно при взаимодействии с другими программами, например, с `xargs -0`.

`-1`
`--single-transaction`

Если `psql` выполняет скрипт, то добавление этой опции заключает скрипт в `BEGIN/COMMIT` для выполнения в рамках одной транзакции. Это гарантирует, что либо все команды успешно завершены, либо никаких изменений не произведено.

Если в самом скрипте используются `BEGIN`, `COMMIT` или `ROLLBACK`, то эта опция не будет иметь желаемого эффекта. Кроме того, если скрипт содержит любую команду, которая не может быть выполнена внутри транзакционного блока, указание этой опции приведет к сбою команды и, следовательно, всей транзакции.

```
-?  
-- help[=тема]
```

Показать справку по `psql` и завершиться. Необязательный параметр `тема` (по умолчанию `options`) выбирает описание интересующей части `psql`: `commands` описывает команды `psql` с обратной косой чертой; `options` описывает параметры командной строки, которые можно передать `psql`; а `variables` выдаёт справку по переменным конфигурации `psql`.

17.4. Код завершения

При нормальном завершении `psql` возвращает 0 в командную оболочку ОС, 1 - если произошла фатальная ошибка в самом `psql` (например, нехватка памяти, файл не найден), 2 - при неудачном соединении с сервером неинтерактивного сеанса, 3 - при ошибке в скрипте и установленной переменной `ON_ERROR_STOP`.

17.5. Использование

17.5.1. Подключение к базе данных

`psql` это клиент для Синергия-БД. Для подключения к базе данных нужно знать имя базы данных, имя сервера, номер порта сервера и имя пользователя, под которым вы хотите подключиться. Эти свойства можно задать через аргументы командной строки, а именно `-d`, `-h`, `-p` и `-U` соответственно. Если в командной строке есть аргумент, который не относится к параметрам `psql`, то он используется в качестве имени базы данных (или имени пользователя, если база данных уже задана). Задавать все эти аргументы необязательно, у них есть разумные значения по умолчанию. Если опустить имя сервера, `psql` будет подключаться через Unix-сокеты к локальному серверу, либо подключаться к `localhost` по TCP/IP в системах, не поддерживающих UNIX-сокеты. Номер порта по умолчанию определяется во время

компиляции. Поскольку сервер базы данных использует то же значение по умолчанию, чаще всего указывать номер порта не нужно. Имя пользователя по умолчанию, как и имя базы данных по умолчанию, совпадает с именем пользователя в операционной системе. Заметьте, что просто так подключаться к любой базе данных под любым именем пользователя вы не сможете. Узнать о ваших правах можно у администратора баз данных.

Если значения по умолчанию не подходят, можно сэкономить на вводе параметров подключения, установив переменные среды PGDATABASE, PGHOST, PGPORT и/или PGUSER. Также удобно иметь файл ~/.pgpass, чтобы не вводить пароли снова и снова.

Альтернативный вариант указания параметров подключения – использование строки conninfo или URI вместо имени базы данных. Этот механизм даёт широкие возможности для управления соединением. Например:

```
$ psql «service=myervice sslmode=require»
```

```
$ psql postgresql://dbmaster:5433/mydb?sslmode=require
```

Этот способ также позволяет использовать LDAP для получения параметров подключения.

Если соединение не может быть установлено по любой причине (например, нет прав, сервер не работает и т.д.), psql вернет ошибку и прекратит работу.

Если и стандартный ввод, и стандартный вывод являются терминалом, то psql установит кодировку клиента в «auto», и подходящая клиентская кодировка будет определяться из локальных установок (переменная окружения LC_TYPE на Unix системах). Если это работает не так как ожидалось, кодировку клиента можно изменить, установив переменную окружения PGCLIENTENCODING.

17.5.2. Ввод SQL команд

Как правило, приглашение psql состоит из имени базы данных, к которой psql в данный момент подключен, а затем строки =>.

Пример.

```
$ psql testdb
```

```
psql (9.4.4)
```

```
Type «help» for help.
```

```
testdb=>
```

В командной строке пользователь может вводить команды SQL. Обычно введенные строки отправляются на сервер, когда встречается точка с запятой, завершающая команду. Конец строки не завершает команду. Это позволяет разбивать команды на несколько строк для лучшего понимания. Если команда была отправлена и выполнена без ошибок, то результат команды выводится на экран.

Если к базе данных, которая не приведена в соответствие шаблону безопасного использования схем, имеют доступ недоверенные пользователи, начинайте сеанс с удаления доступных им для записи схем из пути поиска (`search_path`). Для этого можно добавить `options=-csearch_path=` в строку подключения или выполнить команду `SELECT pg_catalog.set_config('search_path', '', false)` перед другими командами SQL. Это касается не только `psql`, но и любых других интерфейсов для выполнения произвольных SQL-команд.

При каждом выполнении команды `psql` также проверяет асинхронные уведомления о событиях, генерируемые командами `LISTEN` и `NOTIFY`.

Комментарии в стиле C передаются для обработки на сервер, в то время как комментарии в стандарте SQL `psql` удаляет перед отправкой.

17.5.3. Метакоманды

Все, что вводится в `psql` не взятое в кавычки и начинающееся с обратной косой черты, является мета-командой `psql` и обрабатывается самим `psql`. Эти команды делают `psql` полезным для задач администрирования и разработки скриптов.

Формат команды `psql` следующий: бэкслеш, сразу за ним команда, затем аргументы. Аргументы отделяются от команды и друг от друга любым количеством пробелов.

Чтобы включить пробел в значение аргумента, нужно заключить его в одинарные кавычки. Чтобы включить одинарную кавычку в значение аргумента,

нужно написать две одинарные кавычки внутри текста в одинарных кавычках. Всё, что содержится в одинарных кавычках подлежит заменам, принятым в языке C: \n (новая строка), \t (табуляция), \b (backspace), \r (возврат каретки), \f (подача страницы), \цифры (восьмеричное число), и \хцифры (шестнадцатеричное число). Если внутри текста в одинарных кавычках встречается обратная косая перед любым другим символом, то она экранирует этот символ.

Если внутри аргумента не в кавычках встречается имя переменной `psql` с предшествующим двоеточием (:), оно заменяется значением переменной, как описано в Интерполяция SQL. Также будут работать описанные там формы `: 'имя_переменной'` и `: "имя_переменной"`. Конструкция `: { ?имя_переменной }` позволяет проверить, определена ли переменная. Она заменяется значением TRUE или FALSE. Экранирование обратной косой чертой защищает двоеточие от замены.

Текст аргумента, заключённый в обратные кавычки (```), считается командной строкой, которая передаётся в командную оболочку ОС. Вывод от этой команды (с удалёнными в конце символами новой строки) заменяет текст в обратных кавычках. В содержимом этого текста не обрабатываются никакие спецпоследовательности или особые знаки, за исключением того, что все вхождения `:имя_переменной`, где `имя_переменной` – это имя переменной `psql`, заменяются значением этой переменной. Также вхождения `: 'имя_переменной'` заменяются значением переменной, заключённым в апострофы с тем, что это было одним аргументом команды оболочки. Последняя форма почти всегда более предпочтительна, если только вы не абсолютно точно знаете, чего ожидать в переменной. Так как символы перевода строки и возврата каретки могут быть надёжно экранированы не на всех платформах, форма `: 'имя_переменной'` выводит сообщение об ошибке и подстановка значения переменной не производится, когда это значение содержит такие символы.

Некоторые команды принимают идентификатор SQL (например, имя таблицы) в качестве аргумента. Такие аргументы следуют правилам синтаксиса SQL: буквы,

не взятые в кавычки, преобразуются в нижний регистр, буквы, взятые в двойные кавычки (") предотвращают преобразование регистра и позволяют включать пробелы в идентификатор. Внутри двойных кавычек две двойные кавычки сокращаются до одной. Например, `FOO"BAR"BAZ` интерпретируется как `fooBARbaz`, а `"A weird" name"` становится `A weird" name`.

Разбор аргументов останавливается в конце строки или когда встречается другая обратная косая черта, не внутри кавычек. Обратная косая не внутри кавычек рассматривается как начало новой метакоманды. Специальная последовательность `\\` (две обратных косых черты) обозначает окончание аргументов, далее продолжается разбор команд SQL, если таковые имеются. Таким образом, команды SQL и `psql` можно свободно смешивать в одной строке. Но в любом случае аргументы метакоманды не могут выходить за пределы текущей строки.

Многие из метакоманд оперируют с *буфером текущего запроса*. Этот буфер содержит произвольный текст команд SQL, который был введен, но ещё не отправлен серверу для выполнения. В него будут входить и предыдущие строки, а также текст, расположенный в строке метакоманды перед ней.

Определены следующие мета-команды:

`\a`

Если текущий режим вывода таблицы невыровненный, то он переключается на выровненный режим. Если текущий режим выровненный, то устанавливается невыровненный. Эта команда поддерживается для обратной совместимости.

`\c or \connect [dbname [username] [host] [port]]
| conninfo`

Устанавливает новое подключение к серверу Синергия-БД. Параметры подключения можно указывать как позиционно (в перечисленном порядке), так и используя строки соединения `conninfo`.

При использовании позиционных параметров, если какой-либо из параметров подключения `dbname`, `username`, `host` или `port` не указали, используется значение этого параметра от предыдущего соединения. Если нет предыдущего подключения, используются значения по умолчанию, принятые в `libpq`.

При использовании строк соединения `conninfo`, никакие из значений параметров предыдущего подключения использоваться не будут.

Если новое подключение успешно установлено, предыдущее подключение закрывается. Если попытка подключения не удалась (неверное имя пользователя, доступ запрещен, и т.д.), то предыдущее соединение останется активным, только если `psql` находится в интерактивном режиме. Если скрипт выполняется не интерактивно, обработка немедленно останавливается с сообщением об ошибке. Такое различие в поведении было выбрано для удобства пользователя в отношении опечаток с одной стороны и механизма безопасности, при котором скрипты не будут запущены на неправильной базе данных, с другой стороны.

Примеры:

```
=> \c mydb myuser host.dom 6432
=> \c service=foo
=> \c «host=localhost port=5432 dbname=mydb
connect_timeout=10 sslmode=disable»
=> \c
postgres://tom@localhost/mydb?application_name=myapp
\C [ title ]
```

Задаёт заголовок, который будет выводиться для результатов любых запросов или отменяет установленный ранее заголовок. Эта команда эквивалентна `\pset title title`. Название этой команды происходит от «caption», т.к. ранее это использовалось только для задания заголовков HTML таблиц.

```
\cd [ каталог ]
```

Заменяет текущий рабочий каталог на `directory`. Без аргумента устанавливается домашний каталог текущего пользователя.

Для печати текущего рабочего каталога используйте `\! pwd`.

```
\conninfo
```

Выводит информацию о текущем подключении к базе данных.

```
\copy { таблица [ ( column_list ) ] | ( запрос ) }
{ from | to } { 'filename' | program 'command' | stdin |
stdout | pstdin | pstdout } [ [ with ] ( параметр
[, ...] ) ]
```


Выполняет копирование на клиенте. Это операция, которая выполняет SQL команду COPY, но вместо чтения или записи в файл на сервере psql читает или записывает файл и пересылает данные между сервером и локальной файловой системой. Это означает, что для доступа к файлам используются привилегии локального пользователя, а не сервера, и не требуются привилегии суперпользователя SQL.

Когда указана program, psql выполняет command и данные из/в command передаются между сервером и клиентом. Это опять же означает, что для выполнения программ используются привилегии локального пользователя, а не сервера, и не требуются привилегии суперпользователя SQL.

При выполнении `\copy ... from stdin` строки с данными считываются из источника, выполнившего команду, и считываются до тех пор, пока не встретится `\.` или не будет достигнут конец файла. Эта опция полезна для заполнения таблиц прямо в SQL скриптах. При выполнении `\copy ... to stdout` вывод направляется в то же место, что и вывод psql команд. Статус команды COPY count не отображается, чтобы не перепутать со строкой данных. Для чтения/записи стандартного ввода/вывода psql, вне зависимости от источника текущей команды или опции `\o`, используйте `from pstdin` или `to pstdout`.

Синтаксис команды похож на синтаксис SQL команды COPY. Все опции, кроме источника и получателя данных, соответствуют опциям COPY. Поэтому к команде `\copy` применяются специальные правила разбора. В частности, не применяются правила подстановки переменных и экранирование при помощи бэкслеш.

Примечание. Эта операция не так эффективна, как SQL команда COPY, потому что все данные перемещаются между клиентом и сервером. Для больших объемов данных SQL команда может быть предпочтительнее.

`\copyright`

Показывает информацию об авторских правах и условиях распространения Синергия-БД.

```
\crosstabview [ столбВ [ столбГ [ столбТ
[ столбсортГ ] ] ] ]
```

Выполняет содержимое буфера текущего запроса (как \g) и показывает результат в виде перекрёстной таблицы. Заданный запрос должен возвращать минимум три столбца. Столбец результата, заданный параметром столбВ, будет образовывать вертикальные заголовки, а столбец, заданный параметром столбГ, – горизонтальные. Заданный параметром столбТ столбец будет поставлять данные для отображения внутри таблицы. Столбец, выбранный параметром столбсортГ, будет необязательным столбцом сортировки горизонтальных заголовков.

Каждое указание столбца может представлять собой имя или номер столбца (начиная с 1). К именам применяются обычные принятые в SQL правила учёта регистра и кавычек. По умолчанию в качестве столбВ подразумевается столбец 1, а в качестве столбГ – столбец 2. Если столбВ и столбГ задаются явно, они должны различаться. Если столбТ не задан, в результате запроса должно быть ровно три столбца, и в качестве столбТ выбирается столбец, отличный от столбВ и столбГ.

Вертикальный заголовок, выводимый в самом левом столбце, содержит значения из столбца столбВ, в том же порядке, в каком их возвращает запрос, но без дубликатов.

Горизонтальный заголовок, выводимый в первой строке, содержит значения из столбца столбГ, без дубликатов. По умолчанию они располагаются в том порядке, в каком их возвращает запрос. Но если задан необязательный аргумент столбсортГ, он определяет столбец, который должен содержать целые числа, и тогда значения из столбГ будут располагаться в горизонтальном заголовке по порядку значений в столбсортГ.

Внутри перекрёстной таблицы для каждого уникального значения x в столбГ и каждого уникального значения y в столбВ, ячейка, размещённая на пересечении (x, y) содержит значение столбТ в строке результата запроса, в которой значение столбГ равно x , а значение столбВ – y . Если такой строки не находится, ячейка остаётся пустой. Если же находится несколько таких строк, выдаётся ошибка.

\d[S+] [шаблон]

Для каждого отношения (таблицы, представления, индекса,

последовательности, внешней таблицы) или составного типа, соответствующих шаблону `pattern`, показывает столбцы, их типы, табличное пространство (если не по умолчанию) и любые специальные атрибуты, такие как `NOT NULL` или значения по умолчанию. Также показываются связанные индексы, ограничения, правила и триггеры. Для внешних таблиц показывается связанный внешний сервер.

Для некоторых типов отношений `\d` показывает дополнительную информацию по каждому столбцу: значения столбца для последовательностей, индексное выражение для индексов и параметры обработчика внешних данных для внешних таблиц.

Вариант команды `\d+` похож на `\d`, но выводит больше информации: комментарии к столбцам таблицы, наличие в таблице `OID`, для представления показывается его определение, отличные от значений по умолчанию установки `replica identity`.

По умолчанию отображаются только объекты, созданные пользователем. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`.

Примечание. Если `\d` используется без `pattern`, это эквивалентно `\dtvsE` и показывает список всех доступных таблиц, представлений, последовательностей и внешних таблиц. Чисто для удобства.

```
\d[S+] [ шаблон ]
```

Для каждого отношения (таблицы, представления, материализованного представления, индекса, последовательности, внешней таблицы) или составного типа, соответствующих шаблону, показывает все столбцы, их типы, табличное пространство (если оно изменено) и любые специальные атрибуты, такие как `NOT NULL` или значения по умолчанию. Также показываются связанные индексы, ограничения, правила и триггеры. Для сторонних таблиц также показывается связанный сторонний сервер.

Для некоторых типов отношений `\d` показывает дополнительную информацию по каждому столбцу: значения столбца для последовательностей, индексируемые выражения для индексов и параметры обёртки сторонних данных для сторонних таблиц.

Вариант команды `\d+` похож на `\d`, но выводит больше информации: комментарии к столбцам таблицы, наличие в таблице OID, для представления показывается его определение, отличные от значений по умолчанию установки `replica identity`.

По умолчанию отображаются только объекты, созданные пользователем. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`.

Примечание. Если `\d` используется без аргумента шаблон, эта команда удобства ради воспринимается как `\dtvmsE` и выдаёт список всех видимых таблиц, представлений, мат. представлений, последовательностей и сторонних таблиц.

```
\da[S] [ шаблон ]
```

Выводит список агрегатных функций вместе с типом возвращаемого значения и типами данных, которыми они оперируют. Если указан шаблон, показываются только те агрегатные функции, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`.

```
\dA[+] [ шаблон ]
```

Выводит список методов доступа. Если указан шаблон, показываются только те методы доступа, имена которых соответствуют ему. При добавлении `+` к имени команды для каждого метода доступа показывается его описание и связанная функция-обработчик.

```
\db[+] [ шаблон ]
```

Выводит список табличных пространств. Если указан шаблон, показываются только те табличные пространства, имена которых соответствуют ему. При добавлении `+` к имени команды для каждого объекта дополнительно выводятся параметры, объём на диске, права доступа и описание.

```
\dc[S+] [ шаблон ]
```

Выводит список преобразований между кодировками наборов символов. Если указан шаблон, показываются только те преобразования кодировок, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`. При добавлении `+` к имени команды для каждого объекта

дополнительно будет выводиться описание.

```
\dC[+] [ шаблон ]
```

Выводит список приведений типов. Если указан шаблон, показываются только те приведения типов, имена которых соответствуют ему. При добавлении + к имени команды для каждого объекта дополнительно будет выводиться описание.

```
\dd[S] [ шаблон ]
```

Показывает описания объектов следующих видов: ограничение, класс операторов, семейство операторов, правило и триггер. Описания остальных объектов можно посмотреть соответствующими метакомандами для этих типов объектов.

\dd показывает описания для объектов, соответствующих шаблону, или для доступных объектов указанных типов, если аргументы не заданы. Но в любом случае выводятся только те объекты, которые имеют описание. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор S.

Описания объектов создаются SQL-командой COMMENT.

```
\dD[S+] [ шаблон ]
```

Выводит список доменов. Если указан шаблон, показываются только те домены, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор S. При добавлении + к имени команды для каждого объекта дополнительно будут выводиться права доступа и описание.

```
\ddp [ шаблон ]
```

Выводит список прав доступа по умолчанию. Выводится строка для каждой роли (и схемы, если применимо), для которой права доступа по умолчанию отличаются от встроенных. Если указан шаблон, выводятся строки только для тех ролей и схем, имена которых соответствуют ему.

Права доступа по умолчанию устанавливаются командой ALTER DEFAULT PRIVILEGES. Смысл отображаемых привилегий объясняется в описании GRANT.

```
\dE[S+] [ шаблон ]
```

```
\di[S+] [ шаблон ]  
\dm[S+] [ шаблон ]  
\ds[S+] [ шаблон ]  
\dt[S+] [ шаблон ]  
\dv[S+] [ шаблон ]
```

В этой группе команд буквы E, i, m, s, t и v обозначают соответственно: внешнюю таблицу, индекс, материализованное представление, последовательность, таблицу и представление. Можно указывать все или часть этих букв, в произвольном порядке, чтобы получить список объектов этих типов. Например, \dit выводит список индексов и таблиц. При добавлении + к имени команды для каждого объекта дополнительно будут выводиться физический размер на диске и описание, при наличии. Если указан шаблон, выводятся только объекты, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор S.

```
\des[+] [ шаблон ]
```

Выводит список сторонних серверов (мнемоника: «external servers»). Если указан шаблон, выводятся только те серверы, имена которых соответствуют ему. Если используется форма \des+, то выводится полное описание каждого сервера, включая права доступа, тип, версию, параметры и описание.

```
\det[+] [ шаблон ]
```

Выводит список сторонних таблиц (мнемоника: «external tables»). Если указан шаблон, выводятся только те записи, имя таблицы или схемы которых соответствуют ему. Если используется форма \det+, то дополнительно выводятся общие параметры и описание сторонней таблицы.

```
\deu[+] [ шаблон ]
```

Выводит список сопоставлений пользователей (мнемоника: «external users»). Если указан шаблон, выводятся только сопоставления, в которых имена пользователей соответствуют ему. Если используется форма \deu+, то выводится дополнительная информация о каждом сопоставлении пользователей.

Примечание. \deu+ также может отображать имя и пароль удалённого пользователя, поэтому следует позаботиться о том, чтобы не раскрывать их.

`\dew[+] [ шаблон ]`

Выводит список обёрток сторонних данных (мнемоника: «external wrappers»). Если указан шаблон, выводятся только те обёртки сторонних данных, имена которых соответствуют ему. Если используется форма `\dew+`, то дополнительно выводятся права доступа, параметры и описание обёртки.

`\df[anptwS+] [ шаблон ]`

Выводит список функций с типами данных их результатов, аргументов и классификацией: «agg» (агрегатная), «normal», (обычная), «procedure» (процедурная), «trigger» (триггерная) или «window» (оконная). Чтобы получить функции только определённого вида (видов), добавьте в команду соответствующие буквы a, n, p, t или w. Если задан шаблон, показываются только те функции, имена которых соответствуют ему. По умолчанию показываются только функции, созданные пользователями; для включения системных объектов нужно задать шаблон или добавить модификатор S. Если используется форма `\df+`, то дополнительно выводятся характеристики каждой функции: изменчивость, допустимость распараллеливания, владелец, классификация по безопасности, права доступа, язык, исходный код и описание.

Примечание. Чтобы найти функции с аргументами или возвращаемыми значениями определённого типа данных, воспользуйтесь имеющейся в постраничнике возможностью поиска в выводе `\df`.

`\dF[+] [ шаблон ]`

Выводит список конфигураций текстового поиска. Если указан шаблон, показываются только те конфигурации, имена которых соответствуют ему. Если используется форма `\dF+`, то выводится полное описание для каждой конфигурации, включая базовый синтаксический анализатор и используемые словари для каждого типа фрагментов.

`\dFd[+] [ шаблон ]`

Выводит список словарей текстового поиска. Если указан шаблон, показываются только словари, имена которых соответствуют ему. Если используется форма `\dFd+`, то выводится дополнительная информация о каждом словаре,

включая базовый шаблон текстового поиска и параметры инициализации.

`\dFp[+] [ шаблон ]`

Выводит список анализаторов текстового поиска. Если указан шаблон, показываются только те анализаторы, имена которых соответствуют ему. Если используется форма `\dFp+`, то выводится полное описание для каждого анализатора, включая базовые функции и список типов фрагментов.

`\dFt[+] [ шаблон ]`

Выводит список шаблонов текстового поиска. Если указан шаблон, показываются только шаблоны, имена которых соответствуют ему. Если используется форма `\dFt+`, то выводится дополнительная информация о каждом шаблоне, включая имена основных функций.

`\dg[S+] [ шаблон ]`

Выводит список ролей базы данных. Так как понятия «пользователи» и «группы» были объединены в «роли», эта команда теперь эквивалентна `\du`. По умолчанию выводятся только роли, созданные пользователями: чтобы увидеть и системные роли, добавьте модификатор S. Если указан шаблон, выводятся только те роли, имена которых соответствуют ему. Если используется форма `\dg+`, то выводится дополнительная информация о каждой роли; в настоящее время это комментарий роли.

`\dl`

Это псевдоним для `\lo_list`, показывает список больших объектов.

`\dL[S+] [ шаблон ]`

Выводит список процедурных языков. Если указан шаблон, выводятся только те языки, имена которых соответствуют ему. По умолчанию показываются только языки, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор S. При добавлении + к имени команды для каждого языка дополнительно будут выводиться: обработчик вызова, функция проверки, права доступа и является ли язык системным объектом.

`\dn[S+] [ шаблон ]`

Выводит список схем (пространств имён). Если указан шаблон, выводятся

только те схемы, имена которых соответствуют ему. По умолчанию показывааются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор S. При добавлении + к имени команды для каждого объекта дополнительно будут выводиться права доступа и описание, при наличии.

```
\do[S+] [ шаблон ]
```

Выводит список операторов, их операндов и типы результата. Если указан шаблон, выводятся только те операторы, имена которых соответствуют ему. По умолчанию показывааются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор S. При добавлении + к имени команды для каждого оператора будет выводиться дополнительная информация, сейчас это имя функции, на которой основан оператор.

```
\dO[S+] [ шаблон ]
```

Выводит список правил сортировки. Если указан шаблон, выводятся только те правила, имена которых соответствуют ему. По умолчанию показывааются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор S. При добавлении + к имени команды для каждого объекта дополнительно будет выводиться описание, при наличии. Обратите внимание, что отображаются только правила сортировки, применимые к кодировке текущей базы данных, поэтому результат команды может отличаться для различных баз данных этой же установки Синергия-БД.

```
\dp [ шаблон ]
```

Выводит список таблиц, представлений и последовательностей с их правами доступа. Если указан шаблон, отображаются только таблицы, представления и последовательности, имена которых соответствуют ему.

Для установки прав доступа используются команды GRANT и REVOKE.

```
\drds [ шаблон-ролей [ шаблон-баз ] ]
```

Выводит список специфических параметров конфигурации. Эти параметры могут быть специфическими для роли, специфическими для базы данных, или обеих. Параметры шаблон-ролей и шаблон-баз используются для отбора

определённых ролей и баз данных, соответственно. Если они опущены, или указано *, выводятся все параметры конфигурации, в том числе не относящиеся к ролям или базам данных.

Команды ALTER ROLE и ALTER DATABASE используются для определения параметров конфигурации, специфических для роли или базы данных.

`\dRp[+] [ шаблон ]`

Выводит список реплицируемых публикаций. Если указан шаблон, выводятся только те подписки, имена которых соответствуют ему. При добавлении + к имени команды для каждой публикации показываются также связанные с ней таблицы.

`\dRs[+] [ шаблон ]`

Выводит список подписок на репликацию. Если указан шаблон, выводятся только те подписки, имена которых соответствуют ему. При добавлении + к имени команды выводятся дополнительные свойства подписок.

`\dT[S+] [ шаблон ]`

Выводит список типов данных. Если указан шаблон, выводятся только те типы, имена которых соответствуют ему. При добавлении + к имени команды для каждого типа данных дополнительно будет выводиться: внутреннее имя типа, размер, допустимые значения для типа enum и права доступа. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор S.

`\du[S+] [ шаблон ]`

Выводит список ролей базы данных. Так как понятия «пользователи» и «группы» были объединены в «роли», эта команда теперь равнозначна `\dg`. По умолчанию выводятся только роли, созданные пользователями: чтобы увидеть и системные роли, добавьте модификатор S. Если указан шаблон, выводятся только те роли, имена которых соответствуют ему. Если используется форма `\du+`, то выводится дополнительная информация о каждой роли; в настоящее время это комментарий роли.

`\dx[+] [ шаблон ]`

Выводит список установленных расширений. Если указан шаблон, выводятся

только расширения, имена которых соответствуют ему. Если используется форма `\dx+`, то для каждого расширения выводятся все принадлежащие ему объекты.

`\dy[+] [ шаблон ]`

Выводит список событийных триггеров. Если указан шаблон, выводятся только те событийные триггеры, имена которых соответствуют ему. При добавлении `+` к имени команды для каждого объекта дополнительно будет выводиться описание.

`\e или \edit [имя_файла] [номер_строки]`

Если указано имя_файла, файл открывается для редактирования; после выхода из редактора содержимое файла копируется в буфер текущего запроса. Если имя_файла не задано, буфер текущего запроса копируется во временный файл, который затем редактируется тем же образом. Либо, если буфер текущего запроса пуст, во временный файл копируется последний выполненный запрос, и он затем так же редактируется.

Новое содержимое буфера запроса затем разбирается согласно обычным правилам `psql`, при этом весь буфер обрабатывается как одна строка. Все законченные запросы немедленно выполняются; то есть, если буфер запроса содержит точку с запятой или заканчивается ей, выполняется всё его содержимое до этого знака. Оставшийся текст будет ждать выполнения в буфере запроса; вы можете ввести точку с запятой или `\g`, чтобы передать его, либо `\r`, чтобы сбросить его, очистив буфер запроса. Прочтение буфера как одной строки в основном отражается на метакомандах: всё, что находится в буфере после метакоманды, будет воспринято как аргументы метакоманды, даже если этот текст продолжается на нескольких строках. Поэтому таким способом нельзя выполнять скрипты с метакомандами. Для таких скриптов используйте `\i`.

Если указан номер строки, `psql` будет позиционировать курсор на указанную строку файла или буфера запроса. Обратите внимание, что если указан один аргумент и он числовой, `psql` предполагает, что это номер строки, а не имя файла.

`\echo текст [ ... ]`

Выводит аргументы на стандартный вывод, разделяя их одним пробелом, в конце следует перевод строки. Команда полезна для формирования вывода из

скриптов, например:

```
=> \echo `date`
```

```
Tue Oct 26 21:40:57 CEST 1999
```

Если первый аргумент -n без кавычек, то перевод строки в конце не ставится.

```
\ef [описание_функции [номер_строки]]
```

Эта команда извлекает из базы определение заданной функции или процедуры в форме команды CREATE OR REPLACE FUNCTION или CREATE OR REPLACE PROCEDURE и открывает его на редактирование. Редактирование осуществляется таким же образом, как и при выполнении \edit. После выхода из редактора изменённая команда остаётся в буфере запроса; введите точку с запятой или \g, чтобы отправить её серверу, либо \r, чтобы отменить.

Для функции может быть задано только имя или имя и аргументы, например foo(integer, text). Типы аргументов необходимы, если существует более чем одна функция с тем же именем.

Если функция не указана, для редактирования открывается пустая заготовка CREATE FUNCTION.

Если указан номер строки, psql будет позиционировать курсор на указанную строку тела функции. Обратите внимание, что тело функции обычно не начинается на первой строке файла.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы \ef, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

```
\encoding [ кодировка ]
```

Устанавливает кодировку набора символов на клиенте. Без аргумента команда показывает текущую кодировку.

```
\errverbose
```

Повторяет последнее серверное сообщение об ошибке с максимальным уровнем детализации, как если бы переменная VERBOSITY имела значение verbose, а SHOW_CONTEXT – always.

```
\ev [имя_представления [номер_строки]]
```

Эта команда извлекает и открывает на редактирование определение указанного представления в форме команды `CREATE OR REPLACE VIEW`. Редактирование осуществляется так же, как и с `\edit`. После выхода из редактора изменённая команда остаётся в буфере запроса; введите точку с запятой или `\g`, чтобы выполнить её, либо `\r`, чтобы её сбросить.

Если представление не указано, для редактирования открывается пустая заготовка `CREATE VIEW`.

Если указан номер строки, `psql` установит курсор на заданную строку в определении представления.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы `\ev`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

`\f [ строка ]`

Устанавливает разделитель полей для невыровненного режима вывода запросов. По умолчанию используется вертикальная черта (`|`). Равнозначно команде `\pset fieldsep`.

`\g [ имя_файла ]`

`\g [ | команда ]`

Отправляет содержимое буфера текущего запроса на сервер для выполнения. Если передаётся аргумент, вывод запроса записывается в указанный файл или передаётся через поток заданной команде оболочки, а не отображается как обычно. Вывод направляется в файл или команду, только если запрос успешно вернул 0 или более строк, но не когда запрос завершился неудачно или выполнялась команда, не возвращающая данные.

Если буфер текущего запроса пуст, вместо этого повторно выполняется предыдущий запрос. За исключением этой особенности, метакоманда `\g` без аргумента по сути равнозначна точке с запятой. Метакоманда `\g` с аргументом является «одноразовой» альтернативой команде `\o`.

Если аргумент начинается с `|`, весь остаток строки воспринимается как команда, подлежащая выполнению, в которой не производится ни подстановка

переменных, ни раскрытие обратных кавычек. Это продолжение строки просто передаётся оболочке в буквальном виде.

```
\gdesc
```

Показывает описание (то есть имена и типы данных столбцов) результата текущего запроса в буфере. Сам запрос при этом не выполняется; но если он содержит какие-либо синтаксические ошибки, они выдаются обычным образом.

Если буфер текущего запроса пуст, будет повторно описан последний переданный запрос.

```
\gexec
```

Отправляет буфер текущего запроса на сервер, а затем обрабатывает содержимое каждого столбца каждой строки результата запроса (если он непустой) как SQL-оператор, то есть исполняет его. Например, следующая команда создаст индексы по каждому столбцу `my_table`:

```
=> SELECT format('create index on my_table(%I)',
attname)
-> FROM pg_attribute
-> WHERE attrelid = 'my_table'::regclass AND attnum > 0
-> ORDER BY attnum
-> \gexec
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
```

Генерируемые запросы выполняются в том порядке, в каком возвращаются строки, слева направо, если возвращается несколько столбцов. Поля `NULL` игнорируются. Эти запросы передаются для обработки на сервер буквально, так что это не могут быть метакоманды `psql` или запросы, использующие переменные `psql`. В случае сбоя в одном из запросов, выполнение оставшихся запросов продолжается, если только не установлена переменная `ON_ERROR_STOP`. На выполнение каждого запроса оказывает влияние параметр `ECHO`. Применяя команду `\gexec`, рекомендуется устанавливать в `ECHO` режим `all` или `queries`. Такие расширенные

средства, как протоколирование запросов, пошаговый режим, замер времени и т. п., так же действуют при выполнении каждого генерируемого запроса.

Если буфер текущего запроса пуст, будет повторно выполнен последний переданный запрос.

```
\gset [ префикс ]
```

Отправляет буфер текущего запроса на сервер для выполнения и сохраняет результат запроса в переменных `psql`. Выполняемый запрос должен возвращать ровно одну строку. Каждый столбец строки результата сохраняется в отдельной переменной, которая называется так же, как и столбец, например:

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset
=> \echo :var1 :var2
hello 10
```

Если указан префикс, то он добавляется в начале к именам переменных:

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset result_
=> \echo :result_var1 :result_var2
hello 10
```

Если значение столбца `NULL`, то вместо присвоения значения соответствующая переменная удаляется.

Если запрос завершается ошибкой или не возвращает одну строку, то никакие переменные не меняются.

Если буфер текущего запроса пуст, будет повторно выполнен последний переданный запрос.

```
\gx [ имя_файла ]
\gx [ |команда ]
```

Метакоманда `\gx` подобна `\g`, но принудительно включает расширенный режим вывода для текущего запроса.

```
\h или \help [ команда ]
```

Выводит подсказку по синтаксису указанной команды SQL. Если команда не указана, то `psql` выводит список всех команд, для которых доступна справка. Если в

качестве `command` указана звёздочка (*), то выводится справка по всем командам SQL.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы `\help`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

Примечание. Для упрощения ввода команды, состоящие из нескольких слов, можно не заключать в кавычки. Таким образом, можно просто писать `\help alter table`.

`\H` или `\html`

Включает вывод запросов в формате HTML. Если формат HTML уже включён, происходит переключение обратно на выровненный формат. Эта команда используется для совместимости и удобства, но в описании `\pset` вы можете узнать о других вариантах вывода.

`\i` или `\include` имя_файла

Читает ввод из файла `имя_файла` и выполняет его, как будто он был набран на клавиатуре.

Если `имя_файла` задано как - (минус), читается стандартный ввод до признака конца файла или до метакоманды `\q`. Это может быть полезно для совмещения интерактивного ввода со вводом команд из файлов. Заметьте, что при этом поведение `Readline` будет применяться, только если оно активно на внешнем уровне.

Примечание. Если вы хотите видеть строки файла на экране по мере их чтения, необходимо установить для переменной `ЕCHO` значение `all`.

`\if` выражение
`\elif` выражение
`\else`
`\endif`

Эта группа команд реализует вложенные условные блоки. Условный блок должен начинаться командой `\if` и заканчиваться `\endif`. Между ними может быть любое количество предложений `\elif`, которые могут быть дополнительно завершаться одним предложением `\else`. Между командами, формирующими блок условия, могут размещаться (и обычно размещаются) обычные запросы и другие типы команд в обратных кавычках.

Команды `\if` и `\elif` считывают свои аргументы и вычисляют их как логические выражения. Если выражение выдаёт `true`, обработка продолжается как обычно; в противном случае входные строки пропускаются до достижения соответствующих команд `\elif`, `\else` или `\endif`. Как только проверка `\if` или `\elif` оказывается успешной, аргументы последующих команд `\elif` в том же блоке не вычисляются, а считаются ложными. Строки, следующие за `\else`, обрабатываются только если ни одна из предыдущих проверок `\if` или `\elif` не была успешной.

В аргументе выражение команд `\if` и `\elif` производится подстановка переменных и раскрытие кавычек, как и в аргументе любой другой команды с обратной косой. После этого полученное значение оценивается как значение переменной да/нет. Так что истинным значением будет любое однозначное вхождение без учёта регистра одной из строк (или подстрок): `true`, `false`, `1`, `0`, `on`, `off`, `yes`, `no`. Например, строки `t`, `T` и `tR` все будут восприниматься как `true`.

Если выражения не приводятся к значениям `true` или `false`, будет выдано предупреждение, а их результат будет считаться ложным.

Пропускаемые строки разбираются как обычно (в них выявляются запросы и команды с обратной косой), но не передаются серверу, а команды с обратной косой, отличные от условных (`\if`, `\elif`, `\else`, `\endif`), просто игнорируются. В командах условий проверяется только правильность вложенности. Ссылки на переменные в пропускаемых строках не разворачиваются, как не выполняется и раскрытие обратных кавычек.

Все команды с обратной косой в одном условном блоке должны содержаться в одном исходном файле. Если до того, как будут закрыты все локальные блоки `\if`, в основном файле команд будет достигнут конец файла или встретится включение другого файла (команда `\include`), `psql` выдаст ошибку.

Например:

```
-- проверка существования двух отдельных записей в базе
данных и сохранение
-- результатов в двух разных переменных psql
```

```

SELECT
    EXISTS(SELECT 1 FROM customer WHERE customer_id =
123) as is_customer,
    EXISTS(SELECT 1 FROM employee WHERE employee_id =
456) as is_employee
\gset
\if :is_customer
    SELECT * FROM customer WHERE customer_id = 123;
\elif :is_employee
    \echo 'is not a customer but is an employee'
    SELECT * FROM employee WHERE employee_id = 456;
\else
    \if yes
        \echo 'not a customer or employee'
    \else
        \echo 'this will never print'
    \endif
\endif
\ir или \include_relative имя_файла

```

Команда `\ir` похожа на `\i`, но по-разному интерпретирует относительные имена файлов. При выполнении в интерактивном режиме две команды ведут себя одинаково. Однако, при вызове из скрипта `\ir` интерпретирует имена файлов относительно каталога, в котором расположен скрипт, а не текущего рабочего каталога.

```
\l[+] или \list[+] [ шаблон ]
```

Выводит список баз данных на сервере и показывает их имена, владельцев, кодировку набора символов и права доступа. Если указан шаблон, выводятся только базы данных, имена которых соответствуют ему. При добавлении `+` к имени команды также отображаются: размер базы данных, табличное пространство по умолчанию и описание. Информация о размере доступна только для баз данных, к которым текущий пользователь может подключиться.

```
\lo_export oid_БО имя_файла
```

Читает большой объект с заданным `oid_БО` из базы данных и записывает его в файл `имя_файла`. Обратите внимание, что это несколько отличается от серверной функции `lo_export`, действующей с правами пользователя, от имени которого работает сервер базы данных, и в файловой системе сервера.

Примечание. Используйте `\lo_list` для получения OID больших объектов.

`\lo_import имя_файла [ комментарий ]`

Сохраняет файл в большом объекте Синергия-БД. При этом с объектом может быть связан указанный комментарий.

Пример.

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a
picture of me'
lo_import 152801
```

Ответ указывает на то, что большой объект получил OID 152801, который может быть использован для доступа к вновь созданному объекту в будущем. Для удобства чтения рекомендуется всегда связывать объекты с понятными комментариями. OID и комментарии можно посмотреть с помощью команды `\lo_list`.

Обратите внимание, что это немного отличается от функции сервера `lo_import`, так как действует от имени локального пользователя в локальной файловой системе, а не пользователя сервера в файловой системе сервера.

`\lo_list`

Показывает список всех больших объектов Синергия-БД, хранящихся в базе данных, вместе с предоставленными комментариями.

`\lo_unlink oid_БО`

Удаляет большой объект с `oid_БО` из базы данных.

Примечание. Используйте `\lo_list` для получения OID больших объектов.

`\o или \out [ имя_файла ]`
`\o или \out [ |команда ]`

Результаты запросов будут сохраняться в файле `имя_файла` или перенаправляться команде оболочки (заданной аргументом команда). Если аргумент

не указан, результаты запросов перенаправляются на стандартный вывод.

Если аргумент начинается с |, весь остаток строки воспринимается как команда, подлежащая выполнению, в которой не производится ни подстановка переменных, ни раскрытие обратных кавычек. Это продолжение строки просто передаётся оболочке в буквальном виде.

«Результаты запросов» включают в себя все таблицы, ответы команд, уведомления, полученные от сервера баз данных, а также вывод от метакоманд, обращающихся к базе (таких как \d), но не сообщения об ошибках.

Примечание. Чтобы вставить текст между результатами запросов, используйте \qecho.

```
\p или \print
```

Печатает содержимое буфера текущего запроса на стандартный вывод. Если этот буфер пуст, будет напечатан последний выполненный запрос.

```
\password [ имя_пользователя ]
```

Изменяет пароль указанного пользователя (по умолчанию, текущего пользователя). Эта команда запрашивает новый пароль, шифрует и отправляет его на сервер в виде команды ALTER ROLE. Это гарантирует, что новый пароль не отображается в открытом виде в истории команд, журнале сервера или в другом месте.

```
\prompt [ текст ] имя
```

Предлагает пользователю ввести значение, которое будет присвоено переменной имя. Дополнительно можно указать текст подсказки. Если подсказка состоит из нескольких слов, то её текст нужно взять в одинарные кавычки.

По умолчанию, \prompt использует терминал для ввода и вывода. Однако, если используется параметр командной строки -f, \prompt использует стандартный ввод и стандартный вывод.

```
\pset [ параметр [ значение ] ]
```

Эта команда устанавливает параметры, влияющие на вывод результатов запросов. Указание параметр определяет, какой параметр требуется установить. Семантика значения зависит от выбранного параметра. Для некоторых параметров

отсутствие значения означает переключение значения, либо сброс значения, как описано ниже в разделе конкретного параметра. Если такое поведение не упоминается, то пропуск значения приводит к отображению текущего значения параметра.

`\pset` без аргументов выводит текущий статус всех параметров команды.

Имеются следующие параметры:

`border`

Здесь значение должно быть числом. В целом, чем больше это число, тем больше границ и линий будет в таблицах, но детали зависят от формата. В формате HTML заданное значение напрямую отображается в атрибут `border =...`. Для большинства других форматов имеют смысл только значения 0 (нет границы), 1 (внутренние разделительные линии) и 2 (граница таблицы), а значения больше 2 воспринимаются как `border = 2`. Форматы `latex` и `latex-longtable` дополнительно поддерживают значение 3, добавляющее разделительные линии между строками данных.

`columns`

Устанавливает максимальную ширину для формата `wrapped`, а также ограничение по ширине, свыше которого будет требоваться постраничник или произойдёт переключение в вертикальное отображение при режиме `expanded auto`. При значении 0 (по умолчанию) максимальная ширина управляется переменной среды `COLUMNS` или шириной экрана, если `COLUMNS` не установлена. Кроме того, если `columns` равно нулю, то формат `wrapped` влияет только на вывод на экран. Если `columns` не равно 0, то это также влияет на вывод в файл или в другую команду через канал.

`expanded` (или `x`)

Указанное значение допускает варианты `on` или `off`, которые включают или отключают развёрнутый режим, либо `auto`. Если значение опущено, команда включает/отключает режим. Когда развёрнутый режим включён, результаты запроса выводятся в две колонки: имя столбца в левой, данные в правой. Этот режим полезен, если данные не помещаются на экране в обычном «горизонтальном»

режиме. При выборе auto развёрнутый режим используется, когда результат запроса содержит несколько столбцов и по ширине не умещается на экране; в противном случае используется обычный режим. Режим auto распространяется только на форматы `aligned` и `wrapped`. С другими форматами он всегда равнозначен отключённому состоянию.

`fieldsep`

Устанавливает разделитель полей для невыровненного режима вывода запросов. Таким образом, можно формировать вывод, в котором значения будут разделены табуляцией или запятыми. Это может быть предпочтительным для использования в других программах. Для установки символа табуляции в качестве разделителя полей, введите `\pset fieldsep '\t'`. По умолчанию используется вертикальная черта (`|`).

`fieldsep_zero`

Устанавливает разделитель полей для невыровненного режима вывода в нулевой байт.

`footer`

Возможны два варианта значения: `on` или `off`, которые включают или отключают вывод результирующей строки с количеством выбранных записей (`n` строк). Если значение не указано, то команда переключает текущее значение в `on` или `off`.

`format`

Устанавливает один из следующих форматов вывода: `unaligned`, `aligned`, `wrapped`, `html`, `asciidoc`, `latex` (использует `tabular`), `latex-longtable` или `troff-ms`. Допускается сокращение слова до уникального значения.

В формате `unaligned` все столбцы размещаются на одной строке и отделяются друг от друга разделителем полей. Это полезно для создания вывода, который будет читаться другими программами (например, формат с выводом значений, разделённых запятыми или табуляцией).

Формат `aligned` это стандартный, удобочитаемый, хорошо отформатированный текстовый вывод. Используется по умолчанию.

Формат `wrapped` похож на `aligned`, но переносит длинные значения столбцов на новые строки, чтобы общий вывод поместился в заданную ширину. Задание ширины вывода описано в параметре `columns`. Обратите внимание, что `psql` не будет пытаться переносить на новые строки заголовки столбцов. Поэтому формат `wrapped` работает так же, как `aligned` если общая ширина, требуемая для всех заголовков столбцов, превышает установленную максимальную ширину.

Форматы `html`, `asciidoc`, `latex`, `latex-longtable` и `troff-ms` выводят таблицы, которые предназначены для включения в документы с помощью соответствующего языка разметки. Они не являются полноценными документами! Возможно это необязательно в HTML, но в LaTeX необходимо иметь полный упаковщик документа. `latex-longtable` также требует наличия пакетов LaTeX `longtable` и `booktabs`.

`linestyle`

Задаёт стиль отрисовки линий границы: `ascii`, `old-ascii` или `unicode`. Допускается сокращение слова до уникального значения. Это значит, что одной буквы будет достаточно. Значение по умолчанию: `ascii`. Этот параметр действует только в форматах `aligned` и `wrapped`.

Стиль `ascii` использует обычные символы ASCII. Символы новой строки в данных показываются с использованием символа `+` в правом поле. Когда при формате `wrapped` происходит перенос данных на новую строку (без символа новой строки), ставится точка `(.)` в правом поле первой строки и точка в левом поле следующей строки.

Символы новой строки в данных отображаются, используя символ `:` вместо левого разделителя полей. Когда происходит перенос данных на новую строку без символа новой строки, символ `;` используется вместо левого разделителя полей.

Стиль `unicode` использует символы Юникода для рисования линий. Символы новой строки в данных показываются с использованием символа возврата каретки в правом поле. Когда при формате `wrapped` происходит перенос данных на новую строку (без символа новой строки), ставится символ многоточия в правом поле первой строки и в левом поле следующей строки.

Когда значение `border` больше нуля, параметр `linestyle` также определяет

символы, которыми будут рисоваться границы. Обычные символы ASCII работают везде, но символы Юникода смотрятся лучше на терминалах, распознающих их.

`null`

Устанавливает строку, которая будет напечатана вместо значения `null`. По умолчанию не печатается ничего, что можно ошибочно принять за пустую строку. Например, можно было бы предпочесть `\pset null '(null)'`.

`numericlocale`

Если задаётся значение, возможны два варианта: `on` или `off`, которые включают или отключают отображение специфичного для локали символа, разделяющего группы цифр левее десятичной точки. Если значение не указано, то команда переключает вывод чисел с локализованного на обычный и обратно.

`pager`

Управляет использованием постраничника для просмотра результатов запросов и справочной информации `psql`. Если установлена переменная среды `PSQL_PAGER` или `PAGER`, вывод передаётся указанной программе. В противном случае используется платформозависимая программа по умолчанию (например, `more`).

Если `pager` имеет значение `off`, программа постраничного просмотра (постраничник) не используется. Если `pager` имеет значение `on`, эта программа используется при необходимости, т. е. когда вывод на терминал не помещается на экране. Параметр `pager` также может иметь значение `always`, при этом постраничник будет использоваться всегда, независимо от того, помещается вывод на экран терминала или нет. Команда `\pset pager` без указания значения переключает варианты `on` и `off`.

`pager_min_lines`

Если в `pager_min_lines` задаётся число, превышающее высоту страницы, программа постраничного вывода не будет вызываться, пока не наберётся заданное число строк для вывода. Значение по умолчанию – 0.

`recordsep`

Устанавливает разделитель записей (строк) для невыровненного режима

вывода. По умолчанию используется символ новой строки.

`recordsep_zero`

Устанавливает разделитель записей для невыровненного режима вывода в нулевой байт.

`tableattr` (или `T`)

Устанавливает атрибуты, которые будут помещены в тег `table`, при формате вывода HTML. Например, это может быть `cellpadding` или `border`. Заметьте, что, вероятно, не нужно здесь задавать `border`, так как для этого уже есть `\pset border`. Если значение не задано, атрибуты таблицы удаляются.

В формате `latex-longtable` этот параметр контролирует пропорциональную ширину каждого столбца, данные которого выровнены по левому краю. Он указывается как список разделённых пробелами значений, например `'0.2 0.2 0.6'`. Для неуказанных столбцов используется последнее из заданных значений.

`title` (или `C`)

Устанавливает заголовок таблицы для любых впоследствии выводимых таблиц. Это можно использовать для задания описательных тегов при формировании вывода. Если значение не задано, заголовок таблицы удаляется.

`tuples_only` (или `t`)

Возможны два варианта значения: `on` или `off`, которые включают или отключают режим вывода только кортежей. Если значение не указано, то команда переключает с режима вывода только кортежей на обычный режим и обратно. Обычный вывод включает в себя дополнительную информацию, такую как заголовки столбцов и различные колонтитулы. В режиме вывода только кортежей отображаются только фактические табличные данные.

`unicode_border_linestyle`

Устанавливает стиль рисования границ для стиля линий `unicode: single` (одинарный) или `double` (двойной).

`unicode_column_linestyle`

Устанавливает стиль рисования колонок для стиля линий `unicode: single` (одинарный) или `double` (двойной).

`unicode_header_linestyle`

Устанавливает стиль рисования заголовка для стиля линий `unicode: single` (одинарный) или `double` (двойной).

Для некоторых параметров `\pset` есть короткие команды.

`\q` или `\quit`

Выход из `psql`. При использовании в скрипте прекращается только выполнение этого скрипта.

`\qecho` текст [...]

Эта команда идентична `\echo` за исключением того, что вывод будет записываться в канал вывода запросов, установленный `\o`.

`\r` или `\reset`

Сбрасывает (очищает) буфер запроса.

`\s` [имя_файла]

Записывает историю команд `psql` в файл `имя_файла`. Если `имя_файла` не указано, то история команд выводится на стандартный вывод (с использованием постраничника, когда уместно). Этот параметр недоступен, если `psql` был собран без поддержки Readline.

`\set` [имя [значение [...]]]

Задаёт для переменной `psql` имя указанное значение или, если указано несколько значений, все эти значения, соединённые вместе. Если присутствует только один аргумент, значением переменной становится пустая строка. Для сброса переменной используйте команду `\unset`.

`\set` без аргументов выводит имена и значения всех `psql` переменных, установленных в настоящее время.

Имена переменных могут содержать буквы, цифры и знаки подчёркивания. Имена переменных чувствительны к регистру.

Некоторые переменные имеют особенности в том, как они управляют поведением `psql` или автоматически устанавливаются, отражая состояние соединения.

Примечание. Эта команда не имеет отношения к SQL-команде `SET`.

`\setenv имя [ значение ]`

Задаёт для переменной среды имя значение или, если значение не задано, удаляет переменную среды.

Пример.

```
testdb=> \setenv PAGER less
testdb=> \setenv LESS -imx4F
\sف[+] описание_функции
```

Эта команда извлекает из базы и выводит определение заданной функции или процедуры в форме команды CREATE OR REPLACE FUNCTION или CREATE OR REPLACE PROCEDURE. Определение выдаётся в текущий канал вывода запроса, установленный \o.

Для функции может быть задано только имя или имя и аргументы, например `foo(integer, text)`. Типы аргументов необходимы, если существует более чем одна функция с тем же именем.

При добавлении + к имени команды строки вывода нумеруются, первая строка тела функции получит номер 1.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы \sf, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

`\sv[+] имя_представления`

Извлекает из базы данных и выводит определение указанного представления в форме команды CREATE OR REPLACE VIEW. Определение выводится в текущий канал вывода запросов, установленный \o.

При добавлении + к имени команды строки вывода нумеруются, начиная с 1.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы \sv, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

`\t`

Включает/выключает отображение имён столбцов и результирующей строки с количеством выбранных записей для запросов. Эта команда эквивалентна \pset

`tuples_only` и предоставлена для удобства.

`\T` параметры_таблицы

Устанавливает атрибуты, которые будут помещены в тег `table` при формате вывода HTML. Эта команда эквивалентна `\pset tableattr` параметры_таблицы.

`\timing` [on | off]

С параметром данная команда, в зависимости от него, включает/отключает отображение времени выполнения каждого SQL-оператора. Без параметра она меняет состояние отображения на противоположное. Время выводится в миллисекундах; интервалы больше 1 секунды выводятся в формате минуты:секунды, а при необходимости в вывод также добавляются часы и дни.

`\unset` имя

Удаляет `psql` переменную имя.

Большинство переменных, управляющих поведением `psql`, нельзя сбросить; команда `\unset` для них воспринимается как установка значений по умолчанию.

`\w` или `\write` имя_файла

`\w` или `\write` |команда

Выводит буфер текущего запроса в файл `имя_файла` или через канал в команду оболочки команда. Если этот буфер пуст, будет выведен последний выполненный запрос.

Если аргумент начинается с `|`, весь остаток строки воспринимается как команда, подлежащая выполнению, в которой не производится ни подстановка переменных, ни раскрытие обратных кавычек. Это продолжение строки просто передаётся оболочке в буквальном виде.

`\watch` [секунды]

Эта команда многократно выполняет текущий запрос в буфере (как `\g`), пока не будет прервана или не возникнет ошибка. Аргумент задаёт количество секунд ожидания между выполнениями запроса (по умолчанию 2). Результат каждого запроса выводится с заголовком, включающим строку `\pset title` (если она задана), время запуска запроса и интервал задержки.

Если буфер текущего запроса пуст, будет повторно выполнен последний переданный запрос.

`\x [ on | off | auto ]`

Устанавливает или переключает режим развёрнутого вывода таблицы. Это эквивалентно `\pset expanded`.

`\z [ шаблон ]`

Выводит список таблиц, представлений и последовательностей с их правами доступа. Если указан шаблон, отображаются только таблицы, представления и последовательности, имена которых соответствуют ему.

Это псевдоним для `\dp` («показать права доступа»).

`\! [ команда ]`

Без аргументов запускает подчинённую оболочку; когда эта оболочка завершается, `psql` продолжает работу. Если добавлен аргумент, запускает команду оболочки команда.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы `\!`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек. Этот текст просто передаётся оболочке в буквальном виде.

`\? [ тема ]`

Показывает справочную информацию. Необязательный параметр тема (по умолчанию `commands`) выбирает описание интересующей части `psql`: `commands` описывает команды `psql` с обратной косой чертой; `options` описывает параметры командной строки, которые можно передать `psql`; а `variables` выдаёт справку по переменным конфигурации `psql`.

`\;`

Точка с запятой после косой черты не является метакомандой в том смысле, что предыдущие; при её вводе в буфер просто добавляется точка с запятой без обработки.

Обычно `psql` передаёт SQL-команду серверу, как только встречает завершающую команду точку с запятой, даже если текущая строка на этом не

заканчивается. Так, например, при вводе

```
select 1; select 2; select 3;
```

на сервер по отдельности будут переданы три SQL-команды, и результат каждой команды будет выведен перед переходом к следующей. Однако если точка с запятой вводится как `\;`, команда не передаётся на обработку, так что команды до и после этих символов по сути объединяются и передаются серверу в одном запросе. Поэтому, например, при выполнении:

```
select 1\; select 2\; select 3;
```

серверу передаются сразу три SQL-команды при достижении неэкранированной точки с запятой. Сервер выполняет такой запрос как одну транзакцию, если только в строку не включены явные команды BEGIN/COMMIT, которые разделят её на несколько транзакций. `psql` выводит результат только последнего запроса из всей строки. Так, в последнем примере `psql` выводит только 3, хотя на самом деле выполнялись все три команды.

17.5.4. Шаблоны поиска (Patterns)

Различные `\d` команды принимают параметр `pattern` для указания имени(имен) объектов для отображения. В простейшем случае шаблон - это точное имя объекта. Символы внутри шаблона обычно приводятся к нижнему регистру, как и для имен SQL объектов; к примеру `\dt FOO` выводит таблицу с именем `foo`. Как и для SQL имен, двойные кавычки вокруг шаблона предотвращают перевод в нижний регистр. Для включения символа двойной кавычки в шаблон используются два символа двойных кавычек подряд внутри шаблона в двойных кавычках. Опять же, это соответствует правилам для SQL идентификаторов. Например `\dt «FOO»«BAR»` будет выводить таблицу с именем `FOO»BAR` (но не `foo»bar`). В отличие от обычных правил для SQL имен, можно взять в двойные кавычки только часть шаблона, например `\dt FOO»FOO»BAR` будет выводить таблицу с именем `fooFOObar`.

Если `pattern` вообще не указан, `\d` команды выводят все объекты, видимые для текущей схемы. Это эквивалентно указанию `*` в качестве шаблона. Объект

считается видимым, если схема, в которой он находится, лежит на пути поиска, и объект с таким типом и именем на пути поиска ещё не появлялся. Это эквивалентно утверждению, что на объект можно ссылаться по имени, без явного указания схемы. Чтобы увидеть все объекты в базе данных, независимо от видимости, в качестве шаблона используется `*.*`.

Внутри шаблона `*` обозначает любое количество символов, включая отсутствие символов. `?` соответствует любому одному символу. Это соответствует шаблонам имен файлов в Unix. Например, `\dt int*` отображает все таблицы, чьи имена начинаются на `int`. Однако внутри двойных кавычек `*` и `?` теряют свое специальное значение и становятся обычными символами.

Шаблон, содержащий точку (`.`), интерпретируется как шаблон имени схемы, за которым следует шаблон имени объекта. Например, `\dt foo*.*bar*` отображает все таблицы, имена которых включают `bar`, и расположенные в схемах, имена которых начинаются с `foo`. Шаблоны, не содержащие точку, могут соответствовать только объектам текущей схемы. Опять же, точка внутри двойных кавычек теряет свое специальное значение.

Опытные пользователи могут использовать возможности регулярных выражений, такие как классы символов. Например `[0-9]` соответствует любой цифре. Все специальные символы регулярных выражений работают как обычно, за исключением: `.` используется в качестве разделителя, как говорилось выше; `*` соответствует регулярному выражению `.*`; `?` соответствует `.`, а также символ `$`, который не имеет специального значения. При необходимости эти символы можно эмулировать указывая `?` для эмуляции `.`, `(R+|)` для `R*`, `(R|)` для `R?`. `$` не требуется, как символ регулярного выражения, потому что шаблон должен соответствовать имени целиком, в отличие от обычной интерпретации регулярных выражений (другими словами, `$` автоматически добавляется в шаблон). Используйте `*` в начале и/или в конце, если не хотите, чтобы шаблон закреплялся. Обратите внимание, что внутри двойных кавычек, все специальные символы регулярных выражений теряют свое специальное значение и соответствуют сами себе. Также,

специальные символы регулярных выражений не действуют в шаблонах для имен операторов (т.е. в аргументе команды `\do`).

17.6.Расширенные возможности

17.6.1. Переменные

`psql` предоставляет возможности подстановки переменных подобные тем, что используются в командных оболочках Unix. Переменные представляют собой пары имя/значение, где значением может быть любая строка любой длины. Имя должно состоять из букв (включая нелатинские буквы), цифр и знаков подчеркивания.

Чтобы установить переменную, используется `psql` мета-команда `\set`, например:

```
testdb=> \set foo bar
```

присваивает переменной `foo` значение `bar`. Чтобы получить значение переменной, нужно поставить двоеточие перед её именем, например:

```
testdb=> \echo :foo  
bar
```

Это работает как в обычных SQL командах, так и в мета-командах, подробности в разделе Интерполяция SQL ниже.

При вызове `\set` без второго аргумента переменной присваивается пустая строка. Для удаления переменной используется команда `\unset`. Чтобы посмотреть значения всех переменных, нужно вызвать `\set` без аргументов.

Примечание. На аргументы `\set` распространяются те же правила подстановки, что и для других команд. Таким образом можно создавать интересные ссылки, например `\set :foo 'something'`, получая «мягкие ссылки» в Perl или «переменные переменных» в PHP. К сожалению (или к счастью?), с этими конструкциями нельзя сделать ничего полезного. С другой стороны, `\set bar :foo` является прекрасным способом копирования переменной.

Некоторые переменные обрабатываются в `psql` особым образом. Они представляют собой определенные параметры, которые могут быть изменены во время выполнения путем присваивания нового значения, а в некоторых переменных содержится изменяемое состояние `psql`. Хотя эти переменные можно использовать и

для других целей, делать это не рекомендуется, так как поведение программы может быстро стать очень странным. По соглашению, имена специальных переменных состоят из всех заглавных ASCII букв (и, возможно, цифр и знаков подчеркивания). Для максимальной совместимости в будущем старайтесь не использовать такие имена для собственных переменных. Переменные, управляющие поведением `psql`, обычно нельзя сбросить или задать для них недопустимые значения. Команда `\unset` для них допускается, но воспринимается как установка значения по умолчанию. Команда `\set` без второго аргумента воспринимается как присвоение переменной значения `on`, для управляющих переменных, принимающих это значение, и не принимается для других. Также управляющие переменные, принимающие значения `on` и `off`, примут и другие общепринятые написания логических значений, например `true` и `false`.

AUTOCOMMIT

При значении `on` (по умолчанию) после каждой успешно выполненной команды выполняется фиксация изменений. Чтобы отложить фиксацию изменений в этом режиме, нужно выполнить SQL команду `BEGIN` или `START TRANSACTION`. При значении `off` или если переменная не определена, фиксация изменений не происходит до тех пор, пока явно не выполнена команда `COMMIT` или `END`. При значении `off` неявно выполняется `BEGIN` непосредственно перед любой командой, за исключением случаев когда: команда уже в транзакционном блоке; перед самой командой `BEGIN` или другой командой управления транзакциями; перед командой, которая не может выполняться внутри транзакционного блока (например `VACUUM`).

Примечания:

1. Если режим `autocommit` отключен, необходимо явно откатывать изменения в неуспешных транзакциях, выполняя команду `ABORT` или `ROLLBACK`. Также имейте в виду, что при выходе из сессии без фиксации изменений несохраненные изменения будут потеряны.

2. Включенный режим `autocommit` является традиционным для Синергия-БД, а выключенный режим ближе к спецификации SQL. Если вы предпочитаете отключить режим `autocommit`, это можно сделать в общесистемном файле `psqlrc` или в персональном файле `~/.psqlrc`.

COMP_KEYWORD_CASE

Определяет, какой регистр букв будет использован при автоматическом завершении ключевых слов SQL. Если установлено в `lower` или `upper`, будет использоваться нижний или верхний регистр соответственно. Если установлено в `preserve-lower` или `preserve-upper` (по умолчанию), то завершаемое слово будет в том же регистре, что и уже введенное начало слова, но последующие слова, завершаемые полностью, будут в нижнем или верхнем регистре соответственно.

DBNAME

Имя базы данных, к которой вы сейчас подключены. Устанавливается всякий раз при подключении к базе данных (в том числе при старте программы), но эту переменную можно удалить.

ECHO

Со значением `all` все непустые входящие строки выводятся на стандартный вывод по мере их чтения. Это не относится к строкам, считываемым интерактивно. Чтобы выбрать такое поведение при запуске программы, добавьте ключ `-a`. Со значением `queries psql` выводит каждый запрос, отправляемый серверу, на стандартный вывод. Этому значению соответствует ключ `-e`. Со значением `errors` в стандартный канал ошибок выдаются только запросы, вызвавшие ошибки. Ему соответствует ключ `-b`. Со значением `none` (по умолчанию), никакие запросы не выводятся.

ECHO_HIDDEN

Если эта переменная имеет значение `on` и метакоманда обращается к базе данных, сначала выводится текст нижележащего запроса. Это помогает изучать внутреннее устройство Синергия-БД и реализовывать похожую функциональность в своих программах. Чтобы включить такое поведение при запуске программы, воспользуйтесь ключом `-E`. Если вы зададите для этой переменной значение `noexec`, запросы будут просто показываться, но не будут отправляться на сервер и выполняться. Значение по умолчанию – `off`.

ENCODING

Текущая кодировка символов на стороне клиента. Устанавливается всякий раз

при подключении к базе данных (в том числе при старте программы) и при смене кодировки командой `\encoding`, но эту переменную можно изменить или сбросить.

`ERROR`

`true` в случае ошибки последнего SQL-запроса, `false`, если он был выполнен успешно.

`FETCH_COUNT`

Если значение этой переменной – целое число больше нуля, результаты запросов `SELECT` извлекаются из базы данных и отображаются группами с заданным количеством строк, в отличие от поведения по умолчанию, когда перед отображением результирующий набор накапливается целиком. Это позволяет использовать ограниченный размер памяти независимо от размера выборки. При включении этой функциональности обычно используются значения от 100 до 1000. Имейте в виду, что запрос может завершиться ошибкой после отображения некоторого количества строк.

Примечание. Хотя можно использовать любой формат вывода, формат по умолчанию `aligned` как правило выглядит хуже, потому что каждая группа по `FETCH_COUNT` строк форматируется отдельно, что может привести к разной ширине столбцов в разных группах. Остальные форматы вывода работают лучше.

`HISTCONTROL`

Если переменная установлена в `ignorespace`, строки, начинающиеся с пробела, не сохраняются в истории. Если установлена в `ignoredups`, строки, которые уже есть в истории, повторно не сохраняются. Значение `ignoreboth` сочетает в себе оба варианта. Если значение не задано или отличается от перечисленных выше, все строки, введенные в интерактивном режиме, сохраняются в истории.

`HISTFILE`

Имя файла для хранения истории команд. Значение по умолчанию `~/.psql_history`. Например, установив:

```
\set HISTFILE ~/.psql_history- :DBNAME
```

в `~/.psqlrc`, `psql` будет поддерживать отдельную историю для каждой базы

данных.

HISTSIZE

Количество команд для хранения в истории. Значение по умолчанию 500.

HOST

Содержит имя хоста, где расположен сервер базы данных, к которому вы сейчас подключены. Устанавливается всякий раз при подключении к базе данных (в том числе при старте программы), но эту переменную можно удалить.

IGNOREEOF

Если не установлена, то ввод символа EOF (обычно Control+D) в интерактивной сессии psql завершает работу приложения. Если задано числовое значение, то именно такое количество символов EOF будет проигнорировано, прежде чем приложение завершит работу. Если переменная задана, но имеет не числовое значение, то значение по умолчанию 10.

LASTOID

Содержит значение последнего OID, полученного командой INSERT или \lo_import. Корректное значение переменной гарантируется до тех пор, пока не будет отображен результат следующей SQL команды.

LAST_ERROR_MESSAGE
LAST_ERROR_SQLSTATE

Основное сообщение об ошибке и связанный код SQLSTATE для последнего неудавшегося запроса в текущем сеансе psql либо пустая строка и 00000, если в текущем сеансе не происходили ошибки.

ON_ERROR_ROLLBACK

При значении on, если команда в транзакционном блоке выдает ошибку, ошибка игнорируется и транзакция продолжается. При значении interactive такие ошибки игнорируются только в интерактивных сессиях, но не в скриптах. Если переменная не установлена или при значении off команда, вызвавшая ошибку в транзакционном блоке, отменяет всю транзакцию. Режим отката транзакции при возникновении ошибки обеспечивается выполнением неявных команд SAVEPOINT перед каждой командой в транзакционном блоке и откатом к точке сохранения в

случае ошибки.

ON_ERROR_STOP

По умолчанию, после возникновения ошибки обработка команд продолжается. Если эта переменная установлена в значение on, обработка команд будет немедленно прекращена. В интерактивном режиме psql вернется в командную строку; иначе psql прекратит работу с кодом возврата 3, чтобы отличить этот случай от фатальных ошибок, для которых используется код возврата 1. В любом случае, выполнение всех запущенных скриптов (верхнеуровневый скрипт и любые другие, которые он мог запустить) будет немедленно прекращено. Если верхнеуровневая командная строка содержит несколько SQL команд, выполнение завершится на текущей команде.

PORT

Содержит порт сервера базы данных, к которому вы сейчас подключены. Устанавливается всякий раз при подключении к базе данных (в том числе при старте программы), но эту переменную можно удалить.

PROMPT1

PROMPT2

PROMPT3

Указывают, как должны выглядеть приглашения psql.

QUIET

Установка значения on эквивалента опции командной строки -q. Это, вероятно, не слишком полезно в интерактивном режиме.

ROW_COUNT

Число строк, возвращённых или обработанных последним SQL-запросом, либо 0, если запрос завершился неудачно или не возвратил количество строк.

SERVER_VERSION_NAME

SERVER_VERSION_NUM

Номер версии сервера в виде строки, например 9.6.2, 10.1 или 11beta1, и в числовом виде, например, 90602 или 100001. Они устанавливаются при каждом подключении к базе данных (в том числе при запуске программы), но их можно изменить или сбросить.

SHOW_CONTEXT

Этой переменной можно присвоить значения `never` (никогда), `errors` (ошибки) или `always` (всегда), определяющие, когда в сообщениях с сервера будут выводиться поля КОНТЕКСТ. По умолчанию выбран вариант `errors` (что означает, что контекст будет выводиться в сообщениях об ошибках, но не в предупреждениях и уведомлениях). Этот параметр не действует, когда установлен уровень `VERBOSITY terse`. Когда вам потребуется подробная версия только что выданной ошибки, может быть полезна команда `\errverbose`.

SINGLELINE

Установка значения `on` эквивалента опции командной строки `-S`.

SINGLESTEP

Эта переменная эквивалента опции командной строки `-s`.

SQLSTATE

Код ошибки (см. Приложение 1), связанной с неудачным выполнением последнего SQL-запроса, либо `00000` в случае его успешного завершения.

USER

Содержит имя пользователя базы данных, который сейчас подключен. Устанавливается всякий раз при подключении к базе данных (в том числе при старте программы), но эту переменную можно удалить.

VERBOSITY

Этой переменной можно присвоить значения `default`, `verbose` или `terse` для контроля уровня детализации отчетов об ошибках.

VERSION

VERSION_NAME

VERSION_NUM

Эти переменные устанавливаются при запуске программы и отражают версию `psql` соответственно в виде развёрнутой строки, краткой строки (например, `9.6.2`, `10.1` или `11beta1`) и числа (например, `90602` или `100001`). Их можно изменить или сбросить.

17.6.2. Интерполяция SQL

Ключевой особенностью переменных `psql` является возможность подставлять («интерполировать») их в команды SQL, также как и в аргументы мета-команд. Кроме того, `psql` предоставляет средства для корректного использования кавычек для значений переменных, которые используются как SQL литералы или идентификаторы. Синтаксис для подстановки значения без кавычек - это добавить перед именем переменной двоеточие (:). Например:

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

будет запрашивать таблицу `my_table`. Обратите внимание, что это может быть небезопасным: значение переменной копируется буквально, поэтому оно может содержать непарные кавычки или даже мета-команды. При применении необходимо убедиться, что это имеет смысл.

Когда значение будет использоваться в качестве SQL литерала или идентификатора, безопаснее заключить его в кавычки. Если значение переменной используется как SQL литерал, то после двоеточия нужно написать имя переменной в одинарных кавычках. Если значение переменной используется как SQL идентификатор, то после двоеточия нужно написать имя переменной в двойных кавычках. Эти конструкции корректно работают с кавычками и другими специальными символами, которые могут содержаться в значении переменной. Предыдущий пример более безопасно выглядит так:

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :»foo«;
```

Подстановка переменных не будет выполняться, если SQL литералы или идентификаторы заключены в кавычки. Поэтому конструкция `:foo` не превратится во взятое в кавычки значение переменной (и это было бы не безопасно, если бы работало, т.к. обработка кавычек внутри значения переменной была бы некорректной).

Один из примеров использования данного механизма - это копирование содержимого файла в столбец таблицы. Сначала загрузим содержимое файла в

переменную, затем подставим значение переменной как строку в кавычках:

```
testdb=> \set content `cat my_file.txt`  
testdb=> INSERT INTO my_table VALUES (: 'content');
```

Отметим, что это пока не будет работать, если `my_file.txt` содержит байт NUL. `psql` не поддерживает NUL в значениях переменных.

Так как двоеточие может легально присутствовать в SQL командах, попытка подстановки (например для `:name`, `: 'name'` или `:»name»`) не выполняется, если переменная не установлена. В любом случае, можно экранировать двоеточие с помощью обратной косой черты, чтобы предотвратить подстановку.

Специальная конструкция `:{?имя}` возвращает TRUE или FALSE в зависимости от того, существует ли переменная, и таким образом всегда подменяется значением, если только двоеточие не экранировано обратной косой чертой.

Использование двоеточия для переменных является стандартом SQL для встраиваемых языков запросов, таких как ECPG. Использование двоеточия для срезов массивов и приведения типов является расширениями Синергия-БД, что иногда может конфликтовать со стандартным использованием. Использование двоеточия и кавычек для экранирования значения переменной при подстановке в качестве SQL литерала или идентификатора - это расширение `psql`.

17.6.3. Настройка приглашений

Приглашения, выдаваемые `psql`, можно настроить по своему вкусу. Три переменные `PROMPT1`, `PROMPT2` и `PROMPT3` содержат строки и спецпоследовательности, задающие внешний вид приглашения. Приглашение 1 (`PROMPT1`) – это обычное приглашение, которое выдаётся, когда `psql` ожидает ввода новой команды. Приглашение 2 (`PROMPT2`) выдаётся, когда при вводе команды ожидается дополнительные строки, например потому что команда не была завершена точкой с запятой или не закрыты кавычки. Приглашение 3 (`PROMPT3`) выдаётся при выполнении SQL-команды `COPY FROM STDIN`, когда в терминале нужно ввести значение новой строки.

Значения этих переменных выводятся буквально, за исключением случаев, когда в них встречается знак процента (%). В зависимости от следующего символа будет подставляться определенный текст. Существуют следующие подстановки:

- %M

Полное имя хоста (с доменным именем) сервера базы данных, или [local] если подключение выполнено через сокет Unix домена, или [local:/dir/name], если сокет Unix домена не скомпилирован в местоположение по умолчанию.

- %m

Имя хоста сервера базы данных, усеченное до первой точки или [local] если подключение выполнено через сокет Unix домена.

- %>

Номер порта, который прослушивает сервер базы данных.

- %n

Имя пользователя базы данных для текущей сессии. Это значение может меняться в течении сессии в результате выполнения команды SET SESSION AUTHORIZATION.

- %/

Имя текущей базы данных.

- %~

Похоже на %/, но выводит тильду ~, если текущая база данных совпадает с базой данных по умолчанию.

- %#

Если пользователь текущей сессии является суперпользователем базы данных, то выводит #, иначе >. Это значение может меняться в течении сессии в результате выполнения команды SET SESSION AUTHORIZATION.

- %p

PID обслуживающего процесса для текущего подключения.

- %R

В приглашении 1 это обычно символ =, но @, если сеанс находится в неактивной ветви блока условия, либо ^ в одностроочном режиме либо !, если сеанс

не подключён к базе данных (что возможно при ошибке `\connect`). В приглашении 2 %R заменяется символом, показывающим, почему `psql` ожидает дополнительный ввод: `-`, если команда просто ещё не была завершена, но `*`, если не завершён комментарий `/* ... */`; апостроф, если не завершена строка в апострофах; кавычки, если не завершён идентификатор в кавычках; знак доллара, если не завершена строка в долларах; либо `(`, если после открывающей скобки не хватает закрывающей. В приглашении 3 %R не выдаёт ничего.

- %x

Состояние транзакции: пустая строка, если не в транзакционном блоке; `*`, когда в транзакционном блоке; `!`, когда в транзакционном блоке, в котором произошла ошибка и `?`, когда состояние транзакции не определено (например, нет подключения к базе данных).

- %l

Номер строки в текущем операторе, начиная с 1.

- %цифры

Подставляется символ с указанным восьмеричным кодом.

- %:имя:

Подставляется значение `psql` переменной `name`.

- %` команда `

Подставляется вывод команды, как и в обычной подстановке с обратными апострофами.

- %[... %]

Приглашения могут содержать управляющие символы терминала, которые, например, изменяют цвет, фон и стиль текста приглашения или изменяют заголовок окна терминала. Для того, чтобы возможности редактирования Readline работали правильно, непечатаемые символы нужно расположить между `%[` и `]`, чтобы сделать невидимыми. Можно делать несколько таких включений в приглашение, например:

```
testdb=> \set PROMPT1
'%[%033[1;33;40m%] %n@%/%R%[%033[0m%] %# '
```

выдаст жирное (1;), желтое на черном (33;40) приглашение для VT100 совместимых цветных терминалов.

Чтобы вставить знак процента, нужно написать `%%`. По умолчанию используются значения `'%/%R%# '` для `PROMPT1` и `PROMPT2` и `'>> '` для `PROMPT3`.

17.6.4. Редактирование командной строки

`psql` поддерживает библиотеку `Readline` для удобного редактирования командной строки. История команд автоматически сохраняется при выходе из `psql` и загружается при запуске. Завершение клавишей `TAB` также поддерживается, хотя логика завершения не претендует на роль парсера SQL. Запросы, генерируемые завершением по `TAB`, также могут конфликтовать с другими командами SQL, например `SET TRANSACTION ISOLATION LEVEL`. Если по какой-либо причине вам не нравится завершение по клавише `TAB`, его можно отключить в файле `.inputrc` в вашей домашней директории:

```
$if psql
set disable-completion on
$endif
```

17.7. Переменные окружения

`COLUMNS`

Если `\pset columns` равно нулю, управляет шириной формата вывода `wrapped`, а также определяет, нужно ли использовать пейджер и нужно ли переключаться в вертикальный формат в режиме `expanded auto`.

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию.

`PSQL_EDITOR`
`EDITOR`
`VISUAL`

Редактор, используемый командами `\e` и `\ef`. Переменные рассматриваются в перечисленном порядке. Будет использовано значение первой установленной переменной.

PSQL_EDITOR_LINENUMBER_ARG

Если в командах `\e` или `\ef` указан номер строки, эта переменная задает название опции командной строки, которая используется для передачи номера строки в редактор. Для таких редакторов как `vi` или Emacs это знак плюс. Добавьте в конец значения пробел, если он требуется для разделения названия опции и номера строки, например:

PSQL_EDITOR_LINENUMBER_ARG='+'

PSQL_EDITOR_LINENUMBER_ARG='--line '

Значение по умолчанию `+` на Unix системах (соответствует редактору по умолчанию `vi` и многим другим распространенным редакторам).

PSQL_HISTORY

Альтернативное расположение файла с историей команд. Допускается использование тильды (`~`).

PSQL_PAGER

PAGER

Если результат запроса не помещается на экране, он пропускается через эту программу. Обычно это `more` или `less`. От использования постраничника можно отказаться, присвоив переменной `PSQL_PAGER` или `PAGER` пустую строку, либо изменив соответствующие параметры с помощью команды `\pset`. Данные переменные просматриваются в этом же порядке; используется первая установленная. Если не установлена ни одна из переменных, в большинстве платформ используется `more`, а в Cygwin – `less`.

PSQLRC

Альтернативное расположение пользовательского файла `.psqlrc`. Допускается использование тильды (`~`).

SHELL

Команда операционной системы, выполняемая мета-командой `\!`.

TMPPDIR

Каталог для хранения временных файлов. По умолчанию /tmp.

Эта утилита, как и большинство других утилит Синергия-БД, также использует переменные среды, поддерживаемые libpq.

17.8.Файлы

psqlrc и ~/.psqlrc

При запуске без опций -X или -c psql пытается считать и выполнить команды из общесистемного стартового файла (psqlrc), а затем из персонального стартового файла пользователя (~/.psqlrc). Это происходит после подключения к базе данных, но до получения обычных команд. Эти файлы могут использоваться для настройки клиента и/или сервера, как правило, с помощью команд \set и SET.

Общесистемный стартовый файл называется psqlrc, он будет искаться в каталоге установке «конфигурация системы». Для того чтобы узнать этот каталог, надежнее всего выполнить команду pg_config --sysconfdir. По умолчанию он расположен в ../etc/ относительно каталога, содержащего исполняемые файлы Синергия-БД. Имя этого каталога можно задать явно через переменную окружения PGSYSCONFDIR.

Персональный стартовый файл пользователя называется .psqlrc, он будет искаться в домашнем каталоге вызывающего пользователя.

.psql_history

История командной строки хранится в файле ~/.psql_history.

Расположение файла истории можно задать явно через переменную окружения PSQL_HISTORY.

17.9.Замечания

psql лучше всего работает с серверами той же или более старой мажорной версии. Сбой мета-команды наиболее вероятен, если версия сервера новее, чем версия psql. Однако, мета-команды семейства \d должны работать с версиями сервера до 7.4, хотя и необязательно с серверами новее, чем сам psql. Общая

функциональность запуска SQL команд и отображения результатов запросов также должна работать на серверах с более новой мажорной версией, но это не гарантируется во всех случаях.

Если вы хотите подключаться к нескольким серверам с различными мажорными версиями, рекомендуется использовать последнюю версию `psql`. Кроме того можно сохранить копию `psql` от каждой мажорной версии и использовать ту, которая соответствует версии сервера. Но на практике в этих дополнительных сложностях нет необходимости.

17.10. Примеры

Первый пример показывает, что для ввода одной команды может потребоваться несколько строк. Обратите внимание, как меняется приглашение:

```
testdb=> CREATE TABLE my_table (
testdb(> first integer not null default 0,
testdb(> second text)
testdb-> ;
CREATE TABLE
```

Теперь посмотрим на определение таблицы:

```
testdb=> \d my_table
          Таблица "public.my_table"
Столбец |  Тип   | Правило сортировки | Допустимость NULL | По умолчанию
first   | integer |                     | not null          |              0
second  | text    |                     |                   |
```

Теперь изменим приглашение на что-то более интересное:

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

Предположим, что вы внесли данные в таблицу и хотите на них посмотреть:

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-----+-----
 1    | Один
 2    | Два
 3    | Три
```

4 | Четыре

(4 строки)

Таблицу можно вывести разными способами при помощи команды \pset:

```
peter@localhost testdb=> \pset border 2
```

Установлен стиль границ: 2.

```
peter@localhost testdb=> SELECT * FROM my_table;
```

```
+-----+-----+
| first | second |
+-----+-----+
|      1 | Один   |
|      2 | Два    |
|      3 | Три    |
|      4 | Четыре |
+-----+-----+
```

(4 строки)

```
peter@localhost testdb=> \pset border 0
```

Установлен стиль границ: 0.

```
peter@localhost testdb=> SELECT * FROM my_table;
```

```
first second
-----
```

```
1 Один
2 Два
3 Три
4 Четыре
```

(4 строки)

```
peter@localhost testdb=> \pset border 1
```

Установлен стиль границ: 1.

```
peter@localhost testdb=> \pset format unaligned
```

Формат вывода: unaligned.

```
peter@localhost testdb=> \pset fieldsep ','
```

Разделитель полей: «,».

```
peter@localhost testdb=> \pset tuples_only
```

Выводятся только кортежи.

```
peter@localhost testdb=> SELECT second, first FROM
my_table;
```

Один, 1

Два, 2

Три, 3

Четыре, 4

Также можно использовать короткие команды:

```
peter@localhost testdb=> \a \t \x
```

Формат вывода: aligned.

Режим вывода только кортежей выключен.

Расширенный вывод включен.

```
peter@localhost testdb=> SELECT * FROM my_table;
```

```
-[ RECORD 1 ]-
```

```
first      | 1
```

```
second     | Один
```

```
-[ RECORD 2 ]-
```

```
first      | 2
```

```
second     | Два
```

```
-[ RECORD 3 ]-
```

```
first      | 3
```

```
second     | Три
```

```
-[ RECORD 4 ]-
```

```
first      | 4
```

```
second     | Четыре
```

Когда это уместно, результаты запроса можно просмотреть в виде перекрёстной таблицы с помощью команды \crosstabview:

```
testdb=> SELECT first, second, first > 2 AS gt2 FROM
my_table;
```

```
first | second | gt2
-----+-----+-----
1     | one    | f
```


2		two		f
3		three		t
4		four		t

(4 rows)

```
testdb=> \crosstabview first second
```

first		one		two		three		four
-----+-----+-----+-----+-----								
1		f						
2				f				
3						t		
4								t

(4 rows)

Второй пример показывает таблицу умножения, строки в которой отсортированы в обратном числовом порядке, а столбцы – независимо, по возрастанию числовых значений.

```
testdb=> SELECT t1.first as "A", t2.first+100 AS "B",
t1.first*(t2.first+100) as "AxB",
testdb(> row_number() over(order by t2.first) AS ord
testdb(> FROM my_table t1 CROSS JOIN my_table t2 ORDER
BY 1 DESC
```

```
testdb(> \crosstabview "A" "B" "AxB" ord
```

A		101		102		103		104
----+-----+-----+-----+-----								
4		404		408		412		416
3		303		306		309		312
2		202		204		206		208
1		101		102		103		104

(4 rows)

18. REINDEXDB

reindexdb – переиндексировать базу данных Синергия-БД.

18.1. Синтаксис

```
reindexdb [параметр-подключения...] [параметр...] [ --  
schema | -S схема ] ... [ --table | -t таблица ] ... [ -  
-index | -i индекс ] ... [имя_бд]  
  
reindexdb [параметр-подключения...] [параметр...] --all  
| -a  
  
reindexdb [параметр-подключения...] [параметр...] --  
system | -s [имя_бд]
```

18.2. Описание

Утилита `reindexdb` предназначена для перестроения индексов в базе данных Синергия-БД.

Утилита `reindexdb` представляет собой обертку SQL-команды `REINDEX`. Переиндексация базы данных с её помощью по сути не отличается от переиндексации при обращении к серверу другими способами.

18.3. Параметры

`reindexdb` принимает аргументы командной строки, описанные ниже.

`-a`
`--all`

Переиндексировать все базы данных.

`[-d] dbname`
`[--dbname=] dbname`

Задаёт имя базы данных, подлежащей переиндексации. Если это имя не задано и отсутствует параметр `-a` (или `--all`), имя базы данных берётся из переменной окружения `PGDATABASE`. Если и эта переменная не задана, выбирается имя подключающегося пользователя.

`-e`
`--echo`

Выводить команды, которые `reindexdb` генерирует и передает серверу.

`-i индекс`
`--index=индекс`

Пересоздать только указанный индекс. Добавив дополнительные ключи `-i`, можно пересоздать несколько индексов.

-q
--quiet

Подавлять вывод сообщений о прогрессе выполнения.

-s
--system

Переиндексировать системные каталоги базы данных.

-S *схема*
--schema=*схема*

Переиндексировать только указанную схему. Переиндексировать несколько схем можно, добавив несколько ключей -S.

-t *таблица*
--table=*таблица*

Переиндексировать только указанную таблицу. Добавив дополнительные ключи -t, можно переиндексировать несколько таблиц.

-v
--verbose

Вывести подробную информацию во время процесса.

-V
--version

Сообщить версию reindexdb и завершиться.

-?
--help

Показать справку по аргументам командной строки reindexdb и завершиться.

Утилита reindexdb также принимает следующие аргументы командной строки в качестве параметров подключения:

-h *host*
--host=*host*

Указывает имя компьютера, на котором запущен сервер. Если значение начинается с косой черты, оно интерпретируется как имя каталога с доменным сокетом Unix.

-p *port*
--port=*port*

Указывает TCP-порт или расширение файла Unix-сокета, на котором сервер

слушает подключения.

```
-U username  
--username=username
```

Имя пользователя, под которым производится подключение.

```
-w  
--no-password
```

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Опция может быть полезна в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

```
-W  
--password
```

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как `reindexdb` запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако, чтобы понять это, `reindexdb` лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

```
--maintenance-db=dbname
```

Задаёт имя базы данных, через подключение к которой будут находиться другие базы, подлежащие переиндексации. По умолчанию используется `postgres`, а в случае её отсутствия – `template1`.

18.4.Переменные окружения

```
PGDATABASE  
PGHOST  
PGPORT  
PGUSER
```

Параметры подключения по умолчанию. Эта утилита, как и большинство других утилит Синергия-БД, также использует переменные среды, поддерживаемые `libpq`.

18.5.Замечания

Утилите `reindexdb` может потребоваться подключаться к серверу Синергия-БД

несколько раз, и при этом она будет каждый раз запрашивать пароль. В таких случаях удобно иметь файл `~/.pgpass`.

18.6.Примеры

Переиндексирование базы данных `test`:

```
$ reindexdb test
```

Переиндексирование таблицы `foo` и индекса `bar` в базе данных `abcd`:

```
$ reindexdb --table foo --index bar abcd
```

19. VACUUMDB

`vacuumdb` – выполнить очистку и анализ базы данных Синергия-БД.

19.1.Синтаксис

```
vacuumdb [параметр-подключения...] [параметр...] [ --  
table | -t таблица [( столбец [,...] )] ] ... [имя_бд]  
vacuumdb [параметр-подключения...] [параметр...] --all | -a
```

19.2.Описание

Утилита `vacuumdb` предназначена для очистки базы данных Синергия-БД. Кроме того, `vacuumdb` генерирует внутреннюю статистику, которую использует оптимизатор запросов Синергия-БД.

Утилита `vacuumdb` представляют собой обертку SQL-команды `VACUUM`. Выполнение очистки и анализа баз данных с её помощью по сути не отличается от выполнения тех же действий при обращении к серверу другими способами.

19.3.Параметры

Утилита `vacuumdb` принимает аргументы командной строки, описанные ниже.

```
-a  
--all
```

Очистить все базы данных.

```
[-d] dbname  
[--dbname=] dbname
```

Задает имя базы данных, подлежащей очистке. Если это имя не задано и

отсутствует параметр `-a` (или `--all`), имя базы данных берется из переменной окружения `PGDATABASE`. Если и эта переменная не задана, выбирается имя подключающегося пользователя.

`-e`
`--echo`

Выводить команды, которые `vacuumdb` генерирует и передает серверу.

`-f`
`--full`

Произвести «полную» очистку.

`-F`
`--freeze`

Агрессивно «замораживать» версии строк.

`-j njobs`
`--jobs=njobs`

Выполнять команды `VACUUM` или `ANALYZE` в параллельном режиме, запуская одновременно `njobs` команд. Это сокращает время обработки, но увеличивает нагрузку на сервер.

`vacuumdb` откроет `njobs` подключений к базе данных, так что убедитесь в том, что значение `max_connections` достаточно велико, чтобы все эти подключения были приняты.

Заметьте, что использование этого режима с параметром `-f` (`FULL`) может привести к отказам из-за взаимоблокировок, если параллельно начнут обрабатываться определённые системные каталоги.

`-q`
`--quiet`

Подавлять вывод сообщений о прогрессе выполнения.

`-t таблица [ (колонка [, ...]) ]`
`--table=таблица [ (колонка [, ...]) ]`

Производить очистку или анализ только указанной таблицы. Имена колонок можно указать только в сочетании с параметрами `--analyze` и `--analyze-only`. Добавив дополнительные ключи `-t`, можно обработать несколько таблиц.

Если вы указываете колонки, вам, вероятно, придется экранировать скобки в

оболочке.

`-v`
`--verbose`

Вывести подробную информацию во время процесса.

`-V`
`--version`

Сообщить версию `vacuumdb` и завершиться.

`-z`
`--analyze`

Также вычислить статистику для оптимизатора.

`-Z`
`--analyze-only`

Только вычислить статистику для оптимизатора (не производить очистку).

`--analyze-in-stages`

Только вычислить статистику для оптимизатора (без очистки), подобно `--analyze-only`. Но для скорейшего получения полезной статистики, выполнить анализ в несколько проходов (в настоящее время, три) с разными настройками.

Этот параметр полезен при необходимости провести анализ базы данных, только что наполненной данными из архива или командой `pg_upgrade`. С этим параметром `vacuumdb` постарается получить некоторую статистику как можно скорее, чтобы базой можно было пользоваться, а на следующих проходах вычислит полную статистику.

`-?`
`--help`

Показать справку по аргументам командной строки `vacuumdb` и завершиться.

Утилита `reindexdb` также принимает следующие аргументы командной строки в качестве параметров подключения:

`-h host`
`--host=host`

Указывает имя компьютера, на котором запущен сервер. Если значение начинается с косой черты, оно интерпретируется как имя каталога с доменным сокетом Unix.

```
-p port  
--port=port
```

Указывает TCP-порт или расширение файла Unix-сокета, на котором сервер слушает подключения.

```
-U username  
--username=username
```

Имя пользователя, под которым производится подключение.

```
-w  
--no-password
```

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Опция может быть полезна в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

```
-W  
--password
```

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как `vacuumdb` запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако, чтобы понять это, `vacuumdb` лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

```
--maintenance-db=dbname
```

Задаёт имя базы данных, через подключение к которой будут находиться другие базы, подлежащие очистке. По умолчанию используется `postgres`, а в случае её отсутствия – `template1`.

19.4. Переменные окружения

```
PGDATABASE  
PGHOST  
PGPORT  
PGUSER
```

Параметры подключения по умолчанию. Эта утилита, как и большинство других утилит Синергия-БД, также использует переменные среды, поддерживаемые

libpq.

19.5.Диагностика

В случае возникновения трудностей, обратитесь к описаниям `VACUUM` и `psql`, где обсуждаются потенциальные проблемы и сообщения об ошибках. Учтите, что на целевом компьютере должен работать сервер баз данных. При этом применяются все свойства подключения по умолчанию и переменные окружения, которые использует клиентская библиотека `libpq`.

19.6.Замечания

Утилите `vacuumdb` может потребоваться подключаться к серверу Синергия-БД несколько раз, и при этом она будет каждый раз запрашивать пароль. В таких случаях удобно иметь файл `~/.pgpass`.

19.7.Примеры

Очистка базы данных `test`:

```
$ vacuumdb test
```

Очистка и анализ для оптимизатора базы данных `bigdb`:

```
$ vacuumdb --analyze bigdb
```

Очистка одной таблицы `foo` в базе данных `xyzy` и анализ только колонки `bar` этой таблицы для оптимизатора:

```
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzy
```

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

SQL	-	Structured Query Language (язык структурированных запросов)
WAL	-	Write-Ahead Log (журнал упреждающей записи)
БД	-	база данных
КСЗ	-	комплекс средств защиты
ОЗУ	-	оперативное запоминающее устройство
ОС	-	операционная система
ПРД	-	правила разграничения доступа
СУБД	-	система управления базами данных

ПРИЛОЖЕНИЕ 1.

(справочное)

Коды ошибок СУБД «Синергия-БД»

Всем сообщениям, которые выдаёт СУБД «Синергия-БД», назначены пятисимвольные коды ошибок, соответствующие кодам «SQLSTATE», описанным в стандарте SQL. Приложения, которые должны знать, какое условие ошибки имело место, обычно проверяют код ошибки и только потом обращаются к текстовому сообщению об ошибке. Заметьте, что отдельные, но не все коды ошибок, которые выдаёт Синергия-БД, определены стандартом SQL; некоторые дополнительные коды ошибок для условий, не описанных стандартом, были добавлены независимо или позаимствованы из других баз данных.

Согласно стандарту, первые два символа кода ошибки обозначают класс ошибок, а последние три символа обозначают определённое условие в этом классе. Таким образом, приложение, не знающее значение определённого кода ошибки, всё же может понять, что делать, по классу ошибки.

В таблице 1.1 перечислены все коды ошибок, определённые в Синергия-БД. Некоторые коды в настоящее время не используются, хотя они определены в стандарте SQL. Также показаны классы ошибок. Для каждого класса ошибок имеется «стандартный» код ошибки с последними тремя символами 000. Этот код выдаётся только для таких условий ошибок, которые относятся к некоторому классу, но не имеют более определённого кода.

Символ, указанный в столбце «Имя условия», определяет условие в PL/pgSQL. Имена условий могут записываться в верхнем или нижнем регистре. Заметьте, что PL/pgSQL, в отличие от ошибок, не распознаёт предупреждения; то есть классы 00, 01 и 02.

Для некоторых типов ошибок сервер сообщает имя объекта базы данных (таблица, столбец таблицы, тип данных или ограничение), связанного с ошибкой; например, имя уникального ограничения, вызвавшего ошибку `unique_violation`.

Такие имена передаются в отдельных полях сообщения об ошибке, чтобы приложениям не пришлось извлекать его из возможно локализованного текста ошибки для человека. На данный момент полностью охватываются только ошибки класса SQLSTATE 23 (нарушения ограничений целостности), но в будущем должны быть охвачены и другие классы.

Таблица 1.1 – Коды ошибок СУБД «Синергия-БД»

Код ошибки	Имя условия
Класс 00 – Успешное завершение	
00000	successful completion
Класс 01 – Предупреждение	
01000	warning
0100C	dynamic result sets returned
01008	implicit zero bit padding
01003	null value eliminated in set function
01007	privilege not granted
01006	privilege not revoked
01004	string data right truncation
01P01	deprecated feature
Класс 02 – Нет данных (это также класс предупреждений согласно стандарту SQL)	
02000	no data
02001	no additional dynamic result sets returned
Класс 03 – SQL-оператор ещё не завершён	
03000	sql statement not yet complete
Класс 08 – Исключение, связанное с подключением	
08000	connection exception
08003	connection does not exist
08006	connection failure
08001	sqlclient unable to establish sqlconnection
08004	sqlserver rejected establishment of sqlconnection
08007	transaction resolution unknown
08P01	protocol violation
Класс 09 – Исключение с действием триггера	
09000	triggered action exception
Класс 0A – Неподдерживаемая функциональность	
0A000	feature not supported
Класс 0B – Неверное начало транзакции	
0B000	invalid transaction initiation
Класс 0F – Исключение с указателем на данные	
0F000	locator exception
0F001	invalid locator specification
Класс 0L – Неверный праводатель	
0L000	invalid grantor
0LP01	invalid grant operation
Класс 0P – Неверное указание роли	
0P000	invalid role specification
Класс 0Z – Исключение диагностики	
0Z000	diagnostics exception
0Z002	stacked diagnostics accessed without active handler
Класс 20 – Case не найден	
20000	case not found

Продолжение таблицы 1.1

Код ошибки	Имя условия
Класс 21 – Нарушение количества	
21000	cardinality_violation
Класс 22 – Исключение в данных	
22000	data_exception
2202E	array subscript error
22021	character not in repertoire
22008	datetime field overflow
22012	division by zero
22005	error in assignment
2200B	escape character conflict
22022	indicator overflow
22015	interval field overflow
2201E	invalid argument for logarithm
22014	invalid argument for ntile function
22016	invalid argument for nth value function
2201F	invalid argument for power function
2201G	invalid argument for width bucket function
22018	invalid character value for cast
22007	invalid datetime format
22019	invalid escape character
2200D	invalid escape octet
22025	invalid escape sequence
22P06	nonstandard use of escape character
22010	invalid indicator parameter value
22023	invalid parameter value
22013	invalid preceding or following size
2201B	invalid regular expression
2201W	invalid row count in limit clause
2201X	invalid row count in result offset clause
2202H	invalid tablesample argument
2202G	invalid tablesample repeat
22009	invalid time zone displacement value
2200C	invalid use of escape character
2200G	most specific type mismatch
22004	null value not allowed
22002	null value no indicator parameter
22003	numeric value out of range
2200H	sequence generator limit exceeded
22026	string data length mismatch
22001	string data right truncation
22011	substring error
22027	trim error
22024	unterminated c string
2200F	zero length character string
22P01	floating point exception
22P02	invalid text representation
22P03	invalid binary representation
22P04	bad copy file format
22P05	untranslatable character
2200L	not an xml document
2200M	invalid xml document
2200N	invalid xml content
2200S	invalid xml comment
2200T	invalid xml processing instruction
Класс 23 – Нарушение ограничения целостности	

Продолжение таблицы 1.1

Код ошибки	Имя условия
23000	integrity constraint violation
23001	restrict violation
23502	not null violation
23503	foreign key violation
23505	unique violation
23514	check violation
23P01	exclusion violation
Класс 24 – Неверное состояние курсора	
24000	invalid cursor state
Класс 25 – Неверное состояние транзакции	
25000	invalid transaction state
25001	active sql transaction
25002	branch transaction already active
25008	held cursor requires same isolation level
25003	inappropriate access mode for branch transaction
25004	inappropriate isolation level for branch transaction
25005	no active sql transaction for branch transaction
25006	read only sql transaction
25007	schema and data statement mixing not supported
25P01	no active sql transaction
25P02	in failed sql transaction
25P03	idle in transaction session timeout
Класс 26 – Неверное имя SQL-оператора	
26000	invalid sql statement name
Класс 27 – Нарушение при изменении данных в триггере	
27000	triggered data change violation
Класс 28 – Неверное указание авторизации	
28000	invalid authorization specification
28P01	invalid password
Класс 2В – Зависимые описания привилегий всё ещё существуют	
2В000	dependent privilege descriptors still exist
2ВP01	dependent objects still exist
Класс 2D – Неверное завершение транзакции	
2D000	invalid transaction termination
Класс 2F – Исключение в подпрограмме SQL	
2F000	sql routine exception
2F005	function executed no return statement
2F002	modifying sql data not permitted
2F003	prohibited sql statement attempted
2F004	reading sql data not permitted
Класс 34 – Неверное имя курсора	
34000	invalid cursor name
Класс 38 – Исключение во внешней подпрограмме	
38000	external routine exception
38001	containing sql not permitted
38002	modifying sql data not permitted
38003	prohibited sql statement attempted
38004	reading sql data not permitted
Класс 39 – Исключение при вызове внешней подпрограммы	
39000	external routine invocation exception
39001	invalid sqlstate returned
39004	null value not allowed
39P01	trigger protocol violated
39P02	srf protocol violated
39P03	event trigger protocol violated

Продолжение таблицы 1.1

Код ошибки	Имя условия
Класс 3В – Исключение точки сохранения	
3В000	savepoint_exception
3В001	invalid_savepoint_specification
Класс 3D – Неверное имя каталога	
3D000	invalid_catalog_name
Класс 3F – Неверное имя схемы	
3F000	invalid_schema_name
Класс 40 – Откат транзакции	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
Класс 42 – Ошибка синтаксиса или нарушение правила доступа	
42000	syntax_error_or_access_rule_violation
42601	syntax_error
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name
42622	name_too_long
42939	reserved_name
42804	datatype_mismatch
42P18	indeterminate_datatype
42P21	collation_mismatch
42P22	indeterminate_collation
42809	wrong_object_type
428C9	generated_always
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object
42702	ambiguous_column
42725	ambiguous_function
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition

Продолжение таблицы 1.1

Код ошибки	Имя условия
42P15	invalid schema definition
42P16	invalid table definition
42P17	invalid object definition
Класс 44 – Нарушение WITH CHECK OPTION	
44000	with check option violation
Класс 53 – Нехватка ресурсов	
53000	insufficient resources
53100	disk full
53200	out of memory
53300	too many connections
53400	configuration limit exceeded
Класс 54 – Превышение ограничения программы	
54000	program limit exceeded
54001	statement too complex
54011	too many columns
54023	too many arguments
Класс 55 – Объект не в требуемом состоянии	
55000	object not in prerequisite state
55006	object in use
55P02	cant change runtime param
55P03	lock not available
Класс 57 – Вмешательство оператора	
57000	operator intervention
57014	query canceled
57P01	admin shutdown
57P02	crash shutdown
57P03	cannot connect now
57P04	database dropped
Класс 58 – Ошибка системы (ошибка, внешняя по отношению к PostgreSQL)	
58000	system error
58030	io error
58P01	undefined file
58P02	duplicate file
Класс 72 – Ошибка снимка	
72000	snapshot too old
Класс F0 – Ошибка файла конфигурации	
F0000	config file error
F0001	lock file exists
Класс HV – Ошибка обёртки сторонних данных (SQL/MED)	
HV000	fdw error
HV005	fdw column name not found
HV002	fdw dynamic parameter value needed
HV010	fdw function sequence error
HV021	fdw inconsistent descriptor information
HV024	fdw invalid attribute value
HV007	fdw invalid column name
HV008	fdw invalid column number
HV004	fdw invalid data type
HV006	fdw invalid data type descriptors
HV091	fdw invalid descriptor field identifier
HV00B	fdw invalid handle
HV00C	fdw invalid option index
HV00D	fdw invalid option name
HV090	fdw invalid string length or buffer length
HV00A	fdw invalid string format

Окончание таблицы 1.1

Код ошибки	Имя условия
HV009	fdw_invalid_use_of_null_pointer
HV014	fdw_too_many_handles
HV001	fdw_out_of_memory
HV00P	fdw_no_schemas
HV00J	fdw_option_name_not_found
HV00K	fdw_reply_handle
HV00Q	fdw_schema_not_found
HV00R	fdw_table_not_found
HV00L	fdw_unable_to_create_execution
HV00M	fdw_unable_to_create_reply
HV00N	fdw_unable_to_establish_connection
Класс P0 – Ошибка PL/pgSQL	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure
Класс XX – Внутренняя ошибка	
XX000	internal_error
XX001	data_corrupted
XX002	index_corrupted

[illegible][illegible]