Акционерное общество
«Научно-производственное объединение
Русские Базовые Информационные Технологии»

ОПЕРАЦИОННАЯ СИСТЕМА СПЕЦИАЛЬНОГО НАЗНАЧЕНИЯ «ASTRA LINUX SPECIAL EDITION»

Описание защищенной СУБД

Листов 660

2018

АННОТАЦИЯ

Настоящий документ является руководством по использованию защищенной СУБД в операционной системы специального назначения «Astra Linux Special Edition» (далее по тексту — ОС).

В документе приведено описание защищенной СУБД, ее установка и настройка. Описан синтаксис языка запросов SQL, определена структура для хранения данных, заполнение базы данных и обращение к ней с запросами. Приведены доступные типы данных, функции и операторы, которые могут быть использованы в командах SQL. Также описаны вопросы резервного копирования и восстановления данных.

В качестве защищенной СУБД в составе ОС используется СУБД PostgreSQL, доработанная в соответствии с требованием интеграции с ОС в части мандатного управления доступом к информации и содержащая реализацию ДП-модели управления доступом и информационными потоками. Данная ДП-модель описывает все аспекты дискреционного, мандатного и ролевого управления доступом с учетом безопасности информационных потоков.

СУБД PostgreSQL предназначена для создания и управления реляционными БД и предоставляет многопользовательский доступ к расположенным в них данным.

Документ предназначен для администраторов и разработчиков баз данных.

СОДЕРЖАНИЕ

1. СИНТАКСИС SQL

Раздел содержит описание использования языка запросов SQL PostgreSQL. Приводится синтаксис языка запросов SQL, определена структура для хранения данных, способы наполнения базы данных и обращение к ней с запросами. Перечислены доступные типы данных, функции и операторы, которые могут быть использованы в командах SQL.

1.1. Лексическая структура

Входной поток SQL состоит из последовательности команд, команда — из последовательности токенов и завершается точкой с запятой (`;`). Конец входного потока также завершает команду. Какие токены являются правильными, а какие нет, зависит от синтаксиса отдельной команды.

Токен может быть *ключевым словом*, *идентификатором*, *заключенным в кавычки идентификатором*, *литералом* (или константой) или специальным символом. Токены обычно разделяются пробелами (табуляциями, символами новой строки), но это не является необходимым, если нет двусмысленности (которая обычно возникает только, если специальный символ примыкает к каким-либо другим типам токенов).

Например, следующие строки в потоке ввода SQL являются (синтаксически) правильными:

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

Данная последовательность состоит из трех команд, по одной в каждой строке (что необязательно; в одной строке может быть несколько команд, но при этом, и команда может быть разбита на несколько строк).

Кроме того, в потоке ввода SQL могут встречаться *комментарии*, которые не являются токенами и считаются эквивалентами пробелов.

Синтаксис SQL не является очень строгим относительно токенов, идентифицирующих команды, а также операндов и параметров. Первые несколько токенов обычно являются именем команды, как в примере выше, `SELECT`, `UPDATE` и `INSERT`. Но команда `UPDATE` всегда требует наличия токена `SET`, который должен находиться в определенной позиции, как и команда `INSERT` требует, чтобы токен `VALUES` находился в определенном месте.

1.1.1. Идентификаторы и ключевые слова

Токены, такие как `SELECT`, `UPDATE` или `VALUES` в приведенном примере, являются примерами *ключевых слов*, т. е. таких слов, которые имеют фиксированное значение в языке SQL. Токены `MY_TABLE` и `A` являются примерами *идентификаторов*. Они идентифицируют имена таблиц, столбцов или других объектов БД, в зависимости от команды, в которой они

используются. Таким образом, они иногда просто называются «именами». Ключевые слова и идентификаторы имеют одну и ту же лексическую структуру, потому не всегда понятно, является ли токен идентификатором или ключевым словом без знания языка.

Идентификаторы SQL и ключевые слова должны начинаться с букв (a–z, а также с букв с диакритическими отметками и не латинских букв) или символа подчеркивания (_). Остальные символы в идентификаторе или ключевом слове могут быть буквой, подчеркиванием, цифрой (0–9) или знаком доллара ($). Однако знак доллара в идентификаторах не разрешается по стандарту SQL, так что его использование может сделать приложения менее переносимыми. Стандарт также не определяет ключевых слов, содержащих цифры или начинающихся или заканчивающихся подчеркиванием, так что идентификаторы в таком виде являются наиболее безопасными с точки зрения их конфликтов с будущими расширениями этого стандарта.

Длина идентификатора должна быть не более NAMEDATALEN–1 символов; в командах могут использоваться и более длинные имена, но они будут урезаны. По умолчанию значение NAMEDATALEN составляет 256, так что максимальная длина идентификатора может быть 255 символа.

Идентификаторы и ключевые слова являются независимыми от регистра букв. Таким образом,

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

эквивалентно

```
uPDaTE my_TabLE SeT a = 5;
```

Часто используемое соглашение состоит в том, что ключевые слова пишут в верхнем регистре, а имена — в нижнем, т. е.

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

Существует второй вид идентификатора: *разделенный идентификатор* или *заключенный в кавычки идентификатор*. Он имеет форму заключенной в двойные кавычки (") произвольной последовательности символов. Разделенный идентификатор всегда является идентификатором и никогда не может быть ключевым словом. Так "select" может быть использован в качестве названия столбца или таблицы с именем "select", в то время как не заключенное в кавычки слово select является ключевым словом. В результате его использование там, где необходимо ввести имя таблицы или столбца, будет вызывать ошибку. Данный выше пример может быть переписан с использованием идентификаторов, заключенных в кавычки:

```
UPDATE "my_table" SET "a" = 5;
```

Заключенные в кавычки идентификаторы могут содержать любые символы, кроме символа с кодом ноль. (Чтобы включить символ двойной кавычки, необходимо использовать две двойные кавычки.) Это позволяет конструировать такие имена таблиц или столбцов,

которые в других случаях были бы невозможны, например содержащие пробелы или амперсанды. Однако ограничение на длину идентификатора остается в силе.

Один из вариантов идентификатора, заключенного в кавычки, позволяет включать экранированные символы Unicode, которые идентифицируются по их кодам. Такой вариант идентификатора начинается с `U&` (U в верхнем или нижнем регистре, за которым следует амперсанд) сразу же после открывающей двойной кавычки, без каких-либо пробелов между ними, например `U&"foo"`. (Следует обратить внимание, что это создает двусмысленность с оператором `&`, но использование пробелов вокруг данного оператора дает возможность избежать данной проблемы). Внутри кавычек символы Unicode могут быть заданы в экранированной форме с помощью обратной косой черты `\`, за которой следует 4-х разрядный шестнадцатеричный код, или в качестве альтернативы за обратной косой чертой следует знак плюс + и 6-ти разрядный шестнадцатеричный код. Например, идентификатор `"data"` может быть записан как:

```
U&"d\0061t\+000061"
```

Далее представлен менее простой пример, записывающий русское слово «слон» кириллицей:

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

Если в строке появляется другой символ экранирования, отличный от `\`, он может быть указан, используя слово `UESCAPE` после такой строки, например:

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

Символ экранирования может быть любым одиночным символом, отличным от шестнадцатеричной цифры, знаком плюс, одиночной кавычкой, двойной кавычкой или символом пробела. Символ экранирования записывается в одиночных кавычках, а не двойных.

При необходимости включения символа экранирования непосредственно в идентификатор, его необходимо указывать дважды.

Синтаксис экранирования Unicode работает только, когда кодировкой сервера является UTF8. Когда используются другие кодировки сервера, могут быть заданы только коды в диапазоне ASCII (до `\007F` включительно). Для задания суррогатных пар UTF-16 могут быть использованы как 4-х так и 6-ти разрядные формы, для символов с кодами больше, чем U+FFFF, хотя доступность 6-ти разрядной формы технически делает это ненужным. (Суррогатные пары не сохраняются напрямую, но комбинируются в единый код, который затем кодируется в UTF-8.)

Заключение идентификатора в кавычки также делает его зависимым от регистра букв. Например, идентификаторы `FOO`, `foo` и `"foo"` считаются PostgreSQL одинаковыми, а `"Foo"` и `"FOO"` отличаются от предыдущих трех и друг от друга. (Преобразование не заключенных в кавычки имен в нижний регистр в PostgreSQL является несовместимым со стандартном SQL,

согласно которому не заключенные в кавычки имена должны конвертироваться в верхний регистр.) Так, в соответствии со стандартном, имя `foo` должно быть эквивалентно `"FOO"`, а не `"foo"`. Если нужны переносимые приложения, следует либо всегда использовать кавычки, либо никогда не использовать их.

1.1.2. Константы

В PostgreSQL существует три вида *неявно-задаваемых констант*: строки, битовые строки и числа. Константы могут быть заданы с явным указанием типа, что позволит более аккуратно представить значение константы и эффективно ею управлять.

1.1.2.1. Строковые константы

Строковая константа в SQL — это произвольная последовательность символов, заключенных в одинарные кавычки (`'`), например `'This is a string'`. Чтобы включить символ одинарной кавычки внутрь строки, следует написать его дважды, например, `'Dianne''s horse'`. Подобное обозначение не является аналогом символа двойной кавычки (`"`).

Две строковые константы, которые разделены только пустым пространством (с как минимум одним переводом строки), склеиваются и рассматриваются, как если бы константа была задана одной строкой. Например:

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

эквивалентно:

```
SELECT 'foobar';
```

но:

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

является неправильным синтаксисом. (Такое поведение задается стандартом SQL.)

1.1.2.2. Строковые константы с экранированными последовательностями в стиле языка C

PostgreSQL понимает «экранированные» строковые константы, которые являются расширением стандарта SQL. Они задаются с помощью буквы `E` (в верхнем или нижнем регистре), за которой следует открывающая одиночная кавычка, например `E'foo'`. (Если экранированная строковая константа занимает несколько строк, следует писать `E` только перед первой открывающей кавычкой.) Внутри экранированной строки, символ обратная косая черта (`\`), начинает *экранированную обратной косой чертой* последовательность как в языке Си, в которой комбинация обратной косой черты и следующего за ней символа или символов, представляет специальное байтовое значение, как показано в 1.

Таблица 1 – Строковые константы в стиле языка C

Экранированная последовательность	Интерпретация
\b	Удаление предыдущего символа
\f	Подача формы
\n	Новая строка
\r	Возврат каретки
\t	Табуляция
\o, \oo, \ooo (o = 0 – 7)	Байт в восьмеричной системе
\xh, \xhh (h = 0 – 9, A – F)	Байт в шестнадцатеричной системе
\uxxxx, \Uxxxxxxxx (x = 0 – 9, A – F)	16-ти или 32-х разрядное шестнадцатеричное представление символа Unicode

Любой другой символ, следующий за \ представляет сам себя. Таким образом, чтобы включить символ \, следует написать его дважды (\\). Также в экранированную строку может быть включена одинарная кавычка с помощью написания \', в дополнение к обычному способу ''.

Необходимо следить, чтобы создаваемые байтовые последовательности, особенно когда используются восьмеричные или шестнадцатеричные последовательности, были правильными символами в кодировке, которая установлена на сервере. Когда кодировкой на сервере является UTF-8, то вместо этого должны использоваться последовательности Unicode или альтернативный ситаксис экранирования Unicode, описанный в 1.1.2.3. (Это альтернатива ручного ввода UTF-8 по байтам, который очень обременителен.)

Синтаксис экранирования Unicode полностью работает только когда кодировкой на сервере является UTF-8. При использовании других кодировок на сервере, могут быть заданы только коды из диапазона ASCII (до \u007F включительно). Для задания суррогатных пар UTF-16 могут быть использованы как 4-х так и 8-ми разрядные формы, для символов с кодами больше, чем U+FFFF, хотя доступность 8-ми разрядной формы технически делает это ненужным. (Суррогатные пары используются, когда кодировкой сервера является UTF8, они сперва комбинируются в единый код, который затем кодируется в UTF-8.)

ВНИМАНИЕ! Если конфигурационный параметр standard_conforming_strings установлен в off, PostgreSQL распознает экранирование \ как в обычных строковых константах, так и в экранированных. Однако, начиная с версии 9.1, значением по умолчанию является on, при котором символ экранирования \ распознается только в экранированных строках. Такое поведение больше соответствует требованиям стандарта, но может привести к нарушению работы приложений, рассчитывающих на распознавание символа экранирования \ в любых константах. Существует возможность отключения этого параметра, но лучшим

считается отказ от использования символа экранирования \. Если необходимо использовать экранирование \ для представления специального символа, следует записывать константу с E. Кроме параметра standard_conforming_strings поведением, связанным с обработкой \ в строках, управляют также конфигурационные параметры escape_string_warning и backslash_quote.

Символ с нулевым кодом не может быть в строковой константе.

1.1.2.3. Строковые константы с экранированным Unicode

PostgreSQL также поддерживает другой тип синтаксиса экранирования для строк, который позволяет задавать произвольные символы Unicode с помощью их кодов. Строковая константа с экранированным Unicode начинается с U& (U в верхнем или нижнем регистре, за которой следует амперсанд) непосредственно перед открывающей кавычкой, без каких-либо пробелов между ними, например U&'foo'. (Следует отметить, что это создает двусмысленность с оператором &. Использование пробелов вокруг данного оператора, дает возможность избежать данную проблему). Внутри кавычек, символы Unicode могут быть заданы в экранированной форме, с помощью обратной косой черты, за которой следует 4-х разрядный шестнадцатеричный код или, в качестве альтернативы, за обратной косой чертой следует знак плюс и 6-ти разрядный шестнадцатеричный код. Например, строка 'data' может быть записана как:

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

Далее представлен менее простой пример, записывающий русское слово «слон» кириллицей:

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

Если в строке появляется другой символ экранирования, отличный от обратной косой черты, он может быть указан, используя слово UESCAPE после такой строки, например:

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

Символ экранирования может быть любым одиночным символом, отличным от шестнадцатеричной цифры, знаком плюс, одиночной кавычкой, двойной кавычкой или символом пробела. Символ экранирования записывается в одиночных кавычках, а не двойных.

Синтаксис экранирования Unicode полностью работает только когда кодировкой на сервере является UTF-8. При использовании других кодировок на сервере, могут быть заданы только коды из диапазона ASCII (до \u007F включительно). Для задания суррогатных пар UTF-16 могут быть использованы как 4-х так и 6-ти разрядные формы, для символов с кодами больше, чем U+FFFF, хотя доступность 6-ти разрядной формы технически делает это ненужным. (Суррогатные пары не сохраняются напрямую, но комбинируются в единый код, который затем кодируется в UTF-8.)

Кроме того, синтаксис экранирования Unicode работает только в случае установки

конфигурационного параметра `standard_conforming_strings` в значение `on`. Это связано с тем, что в противном случае такой синтаксис может ввести в заблуждение клиентские приложения, анализирующие код SQL, что может привести к возможности использования SQL инъекций или похожих уязвимостей безопасности информации. В случает установки этого конфигурационного параметра в значение `off`, рассматриваемый синтаксис будет отклонен с сообщением об ошибке.

При необходимости включения символа экранирования непосредственно в идентификатор, его необходимо указывать дважды.

1.1.2.4. Строковые константы, экранированные знаками доллара

Несмотря на то, что стандартный синтаксис для задания строковых констант обычно удобен, он может быть труден для понимания, если строка содержит множество одинарных кавычек или символов обратной косой черты, так как каждый из этих символов должен быть сдвоен. Чтобы в таких ситуациях сделать запросы более читабельными, PostgreSQL предоставляет другой способ написания строковых констант – «между знаками доллара». Строковые константы в знаках доллара состоят из знака доллара (`$`), необязательного «тэга» из нуля или более символов, другого знака доллара, произвольной последовательности символов, которые представляют собой содержательную часть строки, знака доллара, такого же тэга, который был вначале и завершающего знака доллара. Ниже приводятся два разных способа задания строки «Dianne's horse» с помощью заключения в знаки `$`:

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

Внутри заключенной в знаки `$` строки одинарные кавычки могут быть использованы без экранирования. Строка в этом случае всегда записывается литерально. Символы `\` здесь не являются специальными также как и знаки `$`, если только они не входят в последовательность, совпадающую с открывающим тэгом.

Можно использовать вложенные строковые константы, заключенные в знаки `$`, если выбирать разные тэги для каждого уровня вложенности. Это чаще всего используется при определении функций. Например:

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

Здесь, последовательность `$q$[\t\r\n\v\\]$q$` представляет заключенную в `$` литеральную строку `[\t\r\n\v\\]`, которая будет распознана, когда PostgreSQL запустит тело функции. Но поскольку данная последовательность не совпадает с внешним долларовым разделителем `$function$`, то она просто является вложенной константой по

отношению к внешней строковой константе.

Тэг строки, заключенной в $, следует тем же правилам, что и не заключенный в $ идентификатор, за исключением того, что он не может содержать знака доллара. Тэги являются зависимыми от регистра, так что tagСодержимое строкиtag — это правильно, а TAGСодержимое строкиtag — неправильно.

Заключенные в $ строки, следующие за ключевым словом или идентификатором, должны отделяться от него пробелами; в противном случае долларовый разделитель будет считаться частью предшествующего ему идентификатора.

Заключенные в $ строки не являются частью стандарта SQL, но зачастую являются более удобным способом записи сложных строковых литералов, чем соответствующий стандарту синтаксис с одинарными кавычками. Этот способ особенно подходит для представления одних строковых констант внутри других, что часто необходимо в определениях функций. При использовании синтаксиса с одинарной кавычкой каждый символ \ в приведенном выше примере должен был бы быть записан как четыре символа \, которые бы подавили два символа \ при разборе первоначальной строки, а затем один символ \ при повторном разборе внутренней строки во время выполнения функции.

1.1.2.5. Битово-строковые константы

Битово-строковые константы выглядят как обычные строковые константы с символом B (в нижнем или верхнем регистре), который идет сразу перед открывающей кавычкой (без пробелов), например B'1001'. В битово-строковой константе допускаются только символы 0 и 1.

Кроме того, битово-строковые константы могут быть заданы в шестнадцатеричной нотации, используя лидирующий символ X (в верхнем или нижнем регистре), например X'1FF'. Такая нотация эквивалентна битово-строковой константе с четырьмя двоичными разрядами для каждого шестнадцатеричного разряда.

Обе формы битово-строковых констант могут занимать несколько строк таким же образом, как и обычные строковые константы. Экранирование символом $ не может быть использовано в битово-строковых константах.

1.1.2.6. Числовые константы

Числовые константы принимаются в следующих общих формах:

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

где digits — это одна или более десятичных цифр (от 0 до 9). По крайней мере одна цифра должна следовать до или после десятичной точки, если она используется. По крайней

мере одна цифра должна следовать за символом экспоненты (e), если этот символ есть. В константе не должно быть пробелов или других символов. Все знаки плюс или минус вначале константы не являются фактической частью константы; эти знаки являются операторами, которые применяются к константе.

Вот несколько примеров правильных числовых констант:

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

Числовая константа, которая не имеет ни десятичной точки, ни символа экспоненты, считается имеющей тип integer (целое), если ее значение умещается в тип integer (32 бита); в противном случае считается, что константа имеет тип bigint (большое целое), если ее значение умещается в тип bigint (64 бита); в противном случае считается, что константа имеет тип numeric. Константы, которые содержат десятичную точку и/или символ экспоненты всегда считаются имеющими тип numeric.

Начальная интерпретация типа числовой константы — это только первый шаг алгоритма определения ее действительного типа. В большинстве случаев константа будет автоматически приведена к наиболее соответствующему ей типу, в зависимости от контекста. При необходимости можно заставить числовое значение интерпретироваться как конкретный тип данных через его указание. Например, можно заставить числовое значение интерпретироваться как тип real (float4), написав

```
REAL '1.23'  -- строковый стиль
1.23::REAL   -- стиль PostgreSQL
```

Существует несколько специальных случаев нотаций приведения типов, которые обсуждаются далее.

1.1.2.7. Константы других типов

Константу *произвольного* типа можно ввести с помощью одной из следующих нотаций:

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

Тест строковой константы передается на вход подпрограммы конвертации для типа type. Результатом является константа указанного типа. Явное указание типа может быть опущено, если нет неоднозначности в понимании того типа, которому должна соответствовать константа (например, когда она напрямую назначается для столбца таблицы), в этом

случае она будет автоматически преобразована к нужному типу.

Строковая константа может быть написана или через обычную SQL-нотацию или через экранирование знаком `$`.

Также возможно задать преобразование типа с помощью синтаксиса в стиле функции
`typename ( 'string' )`
но таким способом могут быть использованы не все имена типов (подробности приведены в 1.2.9).

`CAST()`, `::` и синтаксис в стиле функции могут также быть использованы для задания преобразования типов произвольных выражений в момент их вычисления, как описывается в 1.2.9. Чтобы избежать путаницы, синтаксис `type 'string'` может быть использован только для задания типа простой литеральной константы. Другое ограничение на синтаксис `type 'string'` состоит в том, что такая конструкция не работает для типов массивов; необходимо использовать `::` или `CAST()` для задания типа массива констант.

Синтаксис `CAST()` соответствует стандарту SQL. Синтаксис `type 'string'` является производным стандарта: SQL определяет данный синтаксис только для некоторых типов данных, но PostgreSQL разрешает его для всех типов. Синтаксис с `::` является «историческим» для PostgreSQL, потому как является синтаксисом вызова функций.

1.1.3. Операторы

Имя оператора — это последовательность из `NAMEDATALEN-1` символов (по умолчанию 255) из следующего списка:

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

Для имен операторов существует несколько ограничений:

1) `--` и `/*` не могут быть в имени оператора, т. к. они будут восприняты как начало комментария;

2) имя оператора, состоящее из нескольких символов, не может заканчиваться на `+` или `-`, если только это имя также не содержит один из следующих символов:

```
~ ! @ # % ^ & | ` ?
```

Например, `@-` является разрешенным именем оператора, а `*-` — нет. Это ограничение позволяет PostgreSQL производить разбор SQL-запросов, не требуя обязательного наличия пробелов между токенами.

При работе с именами операторов, которые не соответствуют стандарту SQL, необходимо разделять операторы пробелами, чтобы избежать неоднозначности. Например, при задании в левой части оператора `@` нельзя писать `X*@Y`, необходимо писать `X* @Y`, в этом случае PostgreSQL прочтет это выражение как два имени оператора, а не одно.

1.1.4. Специальные символы

Ряд символов, не являющихся алфавитно-цифровыми, имеют специальное значение:

`$` — знак доллара, за которым идут цифры, используется для представления позиционного параметра в теле описания функции или подготавливаемого оператора. В другом контексте `$` может быть частью идентификатора или экранированной им строковой константы;

`()` — круглые скобки имеют собственный смысл в групповых выражениях и при задании приоритета выполнения операций. В других случаях круглые скобки требуются как часть фиксированного синтаксиса определенных команд SQL;

`[]` — квадратные скобки используются для выбора элементов массива (см. 5.15);

`,` — запятая используется в некоторых синтаксических конструкциях для разделения элементов списка;

`;` — точка с запятой завершает команду SQL. Не может быть использована внутри команды, за исключением использования внутри строковой константы или заключенного в кавычки идентификатора;

`:` — двоеточие используется для выбора «среза» из массива (см. 5.15). В определенных диалектах SQL (таких как Embedded SQL) двоеточие используется как префикс имен переменных;

`*` — звездочка используется в некоторых случаях для указания всех столбцов в строке таблицы или составных значений. Она также имеет специальное значение, когда используется как аргумент какой-либо агрегатной функции, показывая, что агрегат не требует какого-либо явного параметра;

`.` — точка используется в числовых константах и для разделения имен схемы, таблицы и поля.

1.1.5. Комментарии

Комментарий — это произвольная последовательность символов, которая начинается с двойного символа дефиса и продолжается до конца строки, например:

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

Кроме того, возможно использовать блок комментария в стиле языка C:

```
/* Многострочный комментарий
  * с вложением: /* вложенный блок комментариев */
  */
```

где комментарий начинается с `/*` и продолжается пока не встретится `*/`. Показанный выше вложенный блок комментариев соответствует стандарту SQL, но не стандарту языка C, таким образом, один комментарий может содержать большие блоки кода, которые также

могут содержать блоки комментариев.

Комментарии удаляются из входного потока перед синтаксическим анализом и заменяются на пробелы.

1.1.6. Приоритет операторов

Приоритет и ассоциативность операторов жестко встроены в анализатор запросов. Это может создать поведение, которое не является интуитивно понятным. Например, логические операторы < и > имеют приоритет, который отличается от приоритета операторов <= и >=. При использовании комбинаций бинарных и унарных операторов иногда требуется добавлять круглые скобки. Например:

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

будет понято как

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

потому что анализатор не знает, что ! задается как постфиксный оператор, а не как инфиксный. Чтобы получить в данном случае желаемое поведение, требуется писать так:

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

В таблице 2 приведены приоритет и ассоциативность операторов в PostgreSQL в порядке снижения.

Таблица 2 – Приоритет операторов в порядке уменьшения

Оператор/Элемент	Ассоциативность	Описание
.	Слева	Разделитель имени таблицы/столбца
::	Слева	Приведение типа в стиле PostgreSQL
[ ]	Слева	Выбор элемента массива
+ -	Справа	Унарный плюс, унарный минус
^	Слева	Знак степени
* / %	Слева	Умножение, деление, целочисленное деление
+ -	Слева	Сложение, вычитание
IS		IS TRUE, IS FALSE, IS NULL и т. п.
ISNULL		Проверка на NULL
NOTNULL		Проверка на не NULL
все другие	Слева	Все другие стандартные и заданные пользователем операторы
IN		Список членов
BETWEEN		Диапазон вхождения
OVERLAPS		Перекрытие временного интервала
LIKE ILIKE SIMILAR		Совпадение строкового шаблона
< >		Меньше чем, больше чем

Окончание таблицы 2

Оператор/Элемент	Ассоциативность	Описание
=	Справа	Присваивание, назначение
NOT	Справа	Логическое отрицание
AND	Слева	Логическое умножение
OR	Слева	Логическое сложение

Данные правила приоритетов операторов также применимы к операторам, определенным пользователем, имеющим такие же имена, как и встроенные операторы, описанные выше. Например, при определении оператора «+» для какого-либо нового типа данных он будет иметь такой же приоритет, как и встроенный оператор «+», независимо от того, кем он создан.

Когда в синтаксисе OPERATOR используется имя оператора вместе с именем схемы, например:

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

сконструированный с помощью OPERATOR оператор будет иметь по умолчанию приоритет, указанный в таблице 2 для оператора «все другие». Это не зависит от того, какое имя оператора используется внутри OPERATOR().

1.2. Выражения, возвращающие одиночное значение

Выражения, возвращающие одиночное значение (value expression), используются в различных контекстах, таких как список выборки в команде SELECT, новые значения столбцов в командах INSERT или UPDATE или для условий поиска в ряде других команд. Результат этого выражения (т. е. одиночное значение), иногда называют *скаляром*, чтобы отличать его от результата выражения, которое возвращает таблицу. Таким образом, выражения, возвращающие одиночное значение также называют *скалярными выражениями* (или даже просто *выражениями*). Синтаксис выражения позволяет вычислять значения из его частей, используя арифметические, логические, списочные и другие операции.

Виды выражений, возвращающих одиночное значение:

- константа или литеральное значение (см. 1.1.2);
- ссылка на столбец;
- ссылка на позиционный параметр (в теле определения функции или подготовленного оператора);
- индексное выражение;
- выражение выбора поля;
- вызов оператора;
- вызов функции;

- агрегатное выражение;
- вызов оконной функции;
- приведение типа;
- выражение сопоставления;
- скалярный подзапрос;
- конструктор массива;
- конструктор строки;
- другое выражение в круглых скобках, использованных для группировки подвыражений и изменения приоритета.

В дополнение к этому списку существует несколько конструкций, которые могут классифицироваться как выражения, но которые не следуют никаким общим синтаксическим правилам. Они обычно имеют семантику функции или оператора (см. раздел 6). Примером, такого выражения является `IS NULL`.

Далее рассмотрены виды скалярных выражений.

1.2.1. Ссылки на столбец

Ссылка на столбец может иметь вид:

```
correlation.columnname
```

где `correlation` — это имя таблицы (возможно полное имя вместе с именем схемы) или псевдоним таблицы, определенный с помощью предложения `FROM`. Имя отношения и отдельная точка могут быть опущены, если имя столбца является уникальным для всех таблиц, которые используются в текущем запросе (раздел 4).

1.2.2. Позиционные параметры

Ссылка на позиционный параметр используется, чтобы указать значение, передаваемые снаружи оператору SQL. Параметры используются в определениях функций SQL и в подготовленных запросах. Некоторые клиентские библиотеки также поддерживают задание значений данных отдельно от строк команды SQL, в этом случае параметры используются, чтобы сослаться на значения данных, которые находятся вне. Форма ссылки на параметр следующая:

```
$number
```

Например, рассмотрим такое определение функции `dept`:

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

Здесь `$1` ссылается на значение первого аргумента функции в момент, когда вызывает функция.

1.2.3. Элементы массива

Если выражение возвращает значение какого-либо типа массива, то конкретный элемент этого массива может быть извлечен из него с помощью:

```
expression[subscript]
```

а несколько соседних элементов («срез массива») могут быть извлечены с помощью:

```
expression[lower_subscript:upper_subscript]
```

Здесь квадратные скобки должны пониматься буквально. Каждый индекс `subscript` сам является выражением, которое должно быть целым числом.

Обычно массив `expression` должен быть в круглых скобках, но круглые скобки могут быть опущены, когда выражение является ссылкой на столбец или позиционным параметром. Несколько индексов могут быть соединены, если оригинальный массив является многомерным. Например:

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

Круглые скобки в последнем примере необходимы. (см. подробности о массивах в 5.15).

1.2.4. Выбор поля

Если выражение возвращает значение какого-либо составного типа (тип записи), то конкретное поле (`fieldname`) из этой записи может быть извлечено с помощью:

```
expression.fieldname
```

Обычно выражение `expression` должно быть в круглых скобках, но круглые скобки могут быть опущены, когда выражение является ссылкой на таблицу или позиционным параметром. Например:

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

(Таким образом, ссылка на столбец в полном виде является специальным случаем синтаксиса выборки поля.) Отдельным случаем является выборка поля из столбца таблицы, имеющим составной тип:

```
(compositecol).somefield
(mytable.compositecol).somefield
```

В этом случае круглые скобки указывают, что `compositecol` является именем столбца, а не таблицы, а во втором примере, что `mytable` — имя таблицы, а не схемы.

В списке выборки команды `select` возможно указание всех полей составного значения с помощью `.*`:

```
(compositecol).*
```

1.2.5. Вызовы операторов

Существуют три возможных синтаксиса для вызова операторов:

`expression operator expression` (бинарный инфиксный оператор)

`operator expression` (унарный префиксный оператор)

`expression operator` (унарный постфиксный оператор)

где токен `operator` следует синтаксическим правилам (см. 1.1.3) или является одним из ключевых слов `AND`, `OR` и `NOT` или полным именем оператора в форме:

```
OPERATOR(schema.operatorname)
```

Существование отдельных операторов, и являются ли они унарными или бинарными зависит от того, какие операторы были заданы системой или пользователем. В разделе 6 описываются встроенные операторы.

1.2.6. Вызовы функций

Синтаксис вызова функции является именем этой функции (возможно с указанием имени схемы), за которым следуют список аргументов функции, заключенный в круглые скобки:

```
function ([expression [, expression ... ]] )
```

Например, следующая функция вычислит квадратный корень из 2:

```
sqrt(2)
```

Список встроенных функций приведен в разделе 6. Пользователем могут быть добавлены и другие функции.

Аргументы при необходимости могут иметь прикрепленные имена (см. 1.3).

Примечание. Функции, принимающие один аргумент составного типа, могут быть при необходимости вызваны с использованием синтаксиса выбора поля, и наоборот, выбор поля может быть записан как вызов функции. Таким образом, нотация `col(table)` и `table.col` взаимозаменяемы. Такое поведение не является SQL совместимым, но предлагается в PostgreSQL, так как позволяет использовать функции для эмуляции «вычисляемых полей».

1.2.7. Агрегатные выражения

Агрегатное выражение представляет какую-либо агрегатную функцию, вызываемую для строк, выбранных в запросе. Агрегатная функция получает несколько значений, а выдает только одно значение, такое как сумма или среднее значение. Синтаксис агрегатного выражения может быть одним из следующих вариантов:

```
aggregate_name (expression [ , ... ] [ order_by_clause ] )
aggregate_name (ALL expression [ , ... ] [ order_by_clause ] )
aggregate_name (DISTINCT expression [ , ... ] [ order_by_clause ] )
```

```
aggregate_name ( * )
```

где имя агрегата `aggregate_name` является заданным ранее агрегатом (возможно с указанием имени схемы), выражение `expression` является любым значимым выражением, которое само не содержит какого-либо агрегатного выражения, а `order_by_clause` является необязательным предложением `ORDER BY` как описано ниже.

Первая форма агрегатного выражения вызывает агрегат для всех строк на входе. Вторая форма синтаксиса является такой же как и первая, за исключением того, что `ALL` является значением по умолчанию. Третья форма вызывает агрегат для каждого уникального значения выражения(или уникального набора значений, для нескольких выражений), найденного во входном наборе строк. Последняя форма синтаксиса вызывает агрегат один раз для каждой строки на входе, в ней не указывается конкретных значений, и обычно эта форма полезна только для агрегатной функции `count(*)`.

Большинство агрегатных функций игнорируют значения `NULL`, так что строки, для которых одно или более выражений принимают значение NULL — отбрасываются. Если не указано другого, подобное поведение принимается для всех встроенных агрегатов.

Например, `count(*)` выдает полное количество строк на входе; `count(f1)` выдает количество строк на входе, в которых `f1` является `не NULL`, поскольку `count` их игнорирует; `count(distinct f1)` выдает количество уникальных не NULL-значений `f1`.

Обычно входной набор строк передается агрегатной функции в произвольном порядке. Во многих случаях это не имеет значения; например, `min` выдает один и тот же результат независимо от порядка поступления входных значений. Однако, некоторые агрегатные функции (такие как `array_agg` и `string_agg`) выдают результат, зависящий от порядка поступления входных строк. При использовании такого агрегата с помощью необязательной конструкций `order_by_clause` может быть задан желаемый порядок. Конструкция `order_by_clause` имеет тот же синтаксис, что и `ORDER BY` в запросах (см. 4), за тем исключением, что используемые выражения всегда являются только выражениями и не могут быть именами или номерами колонок на выходе.Например:

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

При работе с агрегатными функциями, принимающих несколько аргументов, конструкция `ORDER BY` должна следовать после всех аргументов. Например, следующая запись верна:

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

а такая — нет:

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

Последнее синтаксически верно, но представляет вызов агрегатной функции, принимающий один аргумент, с двумя ключами `ORDER BY` (второй становится бессмысленным, так как является константой.)

Если дополнительно к order_by_clause указано ключевое слово DISTINCT, все выражения ORDER BY, должны соответствовать аргументам данного агрегата; таким образом, невозможно выполнять сортировку выражения, которое не включено в список DISTINCT.

Примечание. Возможность указывать одновременно DISTINCT и ORDER BY в агрегатных функциях является расширением PostgreSQL.

Предопределенные агрегатные функции описываются в 6.20. Пользователем могут быть добавлены другие агрегатные функции.

Агрегатное выражение может применяться только в списке результата или с ключевым словом HAVING в команде SELECT. Использование с другими ключевыми словами, такими как WHERE, запрещено, потому что эти ключевые слова логически выполняются перед тем, как формируются результаты агрегатов.

Когда агрегатное выражение появляется в подзапросе (см. 1.2.11 и 6.22), агрегат обычно выполняется для строк подзапроса. Но есть исключение, если аргументы агрегата содержат только переменные внешнего уровня: агрегат выходит на ближайший внешний уровень и выполняется для строк запроса на этом уровне. Тогда агрегатное выражение целиком является внешней ссылкой для подзапроса, в котором оно используется и работает как константа для любых действий этого подзапроса. Ограничение касательно применения только в списке результата или с ключевым словом HAVING работает в соответствии с уровнем запроса, в который выходит агрегат.

1.2.8. Вызовы оконных функций

Вызов оконной функции представляет собой применение функции, подобной агрегирующей на некоторой порции строк, отобранных запросом. В отличие от вызова обычной агрегатной функции, группировка выбранных строк в одну на выходе запроса не производится — каждая строка остается на выходе запроса отдельной строкой. Однако, оконная функция позволяет сканировать все строки, которые могут быть частью группы, в которую входит текущая строка, в соответствии со спецификацией группировки (списка PARTITION BY) вызова оконной функции. Синтаксис вызова оконной функции может быть одним из следующих:

```
fuction_name ([expression [, expression ...]]) OVER
( window_definition )
fuction_name ([expression [, expression ...]]) OVER window_name
fuction_name (*) OVER ( window_definition )
fuction_name (*) OVER window_name
```

Где конструкция window_definition имеет следующий синтаксис:

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ]
```

```
[ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

В качестве необязательного выражения frame_clause может использоваться одна из следующих конструкций:

```
[ RANGE | ROWS ] frame_start
[ RANGE | ROWS ] BETWEEN frame_start AND frame_end
```

Где выражения frame_start и frame_end могут принимать следующие значения:

```
UNBOUNDED PRECEDING
value PRECEDING
CURRENT ROW
value FOLLOWING
UNBOUNDED FOLLOWING
```

Здесь, expression представляет собой любое выражение, которое само по себе не содержит вызова оконной функции. window_name является ссылкой на именованную спецификацию окна, определенную в запросе выражением WINDOW. Именованные спецификации окна обычно используются в виде OVER wname, но также возможно задание имени окна внутри круглых скобок и с необязательным добавлением предложения сортировки и/или предложения фрейма (имя окна должно быть в этих предложениях, если они есть). Последняя форма синтаксиса следует тем же правилам, что и изменение существующего имени окна внутри предложения WINDOW; дополнительная информация приведена в описании команды SELECT.

Опция PARTITION BY группирует строки запроса в разбиения, которые обрабатываются раздельно оконной функцией. PARTITION BY работает схоже с предложением GROUP BY обычного запроса, за тем исключением, что выражения должны быть именно выражениями, а не именами или номерами выводимых столбцов. Без опции PARTITION BY все строки запроса рассматриваются как одно разбиение. Опция ORDER BY задает порядок обработки оконной функцией строк запроса в одном разбиении. Это выполняется схоже с предложением ORDER BY обычного запроса, но так же не допускаются имена или номера выводимых столбцов. Без указания опции ORDER BY строки обрабатываются в произвольном порядке.

Конструкция frame_clause определяет набор строк, составляющих фрейм окна, для тех оконных функций, которые действуют в пределах фрейма, а не полной выборки строк запроса. Фрейм может быть задан в режимах RANGE и ROWS; в обоих случаях он выполняется с frame_start по frame_end. Если опущено frame_end, в качестве него принимается значение CURRENT ROW (текущая строка).

Значение UNBOUNDED PRECEDING для frame_start означает, что фрейм начинается с первой строки разбиения, и аналогично значение UNBOUNDED FOLLOWING для

frame_end — что он заканчивается последней строкой разбиения.

В режиме RANGE, значение CURRENT ROW для frame_start означает, что фрейм начинается с первого появления текущей строки в ORDER BY (строки которая рассматривается ORDER BY эквивалентной текущей), а для frame_end — что фрейм заканчивается на последнем появлении строки эквивалентной текущей. В режиме ROWS значение CURRENT ROW означает непосредственно текущую строку.

Значение PRECEDING и значение FOLLOWING в настоящий момент разрешены только в режиме ROWS. Они показывают, что фрейм начинается или заканчивается строкой, которая находится за множеством строк перед или после текущей строки. Значение должно быть целым и не содержать каких-либо переменных, агрегатных функций или оконных функций. Также оно не должно быть NULL или отрицательным; но может быть нулем, который выбирает саму текущую строку.

По умолчанию задающей фрейм опцией является RANGE UNBOUNDED PRECEDING, что эквивалентно RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. С указанием опции OREDER BY это означает выборку всех строк с начала разбиения до последнего появления текущей строки в сортировке. Без указания опции OREDER BY во фрейм включаются все строки выборки разбиения, так как все строки становятся эквивалентными текущей.

Существуют ограничения, такие что frame_start не может быть UNBOUNDED FOLLOWING, frame_end не может быть UNBOUNDED PRECEDING, и выбор frame_end, в данном выше списке, не может осуществляться раньше, чем выбор frame_start — например, RANGE BETWEEN CURRENT ROW AND value PRECEDING не допускается.

Встроенные оконные функции рассмотрены в 6.21. Другие оконные функции могут быть определены пользователем. Кроме того, любые встроенные или определенные пользователем агрегатные функции могут быть использованы как оконные функции.

Синтаксис * применяется для вызова агрегатных функций без параметров как оконных функций, например count(*) OVER (PARTITION BY x ORDER BY y). Синтаксис * специально не используется для неагрегатных оконных функций. Агрегатные оконные функции, в отличие от обычных агрегатных функций, не разрешают использование выражения DISTINCT или ORDER BY в пределах списка аргументов функции.

Вызов оконных функций разрешен в запросе только в списке SELECT или предложении ORDER BY.

Дополнительная информация приведена в 4.2.4.

1.2.9. Приведения типа

Приведение типов определяет механизм преобразования из одного типа данных в другой. Для приведений типов PostgreSQL позволяет использовать два эквивалентных варианта синтаксиса:

```
CAST ( expression AS type )
expression::type
```

Синтаксис `CAST` совместим со стандартом SQL; синтаксис с `::` является историческим для PostgreSQL.

Когда приведение типа выполняется для значения выражения одного из известных типов, оно выполняется как преобразование типа в момент выполнения («на лету»). Такое приведение типа будет успешным, только если определена подходящая операция преобразования типов. Следует отметить существенное отличие от использования приведения типа с константами, как было показано в 1.1.2.7. Приведение типа, которое выполняется для строковых литералов, производится как начальное назначение типа значению константы-литерала и, таким образом, оно будет успешным для любого типа (если содержимое строкового литерала соответствует синтаксису ввода значений для данного типа данных).

Явное приведение типа может быть опущено, если нет двусмысленности в том, какое значение выражения должно получиться для типа (например, когда это значение заносится в столбец таблицы); в этом случае система автоматически выполнит приведение типа. Однако автоматическое приведение выполняется только для случаев, которые отмечаются как «OK to apply implicitly» (подходит для неявного применения) в системных каталогах. Другие приведения типов должны быть вызваны с использованием синтаксиса явного приведения типа. Данное ограничение предотвращает неожиданные преобразования при приведении типов без каких-либо предупреждающих сообщений.

Также возможно задать преобразование типа, используя синтаксис, похожий на вызов функции:

```
typename ( expression )
```

Это работает только для типов, чьи имена также являются допустимыми в качестве имен функций. Например, `double precision` в данном случае использовать нельзя, но можно использовать эквивалент данного типа `float8`. Также имена `interval`, `time` и `timestamp` можно использовать, только если они заключены в двойные кавычки, иначе будут синтаксические конфликты. Таким образом, использование синтаксиса приведения типа в виде функций приводит к противоречиям и предположительно должно избегаться.

Примечание. Синтаксис в форме вызова функции фактически является вызовом функции. Когда для приведения типов, во время выполнения, используются один из двух вышеописанных стандартов синтаксиса, это приводит к внутреннему вызову зарегистри-

рованной функции для выполнения преобразования типа. Эти функции преобразования имеют те же имена, что и тип данных на их выходе и, таким образом, «синтаксис в форме функции» является прямым вызовом требуемой функции преобразования. Дополнительная информация приведена в описании CREATE CAST.

1.2.10. Выражения сопоставления

Ключевое слово COLLATE переопределяет способ сопоставления для выражения. Ключевое слово добавляется в конец выражения:

```
expr COLLATE collation
```

где collation является идентификатором способа сортировки (возможно с указанием имени схемы). Ключевое слово COLLATE имеет приоритет ниже чем у операторов; при необходимости могут быть использованы скобки.

Если сопоставление явно не указано, то СУБД или получает значение сопоставления из столбцов, задействованных в выражении, или, если никакие столбцы в выражении не задействованы, оно берется из значения по умолчанию для базы данных.

Два основных применения ключевого слова COLLATE состоят в переопределении порядка сортировки в выражении ORDER BY, например:

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

и в переопределении способа сортировки при вызове функции или оператора, имеющих результат, чувствительный к кодировке, например:

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

Следует отметить, что в последнем случае выражение COLLATE присоединено к входному аргументу оператора, на который должно быть оказано влияние. Не имеет значения к какому именно аргументу присоединено выражение COLLATE, поскольку способ сортировки, применяемый оператором или функцией, определяется по всем аргументам, и явно заданное выражение COLLATE переопределяет сортировку всех остальных аргументов. (Присоединение не совпадающих выражений COLLATE более чем к одному аргументу является ошибкой. Более подробно это описано в разделе 22.2 оригинальной документации на PostgreSQL.) Таким образом, следующее выражение дает такой же результат, как и предыдущее:

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

Тогда как следующее выражение будет ошибочным:

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

поскольку содержит попытку указать способ сортировки результата оператора >, который работает с логическим типом данным boolean, не поддерживающим сопоставления.

1.2.11. Скалярные подзапросы

Скалярный подзапрос — это обычный запрос SELECT в круглых скобках, который возвращает ровно одну строку с одним столбцом (см. информацию о написании запросов в

разделе 4). После выполнения запроса SELECT возвращаемое значение используется во внешнем выражении. Использование запроса, который возвращает более чем одну строку или более чем один столбец в качестве скалярного подзапроса, является ошибкой. (Но если во время выполнения подзапрос не вернул ни одной строки — это не является ошибкой; такой результат считается значением NULL.) Подзапрос может ссылаться на переменные из внешнего запроса, которые во время выполнения подзапроса будут использоваться как константы. Информация о других выражениях, используемых в подзапросах, приведена в 6.22.

Следующий пример ищет наибольшее количество людей, проживающих в городах для каждого штата:

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

1.2.12. Конструкторы массивов

Конструктор массивов — это выражение, которое осуществляет построение массива из значений его элементов. Простой конструктор массива содержит одно ключевое слово ARRAY, левую квадратную скобку [, список выражений (разделенных запятыми) для элементов массива и, наконец, правую квадратную скобку]. Например:

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

По умолчанию тип элемента массива является общим типом выражений для членов массива и определяется по тем же правилам, что и в UNION или CASE. Можно переопределить данный тип, явно указав приведение конструктора массива к нужному типу, например:

```
SELECT ARRAY[1,2,22.7]::integer[];
  array
----------
 {1,2,23}
(1 row)
```

Такое приведение типа дает тот же эффект, что и приведение типа каждого элемента массива индивидуально. Больше о приведении типов см. 1.2.9.

Значения многомерного массива могут быть построены с помощью вложенных конструкторов массивов. Во внутренних конструкторах ключевое слово ARRAY можно опустить. Например, следующие примеры дадут один и тот же результат:

```
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)
```

Поскольку многомерные массивы должны быть прямоугольными, внутренние конструкторы на том же уровне должны выдавать подмассивы одинаковой размерности. Любые приведения типов, применяемые к внешнему конструктору ARRAY, автоматически применяются ко всем внутренним конструкторам.

Элементы многомерного конструктора массива могут быть любыми массивами правильного вида, а не только конструкторами подмассивов. 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)
```

Можно сконструировать пустой массив, но поскольку нельзя создать массив без типа, необходимо явно привести пустой массив к нужному типу. Например:

```
SELECT ARRAY[]::integer[];
 array
-------
   {}
(1 row)
```

Также возможно сконструировать массив из результатов подзапроса. В этой форме конструктор массива записывается с ключевым словом ARRAY, за которым следуют круглые скобки (не квадратные) с подзапросом. 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)
```

Подзапрос должен возвращать один столбец. Результирующий одномерный массив будет состоять из элементов строк, которые возвратил подзапрос, а тип элементов будет соответствовать типу столбца.

Индексирование элементов массива, конструируемого с ARRAY, всегда начинается с единицы. Дополнительная информация о массивах приведена в 5.15.

1.2.13. Конструкторы строк (записей)

Примечание. Ниже под строками понимаются строки таблиц — записи, которые возвращаются запросами, а не строки символов.

Конструктор строк — это выражение, которое осуществляет построение значения строки (также называемое «составным значением») из значений полей этой строки. Конструктор строки состоит из ключевого слова ROW, за которым следует левая круглая скобка (, нуль или более выражений (разделенных запятыми) значений полей данной строки и, в конце — правая круглая скобка). Например:

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

Ключевое слово ROW является необязательным, если в списке более одного выражения.

Конструктор строки может включать синтаксис rowvalue.*, который будет расширен в список элементов значения строки, только если синтаксис .* используется в списке оператора SELECT верхнего уровня. Например, если таблица t имеет столбцы f1 и f2, то два следующих оператора являются одинаковыми:

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

Примечание. В PostgreSQL до версии 8.2 синтаксис .* не расширялся, и написание ROW(t.*, 42) создавало строку из двух полей, где первое поле было другим значением типа запись. Новое поведение более удобно. Для использования старого поведения встроенных значений строк необходимо записывать внутреннее значение строки без .*, например ROW(t, 42).

По умолчанию значение, созданное с помощью выражения ROW, имеет неизвестный тип записи. При необходимости, оно может быть приведено к именованному составному типу или к типу строки таблицы, или к составному типу, созданному с помощью CREATE TYPE AS. Во избежание двусмысленности может требоваться явное приведение типа. Например:

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);
CREATE FUNCTION getf1(mytable)
  RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;
-- Приведение типа не нужно, потому что
-- существует только одна функция 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 типа необходимо,
-- чтобы показать какая функция вызывается:
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique
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)
```

Конструкторы строк могут быть использованы для построение составных значений, для их сохранения в столбцах таблицы с составным типом данных или для передачи этих значений в функцию, которая работает с параметром составного типа. Также является возможным сравнить два строковых значения или протестировать строку на `IS NULL` или `IS NOT NULL`, например:

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');
SELECT ROW(table.*) IS NULL FROM table;  -- определить все NULL-строки
```

Подробнее это рассмотрено в 6.23. Конструкторы строк могут также быть использованы в подзапросах (см. 6.22).

1.2.14. Правила вычисления выражений

Порядок вычисления выражений не определен. В частности, ввод оператора или функции необязательно вычисляется слева направо или в другом фиксированном порядке.

Тем не менее, если результат выражения может быть определен вычислением только некоторых его частей, то все другие подвыражения могут и не вычисляться. Например, если пишется:

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

то somefunc() (предположительно) не должна вызываться. Тоже самое должно быть в случае:

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

Это не тоже самое, что «кратчайший путь» слева направо для логических операторов, как в некоторых языках программирования.

Не следует использовать функции с посторонними эффектами как часть сложных выражений. Особенно опасно полагаться на посторонние эффекты или порядок вычисления в предложениях WHERE и HAVING, так как эти выражения сильно перерабатываются как часть разработки плана выполнения запроса. Логические выражения (комбинации AND/OR/NOT) в таких предложениях могут быть переорганизованы в любую форму, которую позволяют правила логической арифметики.

Когда необходимо соблюсти порядок вычисления, можно использовать конструкцию CASE (см. 6.17). Например, небезопасный способ избежать деления на ноль в выражении WHERE:

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

Безопасный способ:

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

Конструкция CASE, используемая в данном случае, предотвращает попытки оптимизации. (В данном примере будет лучше решить эту проблему, используя вместо написанного y > 1.5*x.)

1.3. Вызовы функций

PostgreSQL позволяет функциям, которые имеют именованные параметры быть вызванными в *позиционной* или в *именованной* нотации. Именованная нотация особенно удобна для функций, которые имеют большое количество параметров, так как данная нотация более явно и надежно выполняет сопоставления между параметрами и фактическими аргументами. В позиционной нотации, вызов функции записывается с значениями ее аргументов в том же порядке, в котором аргументы определены в объявлении функции. В именованной нотации, аргументы могут следовать в любом порядке, а сопоставление аргументов и параметров функции происходит по именам.

В любой нотации, параметры которые имеют значения по умолчанию, заданные при объявлении функции, нет необходимости прописывать при вызове функции. Но это особенно применимо в именованной нотации, потому что может быть опущена любая комбинация параметров; в то время как в позиционной нотации, параметры могут быть опушены только справа налево.

PostgreSQL также поддерживает *смешанную* нотацию, которая сочетает позици-

онную и именованную нотации. В этом случае, позиционные параметры записываются первыми, а именованные параметры следуют за ними.

Применение всех трех нотаций иллюстрируют следующие примеры, используя следующие определение 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`. В дополнение к ним, есть один необязательный параметр `uppercase`, который по умолчанию установлен в `false`. Вводимые значения `a` и `b` будут соединены вместе, а приведение к верхнему или нижнему регистру будет зависеть от параметра `uppercase`. Остальные подробности определения этой функции сейчас не важны.

1.3.1. Использование позиционной нотации

Позиционная нотация является в PostgreSQL традиционным механизмом для передачи аргументов функциям. Пример:

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

Все аргументы указываются по порядку. Результат будет в верхнем регистре, так как для параметра `uppercase` указано значение `true`. Другой пример:

```
SELECT concat_lower_or_upper('Hello', 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

Здесь, параметр `uppercase` опущен, так что берется значение из умолчания `false`, и результат будет в нижнем регистре. В позиционной нотации, аргументы могут быть опущены

справа налево, столько раз, сколько для них указано значений по умолчанию.

1.3.2. Использование именованной нотации

В именованной нотации, каждое имя аргумента указывается с помощью `:=`, чтобы отделить его от значения аргумента. Например:

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

И снова, аргумент `uppercase был опущен`, так что его значение устанавливается в `false`. Одно из преимуществ использования именованной нотации в том, что аргументы могут быть указаны 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)
```

1.3.3. Использование смешанной нотации

Смешанная нотация сочетает в себе позиционную и именованную нотацию. Однако, как уже говорилось, именованные аргументы не могут следовать перед позиционными аргументами. Например:

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

В данном выше запросе, аргументы `a` и `b` задаются позиционно, в то время как аргумент `uppercase` указывается по имени. Рассмотренный пример прост и служит целям документирования. С более сложной функцией, имеющей множество параметров, у которых есть значения по умолчанию, именованная или смешанная нотация может сохранить множество времени при написании запросов, а также снизить вероятность ошибок.

2. ОПРЕДЕЛЕНИЕ ДАННЫХ

В разделе приведены сведения о том, как в существующей базе данных создавать структуры данных. В реляционной базе данных все данные хранятся в таблицах, так что главная цель раздела — рассказать о том, как создавать и изменять таблицы, а также о существующих возможностях по управлению данными, которые хранятся в таблицах. В дополнение, приводится информация, как можно организовать таблицы внутри схем и какие привилегии могут быть предоставлены для таблиц. В заключении, рассмотрены другие возможности, которые влияют на хранение данных, такие как наследование, представления, функции и триггеры.

2.1. Основы таблиц

Таблица в реляционной БД подобна таблице на листе бумаги: она состоит из столбцов и строк. Число и порядок столбцов фиксирован, и каждый из них имеет собственное имя. Число строк меняется в зависимости от объема данных, сохраненных в этой таблице на текущий момент. Язык SQL не предполагает какой-либо определенный порядок строк в таблице. При чтении таблицы строки могут выбираться в произвольном порядке, если только специально не была задана их сортировка (см. раздел 4). Язык SQL не требует, чтобы каждая строка в таблице имела уникальный идентификатор, таким образом, в одной таблице может существовать несколько полностью идентичных строк. Это следует из математической модели, используемой в языке SQL.

Каждый столбец имеет свой тип данных. Тип данных ограничивает список возможных значений, которые могут находится в этом столбце и определяет семантику данных, хранящихся в нем, благодаря чему эти данные могут использоваться при каких-либо вычислениях. Например, если объявлено, что столбец имеет числовой тип данных, то в него не могут быть помещены произвольные строки текста, а данные, хранимые в этой колонке могут быть использованы для математических вычислений. В противоположность этому, если объявлено, что столбец имеет тип данных символьная строка, то в него можно поместить любой вид данных, но сами эти данные не могут быть использованы для математических вычислений, в то время как для этих данных доступны другие операции, например такие, как слияние строк.

PostgreSQL включает большой список встроенных типов данных, которые подходят для работы многих приложений. Пользователи также могут задавать свои собственные типы данных. Большинство встроенных типов данных имеют очевидные имена и семантику (см. 5). Наиболее часто используемые типы данных:

- `integer` — для целых чисел;
- `numeric` — для чисел с плавающей точкой;

- text — для текстовых строк;
- data — для календарных дат;
- time — для времени суток;
- timestamp — для хранения временных отметок, состоящих из времени и даты.

Для создания таблицы используется команда CREATE TABLE. В этой команде необходимо как минимум задать имя создаваемой таблицы, имена столбцов и их типы. Например:

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

Данная команда создаст таблицу с именем my_first_table и двумя столбцами. Первый столбец имеет имя first_column и тип text, а второй — second_column и тип integer. Имена таблиц и колонок должны выбираться по синтаксическим правилам формирования идентификаторов, о которых рассказывается в 1.1.1. Имена типов, за некоторым исключением, так же являются идентификаторами. Список столбцов должен быть разделен запятыми и заключен в круглые скобки.

Как правило, таблицам и колонками даются такие имена, которые говорят о том, какие данные в них хранятся. Вот пример, который более приближен к реальности:

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

(Столбцы типа numeric могут хранить дробные значения, например такие как денежные.)

Примечание. При создании множества взаимосвязанных таблиц, рекомендуется использовать единый и непротиворечивый подход к выбору имен для таблиц и столбцов.

Количество столбцов таблицы имеет предел. Конкретное максимальное число столбцов в таблице зависит от их типов и колеблется от 250 до 1600. Однако, создание таблиц, содержащих такое множество колонок является очень неэффективным подходом и часто является следствием некорректного проектирования базы данных.

В случае, если таблица больше не нужна, ее можно удалить с помощью команды DROP TABLE. Например:

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

Попытка удалить несуществующую таблицу будет воспринята системой как ошибка. Однако существует множество SQL-скриптов, которые пытаются удалить таблицы без про-

верки их существования перед повторным созданием, игнорируя сообщения об ошибках, что позволяет им выполняться независимо от существования таблиц. (Возможно использование варианта конструкции `DROP TABLE IF EXISTS` для избежания сообщений об ошибках, но это не входит в стандарт SQL.)

Изменение существующей таблицы описано в 2.5.

Приведенных сведений достаточно для создания полнофункциональных таблиц. Далее описываются дополнительные возможности по созданию таблиц, которые позволяют быть уверенными в целостности данных, безопасности или удобстве.

2.2. Значения по умолчанию

Для столбца может быть задано значение по умолчанию, которое будет использовано при вставке новой строки и отсутствии явно заданного значения для данного столбца. Кроме того, команда манипулирования данными может явно затребовать установить значение по умолчанию для какого-либо столбца, при этом не зная о самом значении. (Подробности о командах манипулирования данными можно найти в 3)

Если значение по умолчанию явно не задано, то значением по умолчанию является `null`. Обычно это правильно, потому что значение `null` может быть использовано для обозначения неизвестных данных.

При создании таблицы значение по умолчанию указывается следом за типом данных. Например:

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

Значение по умолчанию может также быть каким-либо выражением, которое будет выполнятся во время вставки значения по умолчанию (а не во время создания таблицы). Типичный пример — это столбец с типом данных, который объединяет дату и время — `timestamp`, где значением по умолчанию является выражение `CURRENT_TIMESTAMP`. При вставке новой строки, это выражение получает текущую дату и время. Другой типичный пример — это генерация «уникального значения» для каждой строки. В PostgreSQL это обычно реализуется следующим образом:

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

где `nextval()` — функция, возвращающая следующее значение из объекта «последовательность» (см. 6.16). Аналогичного результата можно достичь, используя сокращенную

форму:

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

Сокращение SERIAL рассматривается в 5.1.4.

2.3. Ограничения целостности

Типы данных являются одним из способов ограничить набор значений, которые могут быть сохранены в столбцах таблицы. Однако для многих приложений предоставляемых ими ограничений недостаточно. Например, столбец, который должен содержать цены на товары, должен хранить только положительные числа. Но стандартного типа данных, который бы допускал только положительные числа не существует. В других случаях, может понадобится ограничить данные в колонке в соответствии с значениями других столбцов или строк. Например, в таблице, содержащей информацию о товарах должна быть только одна строка для каждого товара.

Для решения подобных задач язык SQL позволяет задать ограничения для столбцов и таблиц. Ограничения позволяют гибко контролировать данные в таблицах. Они не дадут пользователю занести в таблицу данные, которые нарушат заданные ими правила. Это справедливо даже для значений по умолчанию.

2.3.1. Ограничения Check (проверки)

Ограничение целостности check (проверка) является наиболее часто используемым видом ограничения. Оно позволяет задать для определенного столбца, выражение, которое будет осуществлять проверку, помещаемого в этот столбец значения. Если значение удовлетворяет, заданному ограничению, то выражение должно возвращать Логическое значение «истина». Например, требование того, что бы цены продуктов были положительными, можно задать так:

```
product prices, you could use:
  CREATE TABLE products (
  product_no integer,
  name text,
  price numeric CHECK (price > 0)
);
```

Определение проверки следует за определением типа столбцов так же, как и определение значения по умолчанию. Проверки и значение по умолчанию могут быть заданы в произвольном порядке. Проверка состоит из ключевого слова CHECK и следующего за ним выражения в скобках. Проверка должна включать в себя ограничиваемый столбец, в

противном случае, она не имеет смысла.

Для проверки существует возможность задания имени. Это делает сообщения о нарушении такого ограничения более информативными и позволяет изменять эту проверку, ссылаясь на нее по имени. Например:

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

Имя проверки задается с использованием ключевого слова `CONSTRAINT`, за которым следуют имя и ее определение. (Если имя ограничения не было задано, система сама выберет для проверки имя.)

В проверке может использоваться несколько столбцов. Например, если в таблице хранится две цены товара — розничная и закупочная, то при занесении данных целесообразно проверять, чтобы розничная цена была не ниже закупочной:

```
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 две проверки аналогичны рассмотренной ранее, третья использует новый синтаксис. Она не привязана к конкретному столбцу, а представлена отдельным элементом в перечислении столбцов. Определения столбцов и определения подобных проверок могут появляться в произвольном порядке.

Первые две проверки являются проверками столбцов, тогда как третья – табличной проверкой, поскольку она задана отдельно от определения конкретного столбца. Проверки для столбцов всегда могут быть записаны как проверки для таблиц. Тогда как обратное не всегда возможно, поскольку ограничение на столбец ссылается только на столбец, на который оно накладывается. (PostgreSQL не требует этого, но следует следовать этому правилу в целях совместимости с другими СУБД.) Вышеприведенный пример может быть написан иным образом:

```
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:

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

Имена так же могут быть назначены проверкам для таблиц аналогично проверкам для столбцов:

```
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)
);
```

Необходимо отметить, что проверка считается пройденной успешно, если заданное выражение является истинным (возвращающим значение `true`) или имеет неопределенное значение (`NULL`). Таким образом, проверки по умолчанию не предотвращают появление NULL-значений в столбцах таблиц. Для того чтобы в столбцах содержались только определенные значения, необходимо либо явно обеспечить такую проверку, либо использовать не NULL-проверки.

2.3.2. Ограничения NOT NULL

Ограничение NOT NULL показывает, что столбец таблицы, для которого задана такая проверка, не может содержать NULL-значений. Например:

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

Ограничение NOT NULL всегда записывается как ограничение на столбец, который оно ограничивает. По функциональности оно эквивалентно заданию для столбца следующей общей проверки `CHECK (column_name IS NOT NULL)`. Однако в PostgreSQL задание явного NOT NULL ограничения эффективнее, чем задание эквивалентной общей проверки. С другой стороны, для NOT NULL ограничений невозможно явное задание имени, как это можно сделать для общей проверки.

Для столбца может быть задано более одного ограничения. Все ограничения записываются одно за другим:

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

Порядок указания ограничений несущественен и не влияет на порядок реального их выполнения.

Ограничение NOT NULL имеет свою противоположность: ограничение NULL. Это не означает, что колонка должна содержать значение null, которое является бесполезным. Это просто означает, что при выборе значения по умолчанию для данной колонки может использоваться значение null. Ограничение NULL не определено в стандарте SQL и не должно использоваться при написании переносимых приложений. (Оно было добавлено только в PostgreSQL для совместимости с некоторыми другими СУБД). Однако это дает возможность управлять переключением типа ограничений в скрипте. Например можно начать:

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

после чего добавить ключевое слово `NOT`, как было описано ранее.

П р и м е ч а н и е. Во многих БД большинство столбцов должны быть помечены, как не содержащие NULL-значений.

2.3.3. Ограничения уникальности

Ограничения уникальности гарантируют, что данные, содержащиеся в столбце или группе столбцов, являются уникальными во множестве строк данной таблицы. Например:

```
CREATE TABLE products (
  product_no integer UNIQUE,
  name text,
```

```
  price numeric
);
```

для случая, когда это ограничение записывается для столбца, и:

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

когда это ограничение записывается для таблицы.

Если ограничение уникальности относится к группе столбцов, столбцы перечисляются через запятую:

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

Комбинация значений указанных столбцов должна быть уникальна во всей таблице, хотя значения в каждом отдельном столбце необязательно должны быть уникальны.

Для ограничения уникальности, также возможно явное задание имени:

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

Добавление ограничения уникальности автоматически создает уникальный индекс btree на столбце или группе столбцов, используемых в ограничении. Ограничение уникальности только на некоторых строках может быть обеспечено путем создания частичного индекса.

Нарушение проверки на уникальность возникает, когда, по крайней мере, две строки в таблице имеют эквивалентные значения в соответствующих столбцах, в ней задействованных. При этом, NULL-значения всегда считаются неэквивалентными. Таким образом, такое ограничение не предотвращает появление множества записей с эквивалентными значениями уникальных столбцов, если, по крайней мере, один из них содержит NULL-значение. Такое поведение соответствуют SQL-стандарту, однако, некоторые СУБД могут интерпретировать это ограничение иначе. Это следует учитывать при разработке переносимых

приложений.

2.3.4. Первичные ключи

Технически, ограничение первичного ключа является комбинацией ограничений уникальности и NOT NULL. Таким образом, следующие две таблицы будут эквивалентными с точки зрения допустимых данных:

```
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
);
```

Первичный ключ может состоять из нескольких столбцов, подобно ограничению уникальности:

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

Первичный ключ показывает, что столбец или группа столбцов могут быть использованы для однозначной идентификации строк в таблице (проверки на уникальность не могут использоваться в данном качестве, т. к. не исключают NULL-значений). Это удобно и для документирования, и для клиентских приложений. Многие приложения с графическим интерфейсом, позволяющие изменять значения строк, нуждаются в первичном ключе на таблице для возможности однозначно идентифицировать модифицируемую строку.

Добавление первичного ключа автоматически создает уникальный `btree` индекс на столбец или группу столбцов, используемых в качестве первичного ключа.

Таблица может иметь не более одного первичного ключа (тогда как уникальных и не NULL-столбцов может быть множество). Теория реляционных БД определяет, что все таблицы в БД должны иметь первичные ключи, однако PostgreSQL не требует обязательного следования этому правилу.

2.3.5. Внешние ключи

Ограничение внешнего ключа определяет, что столбец (или группа столбцов) может содержать только те значения, которые содержатся в столбце (или группе столбцов) уже существующих строк в другой таблице. Это ограничение обеспечивает *ссылочную целостность* между двумя связанными таблицами. Например, если имеется справочник товаров:

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

и список заказов на эти товары. Требуется обеспечить заказ только тех товаров, которые описаны в справочнике. Для этого определяется ограничение внешнего ключа в таблице заказов, ссылающееся на таблицу товаров:

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

В результате такого определения таблицы заказов в ней невозможно создать строку с не-NULL кодом товара `product_no`, для которого нет соответствующей строки в таблице товаров.

В приведенном примере, таблица `orders` является *ссылающейся (referencing)* таблицей, а таблица `products` является *ссылочной (referenced)* таблицей. Похожим образом, столбцы являются ссылающимися и ссылочными.

В случае, если внешний ключ таблицы ссылается на первичный ключ другой таблицы, можно опустить список столбцов в определении внешнего ключа. Например, таблицу заказов можно было бы описать так:

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

поскольку при отсутствии указания списка столбцов первичный ключ, на который ссылается ограничение, используется как список столбцов, на которые осуществляется ссылка.

Внешний ключ может ссылаться более чем на один столбец другой таблицы. Как и прочие ограничения подобного вида оно записывается как ограничение для таблицы:

```
CREATE TABLE t1 (
```

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

Количество и типы столбцов в определении внешнего ключа должны совпадать.

Для ограничения внешнего ключа возможно явное задание имени.

Таблица может иметь более чем один внешний ключ. Это необходимо, например, для реализации отношения «многие-ко-многим» между сущностями. В предыдущем примере реализации таблиц заказов и товаров каждый заказ мог содержать только один товар. Реализовать более жизненную систему заказов (когда одним заказом можно оформить несколько товаров) можно, например, так:

```
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)
);
```

В последней таблице внешние ключи одновременно являются и первичным ключом таблицы.

В примере выше заказаны могут быть только товары, описанные в справочнике. В случае удаления товара из справочника (на товар остается ссылка в таблице заказов), язык SQL позволяет задать поведение системы. Имеется несколько возможностей:

- запретить удаление такого товара;
- удалить все заказы, ссылающиеся на данный товар;
- выполнить какие-либо другие действия.

В последнем примере: запретить удаление товара, если он используется в списке заказов, пока эти заказы не будут явно удалены; при удалении заказа удалить все соответствующие ему записи заказанных товаров:

```
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)
);
```

Блокировка или распространение действия на связанные таблицы являются наиболее частыми опциями внешних ключей:

- `RESTRICT` — предотвращает удаление строки, на которую существует внешняя ссылка;
- `NO ACTION` — означает, что если существуют строки, ссылающиеся на удаляемую, возникнет ошибка (действие по умолчанию), если явно не задано другое. Отличие от предыдущего варианта заключается в том, что `NO ACTION` поддерживает возможность отложенной до завершения транзакции проверки, `RESTRICT` — нет;
- `CASCADE` — указывает, что при удалении строки, на которую ссылаются другие строки, они так же автоматически должны быть удалены;
- `SET NULL` и `SET DEFAULT` — при удалении строки, на которую ссылаются другие строки, значения ссылающихся столбцов должны быть установлены в NULL или значение по умолчанию, соответственно. Эти значения не являются исключениями из правила проверки внешних ключей. Например, для `SET DEFAULT` значение по умолчанию должно удовлетворять ограничению внешнего ключа, иначе попытка выполнения действия будет признана неудачной.

Аналогично ON DELETE спецификации имеется спецификация ON UPDATE, которая будет задействована при изменении значения соответствующих столбцов в таблице, на которую ссылается внешний ключ. При этом возможны те же действия. В этом случае CASCADE подразумевает копирование измененного значения ссылочных столбцов в строке в столбцы строк, которые на нее ссылаются.

Обычно требуется, чтобы ссылающиеся строки удовлетворяли ограничению внешнего ключа, даже если какой-либо из ссылающихся столбцов содержит NULL-значение. Если к определению ограничения внешнего ключа добавлена опция MATCH FULL, ссылающаяся строка не удовлетворяет ограничению внешнего ключа, только если все ее ссылающиеся столбцы содержат NULL-значения (так что наличие одновременно NULL и не-NULL значений в ее ссылающихся столбцах всегда нарушает ограничение MATCH FULL). В случае необходимости обеспечения удовлетворения ограничения внешнего ключа в любых случаях, следует определять ссылающиеся столбцы с ограничением NOT NULL.

Внешние ключи должны ссылаться либо на первичные ключи других таблиц, либо на столбцы, которые определены как уникальные. Это подразумевает наличие индекса у ссылочного столбца (который лежит в основе первичного ключа или ограничения уникальности). Таким образом обеспечивается эффективность проверки ссылающихся строк.

Поскольку удаление DELETE строки из ссылающейся таблицы или изменении UPDATE ссылающегося столбца требует сканирования ссылочной таблицы на предмет строк, совпадающих со старым значением, рекомендуется индексирование ссылочных столбцов. Поскольку это не всегда необходимо, и существует множество способов индексирования, объявление ограничения внешнего ключа не приводит к автоматическому созданию индекса на ссылочные столбцы.

Подробная информация по изменению и удалению данных приведена в 3. Дополнительная информация по определению ограничений внешнего ключа приведена в описании конструкции CREATE TABLE.

2.3.6. Ограничения исключения

Ограничение исключения гарантирует, что при сравнении любых двух строк по заданным столбцам или выражениям с помощью заданного оператора, по крайней мере одно из этих сравнений оператора возвратит false или null. Ограничение имеет следующий синтаксис:

```
CREATE TABLE circles (
  c circle,
  EXCLUDE USING gist (c WITH &&)
);
```

Подробности приведены в описании конструкции

CREATE TABLE ... CONSTRAINT ... EXCLUDE.

Добавление ограничения исключения автоматически создает индекс типа, указанного при объявлении ограничения.

2.4. Системные столбцы

Каждая таблица имеет ряд системных столбцов, неявно создаваемых системой. Их имена не могут быть использованы для задания пользовательских столбцов. (Следует отметить, что это ограничение отличаются от того, когда имя является или не является ключевым словом; заключение имени в кавычки не позволяет избежать этого ограничения.)

oid

Уникальный идентификатор строки (object ID). Этот столбец присутствует, если таблица создавалась с флагом WITH OIDS или если в момент создания был установлен конфигурационный параметр default_with_oids. Этот столбец имеет тип oid. Информация по этому типу приведена в 5.18.

tableoid

OID таблицы, содержащей данную строку. Этот столбец используется в командах, выбирающих данные из иерархии таблиц, т. к. без него невозможно в дальнейшем определить таблицу — источник строк. Столбец tableoid может быть объединен со столбцом oid системной таблицы pg_class для определения имени таблицы.

xmin

Идентификатор транзакции (transaction ID), в рамках которой была создана данная версия строки. (Версия строки является особенным состоянием строки; каждое обновление строки создает новую версию для той же логической строки.)

cmin

Идентификатор команды (начиная с нуля), создавшей строку в рамках одной транзакции.

xmax

Идентификатор транзакции, удалившей данную строку или нуль для неудаленных версий строк. Возможно ненулевое значение этого столбца для видимых в транзакции строк. Это обычно означает, что транзакция, удалившая строку, еще не завершилась или операция удаления была отменена.

cmax

Идентификатор команды, удалившей эту строку или нуль.

ctid

Указатель физического расположения строки в таблице. Несмотря на то, что ctid может быть использован для быстрого обращения к строкам, этого делать не следует,

поскольку он изменяется каждый раз при обновлении строки или использовании команды `VACUUM FULL`. Для логической идентификации строк следует использовать либо `oid`, либо самостоятельно определенный идентификатор.

`maclabel`

Мандатная метка строки. Необходима для работы мандатной политики разграничения доступа. Является расширением версии СУБД для ОС Astra Linux Special Edition (см.**??**).

Идентификатор OID представляет собой 32-разрядное целочисленное значение, которое получается с единого для всего кластера счетчика. В больших БД возможна ситуация, когда происходит переполнение этого счетчика. Неверно считать значения OID уникальными, если это не обеспечивается специальным образом. Для уникальной идентификации строк в таблице рекомендуется использовать специально выделенные для этого последовательности. Однако OID может быть использован при выполнении следующих условий:

1) должны быть созданы ограничения уникальности на столбцы OID для всех таблиц, в которых предполагается использовать OID для идентификации строк. В случае существования подобных ограничений, система постарается не генерировать OID, совпадающий с уже существующими строками (это возможно для таблиц, содержащих менее 2^{32} (4 миллиарда) строк, лучше когда таблицы содержат меньше строк, иначе может снизиться производительность);

2) OID никогда не должен считаться уникальным во всех таблицах. Если требуется уникальное значение для всей БД возможно использование комбинации `tableoid` и OID;

3) таблицы должны быть созданы с флагом `WITH OIDS` (по умолчанию они создаются с флагом `WITHOUT OIDS`).

Идентификаторы транзакций — 32-разрядные целочисленные значения. В больших БД возможно переполнение счетчика транзакций. Это не является фатальной проблемой и устраняется соответствующими процедурами обслуживания базы данных (см. 13). Однако, неразумно иметь долговременную зависимость от уникальности идентификаторов транзакций (более чем один миллиард транзакций).

Идентификаторы команд — 32-разрядные целочисленные значения. Это дает жесткое ограничение в 2^{32} (4 миллиарда) SQL-команд в рамках одной транзакции (ограничение на количество SQL-команд, а не на количество обработанных строк). Команды, действительно изменившие содержимое БД, получают такой идентификатор.

2.5. Редактирование структуры таблиц

После создания таблицы при обнаружении ошибки или при изменении требований к ней, ее можно удалить и создать заново, однако это не всегда целесообразно. Например, если в таблицу уже занесены данные и (или) на эту таблицу есть ссылки из других таблиц (используются внешние ключи или наследование), то придется удалять и заново создавать все связанные таблицы. PostgreSQL предоставляет набор команд для внесения изменений в существующие таблицы. Важно отметить, что эти действия концептуально отличаются от изменений самих данных, содержащихся в таблице: подразумевается изменение структуры таблицы. С их помощью можно:

- добавлять столбцы;
- удалять столбцы;
- добавлять ограничения;
- удалять ограничения;
- менять значения по умолчанию;
- менять тип столбцов;
- переименовывать столбцы;
- переименовывать таблицы.

Все перечисленные операции выполняются с помощью команды `ALTER TABLE`, на странице описания которой можно найти подробности перечисленных здесь действий.

2.5.1. Добавление столбца

Для добавления столбца используется команда:

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

Добавленный столбец сразу после добавления заполняется значением по умолчанию (NULL, если не было указано значение по умолчанию конструкцией `DEFAULT`).

Одновременно с добавлением столбца могут быть заданы ограничения. Например:

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

Фактически, все опции, которые используются при выполнении команды `CREATE TABLE`, могут быть применены и здесь. Следует учитывать, что значение по умолчанию должно удовлетворять заданным ограничениям, иначе выполнение команды `ADD` закончится ошибкой. Значение по умолчанию может быть добавлено к определению столбца после добавления нового столбца.

П р и м е ч а н и е. Добавление столбца с указанным значением по умолчанию приведет к обновлению всех строк таблицы (для сохранения этого значения). В тоже время, если значение по умолчанию не указано, можно избежать физической перезаписи. Если предполагается заполнить столбец в основном не значением по умолчанию, то устано-

вить с помощью UPDATE желаемое значение, а затем добавить определение значения по умолчанию, как было описано выше.

2.5.2. Удаление столбца

Для удаления столбца используется команда:

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

После этого данные столбца исчезают. Так же удаляются ограничения, включающие указанный столбец. Если на столбец существовали ссылочные ограничения из других таблиц, PostgreSQL не даст его удалить, без указания ключевого слова CASCADE:

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

См. 2.12 где описывается обеспечивающий это механизм.

2.5.3. Добавление ограничения

При добавлении ограничений используется синтаксис ограничений для 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;
```

Для добавления ограничения NOT NULL, которое не может быть записано как ограничение для таблицы, используется команда:

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

При добавлении ограничения, оно сразу же проверяется для всех существующих в таблице строк, поэтому строки таблицы уже должны им удовлетворять.

2.5.4. Удаление ограничения

Для удаления ограничения необходимо знать его имя. Если это имя было задано явно, то используется оно. В противном случае система использует автоматически сгенерированное имя ограничения, которое пользователю придется найти самостоятельно. Для этого может оказаться полезной команда `\d tablename` интерактивного SQL-интерпретатора psql.

После нахождения тем или иным способом имени ограничения для его удаления можно воспользоваться командой:

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

Если осуществляется попытка удалить ограничение со сгенерированным системным именем типа $2, необходимо использовать кавычки для формирования корректного идентификатора.

Так же как и при удалении столбцов, может потребоваться добавление ключевого слова CASCADE при удалении ограничения, имеющего зависимости. Примером может

служить внешний ключ, зависящий от ограничений уникальности или первичного ключа на используемых столбцах.

Эта команда работает для всех ограничений, за исключением ограничений NOT NULL. Для их удаления следует использовать команду:

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

Примечание. NOT NULL ограничения не имеют имен.

2.5.5. Изменение значений по умолчанию

Для установки столбцу нового значения по умолчанию применяется следующая команда:

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

Это никак не повлияет на существующие данные в таблице, изменится только значение по умолчанию для будущих команд `INSERT`.

Удаление значения по умолчанию:

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

Это эквивалентно установке значения по умолчанию в NULL-значение. Выполнение команды удаления значения по умолчанию для столбца, у которого не было этого значения, ошибкой не является, поскольку неявно для него значением по умолчанию является NULL-значение.

2.5.6. Изменение типа столбца

Для конвертирования столбца в другой тип данных используется команда типа:

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

Операция выполнится успешно, если можно будет каждое существующее значение в столбце привести к новому типу с помощью неявного преобразования типов. В более сложных случаях можно с помощью конструкции `USING` определить, как будут вычисляться новые значения из старых.

PostgreSQL сделает попытку привести к указанному типу значение по умолчанию, как и остальные ограничения. Но это может завершиться неудачно или привести к непредсказуемым результатам. Более корректным является удаление всех ограничений на столбец перед изменением его типа. А затем создание их заново.

2.5.7. Переименование столбца

Для переименования столбца используется команда:

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

2.5.8. Переименование таблицы

Для переименования таблицы используется команда:

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

2.6. Права пользователей

При создании объекта БД, ему назначается владелец. Обычно владелец — это та роль, которая запустила оператор создания данного объекта. Для большинства видов объектов, начальное состояние таково, что делать что-либо с объектом может только владелец (или суперпользователь). Чтобы разрешить другим ролям использовать его, им должны быть предоставлены соответствующие права.

Существует несколько различных прав на:

- выборку данных (SELECT);
- вставку данных (INSERT);
- обновление (UPDATE) ;
- удаление (DELETE);
- удаление всех данных из таблицы (TRUNCATE);
- создание ссылок на таблицу (REFERENCES);
- определение для таблицы триггеров (TRIGGER);
- создание постоянных объектов (CREATE);
- установку соединения (CONNECT);
- создание временных объектов (TEMPORARY);
- запуск функций (EXECUTE);
- использование объектов (USAGE).

Права доступа применяются к отдельному объекту, в зависимости от его типа (таблица, функция и т. д.). Для более подробной информации о правах пользователей в PostgreSQL см. описание команды GRANT.

Права на модификацию или удаление объекта всегда являются привилегиями только владельца.

Объекту может быть назначен новый владелец с помощью команды ALTER для соответствующего вида объекта, например, ALTER TABLE. Суперпользователи могут делать это всегда; обычные роли могут делать это только если они одновременно являются текущим владельцем данного объекта (или членом роли текущего владельца) и членом роли нового владельца.

Для предоставления прав используется команда GRANT. Например, для пользователя joe и таблицы accounts право на обновление данных в ней может быть предоставлено с помощью команды:

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

Если вместо списка прав указать специальное право ALL, то будут предоставлены все возможные права в зависимости от типа объекта.

Специальное имя пользователя PUBLIC используется для предоставления прав всем

пользователям системы. Так же «групповые» роли могут оказать помощь в предоставлении прав в случае большого количества пользователей (см. раздел 9).

Для отзыва прав используется команда `REVOKE`, синтаксис которой аналогичен синтаксису команды `GRANT`, а действие полностью противоположно:

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

Специальные права владельца объекта (т. е. права на выполнение команд `DROP`, `GRANT`, `REVOKE` и т. п.) всегда неявно предоставлены владельцу и не могут быть у него отозваны или переданы другому пользователю. Однако владелец объекта может отозвать свои собственные обычные права на использование объекта, например, чтобы перевести объект в режим «только чтение» для себя так же, как и для других пользователей.

Обычно только владелец объекта (или суперпользователь) может предоставлять или отбирать права на доступ к объекту. Однако существует возможность предоставить право «с правом делегирования», что дает получателю привилегию передавать полученное право другим. Если в дальнейшем эта привилегия будет отозвана, все кто получил право от получателя (непосредственно или через цепочку передач прав) потеряют эти привилегии. Подробнее см. описание команд `GRANT` и `REVOKE`.

ВНИМАНИЕ! В реализации СУБД для текущей версии ОС существуют ограничения при использовании ролевого управления доступом (см. **??**).

2.7. Схемы (пространства имен)

Кластер БД PostgreSQL содержит один или более именованных объектов, называемых «базами данных». Информация о пользователях и группах разделяется между всеми БД кластера. Все прочие данные являются частью отдельной БД. Любому отдельному соединению с БД доступны только ее данные.

Примечание. Пользователи кластера не обязательно имеют права доступа к любой БД кластера. Разделение имен пользователей означает, что пользователь не может именоваться в разных БД по-разному; в тоже время система может быть сконфигурирована таким образом, что пользователь получит доступ только к одной.

Каждая БД состоит из одной или более именованных *схем*, которые в свою очередь содержат таблицы. Схемы также содержат и другие виды именованных объектов, включая типы данных, функции и операторы. Одни и те же имена объектов могут использоваться в разных схемах без возникновения конфликтов (поэтому их еще можно назвать «пространствами имен»). Например, и `schema1`, и `myschema` могут содержать таблицу с именем `mytable`. В отличие от БД схемы не являются полностью независимыми контейнерами объектов. Пользователь в рамках одного соединения может одновременно работать с объектами, расположенными в разных схемах одной и той же БД, если он обладает соответствующими правами доступа.

Имеется несколько причин для использования схем:

1) множество пользователей могут использовать одну и ту же БД, не мешая друг другу;

2) с помощью схем можно разбить объекты БД на логические группы, что может упростить их обслуживание (например, контроль доступа к ним пользователей);

3) приложения сторонних разработчиков могут быть помещены в различные схемы, чтобы избежать конфликта имен.

Схемы являются аналогами директорий в ОС за исключением того, что схемы не могут вкладываться одна в другую.

2.7.1. Создание схемы

Для создания новой схемы используется команда `CREATE SCHEMA`. Например:

```
CREATE SCHEMA myschema;
```

Для создания или доступа к объектам в схеме используется полное имя объекта, состоящее из имени схемы и имени объекта, разделенные точкой:

```
schema.table
```

Подобная форма задания имени таблицы допустима в любом месте, где это имя может появиться, включая команды модификации структуры таблицы и команды доступа к данным. (Для краткости здесь и далее говорится только о таблицах, хотя те же самые идеи применимы и к другим видам объектов, таким как типы данных и функции.)

В действительности может быть использовано общее имя, указывающее дополнительно БД, к которой принадлежит объект:

```
database.schema.table
```

Данная форма используется для совместимости со стандартом SQL. Указанная БД должна соответствовать БД, с которой было установлено соединение.

Таким образом, для создания таблицы в новой схеме используется:

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

Для удаления пустой (не содержащей ни одного объекта) схемы используется команда:

```
DROP SCHEMA myschema;
```

Для удаления схемы, содержащей объекты (все объекты при этом так же будут удалены) используется команда:

```
DROP SCHEMA myschema CASCADE;
```

Иногда необходимо создать схему, владельцем которой является другой пользователь (это один из способов ограничить зону активности пользователя):

```
CREATE SCHEMA schemaname AUTHORIZATION username;
```

Если в этой команде пропустить имя схемы, то будет создана схема с именем, идентичным имени пользователя. О применении этого см. 2.7.6.

Имена схем, начинающиеся с символов `pg_` являются зарезервированными для системных целей и не могут быть созданы пользователями.

2.7.2. Схема `public`

Ранее было показано, как создавать таблицы, не задавая для них явно имя схемы. По умолчанию такие таблицы (и прочие объекты) автоматически помещаются в схему с именем `public`. Каждая вновь созданная БД имеет такую схему. Таким образом, следующие команды в этом случае эквивалентны:

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

и

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

2.7.3. Путь поиска схемы

Полные имена объектов не всегда удобно писать, кроме этого, лучше не требовать от приложения всегда указывать схему, в которой расположены его объекты. Таким образом, к объектам часто обращаются с использованием *кратких имен*, которые содержат только имена объектов. Система определяет, какой конкретно объект имеется в виду с помощью *пути поиска*, который представляет собой список схем для просмотра. При этом используется первый же найденный объект с заданным именем. Если во всех схемах, заданных в пути поиска, такой объект не найден, система сообщает об ошибке, даже если в БД существует схема с заданным объектом.

Первая схема в текущем пути поиска называется «схемой по умолчанию». Это первая схема для поиска объектов, и в ней располагаются вновь создаваемые объекты с кратким именем.

Чтобы узнать текущий путь поиска объектов, используется команда:

```
SHOW search_path;
```

В установке по умолчанию такой запрос возвратит:

```
search_path
------
"$user",public
```

Первый элемент определяет, что первой будет просматриваться схема с именем, совпадающим с именем текущего пользователя. Если такой схемы не существует, то этот элемент пути поиска просто игнорируется. Второй элемент указывает на общую схему.

Первая существующая схема в пути поиска является местоположением по умолчанию, для создания новых объектов. Именно по этой причине, по умолчанию объекты создаются в схеме `public`. Когда на объекты ссылаются из любых других контекстов без

указания схемы (модификация таблиц, данных или команды запросов) происходит перебор схем в пути поиска пока не будет найден совпавший объект. Следовательно, в конфигурации по умолчанию, все операции без указания имени схемы используют только схему public.

Чтобы установить другой путь поиска объектов, используется команда:

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

(Здесь схема $user опущена, так как в данный момент не нужна.) Теперь можно обратиться к созданной ранее таблице без явного указания схемы:

```
DROP TABLE mytable;
```

Кроме этого все вновь создаваемые объекты с кратким именем будут создаваться в схеме myschema.

Для исключения из пути поиска общей схемы можно записать:

```
SET search_path TO myschema;
```

Таким образом, единственным отличием общей схемы от схем, создаваемых пользователем, является то, что она создается автоматически при создании новой БД. В частности, так же как и прочие схемы, она может быть удалена.

Другие способы манипулирования путем поиска схем описаны в. 6.25.

Путь поиска работает абсолютно идентично для всех видов объектов БД: таблиц, типов, функций, операторов и т. д. Если необходимо обратиться в выражении к оператору по полному имени, следует использовать синтаксическую форму:

```
OPERATOR(schema.operator)
```

Это необходимо для устранения синтаксической неоднозначности. Например:

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

2.7.4. Схемы и права пользователей

По умолчанию пользователи не могут использовать объекты в схемах, которые им не принадлежат. Чтобы это стало возможным, необходимо дать им это право (USAGE) специально. Для использования пользователями объектов схемы необходимы дополнительные права доступа для каждого объекта отдельно в зависимости от его типа.

Кроме права использования объектов в схеме, пользователю может быть предоставлено право создавать (CREATE) в ней новые объекты. По умолчанию все пользователи имеют право создавать и использовать объекты в схеме public. Это право может быть отозвано на общих основаниях:

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

Первое слово public относится к имени схемы, второе PUBLIC означает любой пользователь. В первом случае — это идентификатор, во втором — ключевое слово.

2.7.5. Схема системного каталога

Кроме общей схемы и схем, создаваемых пользователями, каждая БД содержит схему системного каталога (`pg_catalog`), которая содержит системные таблицы и все встроенные типы данных, функции и операторы. Схема `pg_catalog` всегда является частью пути поиска вне зависимости от того, указана он там явно или нет. Если схема `pg_catalog` не указана явно, то при поиске объектов она просматривается самой первой. Это сделано для того, чтобы все встроенные возможности БД всегда были доступны. Однако возможно явно поместить схему `pg_catalog` в конец пути поиска, если необходимо, чтобы пользовательские объекты могли скрывать встроенные.

Все системные таблицы и представления начинаются с сочетания символов `pg_`. PostgreSQL не требует от пользователей избегать подобных имен при именовании собственных объектов, однако это рекомендуется во избежание конфликтов имен, если в будущем, в какой-либо версии появится системная таблица с таким же именем. (С путем поиска по умолчанию, неполное имя такой таблицы будет распознано как имя системной таблицы.)

2.7.6. Способы использования

Схемы могут быть использованы для организации данных различными способами. Существует несколько рекомендованных способов использования, которые легко обеспечиваются стандартной конфигурацией:

1) если не создано ни одной схемы, все пользователи неявно получают доступ к общей схеме `public`. Это имитирует ситуацию, когда схемы не применяются. Такая конфигурация рекомендуется при одном или малом количестве пользователей БД. Так же это обеспечивает легкий переход от систем, где не используется подход на основе схем;

2) возможно создание схемы для каждого пользователя с именем, совпадающим с именем пользователя. Путь поиска по умолчанию начинается с `$user`, которое разыменовывается в имя пользователя. Следовательно, если каждый пользователь имеет собственную схему, он по умолчанию получает к ней доступ. Если у пользователей отобраны права доступа к общей схеме `public`, или она удалена, пользователи становятся полностью ограничены в своих схемах;

3) при установке совместно используемых приложений (таблиц, которые используют все, дополнительные функции от сторонних разработчиков и т. п.) можно размещать их в отдельных схемах. При этом необходимо помнить о соответствующих правах доступа к ним пользователей. Пользователи могут ссылаться на эти дополнительные объекты путем указания полных имен, включая схему или добавляя схемы в путь поиска.

2.7.7. Переносимость

В SQL-стандарте не предусмотрена возможность создания в одной схеме объектов, принадлежащих разным пользователям. Более того, некоторые реализации не допускают создания схем с именами, отличными от имен их владельцев. Фактически в СУБД, которые обеспечивают только базовую поддержку схем, концепции схем и пользователей практически эквивалентны. Таким образом, пользователи таких БД под полными именами объектов часто подразумевают `username.tablename`. PostgreSQL позволяет легко эмулировать такое поведение.

SQL-стандарт также не имеет концепции общей схемы (`public`). Для максимального соответствия стандарту, не следует использовать (возможно потребуется даже удалить) схему `public`.

Многие СУБД могут вообще не поддерживать схемы или предоставлять (возможно, ограниченную) возможность перекрестного доступа к объектам разных БД. Максимальная совместимость с подобными системами может быть достигнута при полном отказе от использования схем.

2.8. Наследование

PostgreSQL реализует наследование таблиц, которое может оказаться полезным инструментом для разработчиков базы данных. (Возможность наследования определяется стандартом SQL:1999 и более поздними стандартами и во многих отношениях отличается от возможностей, описываемых здесь.)

Наследование таблиц можно рассмотреть на примере. Допустим, создается модель городов. Каждый штат содержит множество городов, но только один из них является столицей. Требуется быстрый поиск столицы штата. Создается две таблицы: одна для столиц, другая для остальных городов. Также может потребоваться получить информацию о городе независимо от того, столица он или нет. Наследование может помочь решить эту задачу. Таблица `capitals` определяется как наследующая от таблицы `cities`:

```
CREATE TABLE cities (
  name text,
  population float,
  altitude int -- in feet
);

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

В данном случае таблица `capitals` наследует все столбцы, заданные в «родитель-

ской» таблице cities. В таблице столиц также есть дополнительный столбец с аббревиатурой названия штата — state.

В PostgreSQL одна таблица может наследовать столбцы из нуля и более таблиц, и запрос может возвращать либо строки самой таблицы, либо строки всех таблиц, находящихся ниже по иерархии. Последний вариант используется по умолчанию. Например, следующая команда выведет имена всех городов, включая столицы, которые находятся на высоте более 500 метров выше уровня моря:

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

С другой стороны, следующая команда выведет все города, которые не являются столицами и расположены на высоте более 500 метров выше уровня моря:

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

Здесь ключевое слово ONLY перед именем таблицы показывает, что команда относится только к строкам этой таблицы и не включает строки таблиц, находящихся ниже по иерархии. Многие из команд (SELECT, UPDATE и DELETE) поддерживают использование этого ключевого слова.

Так же возможно добавить символ * после имени таблицы для явного указания включения строк всех таблиц, находящихся ниже по иерархии.

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

Указание * не является необходимым, т. к. это поведение используется по умолчанию (если это не изменено с помощью конфигурационного параметра sql_inheritance). В то же время указание * может быть удобно для более наглядного выделения этого варианта запроса.

В ряде случаев необходимо знать таблицу, к которой принадлежит конкретная строка. Для этого предназначен системный атрибут tableoid (имеется у каждой таблицы):

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

Чтобы увидеть наименования таблиц вместо числовых значений, можно объединить результат этой выборки с системной таблицей pg_class:

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

Наследование не обеспечивает автоматическую вставку данных командами INSERT и COPY в другие таблицы по иерархии наследования. Для приведенного примера следующая команда INSERT выполнена не будет:

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

INSERT всегда вставляет данные в указанную в запросе таблицу. В некоторых случаях возможно реализовать перенаправление вставки данных с помощью правила. Однако это не поможет в этом случае, т. к. таблица cities не содержит столбец state, и команда будет отклонена еще до применения правила.

Все ограничения проверки и NOT NULL-проверки родительской таблицы автоматически наследуются ее потомками. Другие типы ограничений (уникальности, первичные и вторичные ключи, индексы) не наследуются.

Таблица может наследоваться более чем от одной родительской таблицы, в этом случае она получает объединенный набор столбцов родительских таблиц. К нему добавляются столбцы, определенные для самой дочерней таблицы. Если встречаются столбцы с одинаковым наименованием у родительской таблицы или у родительской и дочерней, они соединяются в один, так что в дочерней таблице оказывается один столбец с таким именем. Для соединения столбцы должны иметь одинаковый тип данных, иначе возникнет ошибка. Соединенный столбец получает все ограничения проверки, определенные для него в каждой из таблиц, и помечается как не содержащий NULL-значений, если это указано хотя бы в одной таблице.

Наследование таблиц обычно устанавливается в момент создания дочерней таблицы использованием конструкции INHERITS в операторе CREATE TABLE. Кроме того, таблице, уже определенной совместимым способом, можно установить новую иерархическую зависимость INHERIT вариантом ALTER TABLE. Для этого таблица уже должна включать в себя столбцы того же типа и наименования, как в родительской, и ограничения проверки с теми же именами и правилами расчета. Аналогично связь наследования может быть удалена использованием NO INHERIT варианта ALTER TABLE. Динамическая установка и удаление связей наследования могут быть полезны в случае использования наследования для горизонтального разбиения таблиц (2.9).

Одним из способов создания совместимых таблиц, которые в последствие могут быть объявлены дочерними, является использование конструкции LIKE в CREATE TABLE. Это позволяет создать таблицу с таким же набором столбцов, как у исходной. Если в исходной таблице присутствуют ограничения CHECK, может быть использована опция INCLUDING CONSTRAINTS конструкции LIKE, так что новая таблица получит все ограничения проверки, совпадающие с исходной таблицей, что даст ей возможность считаться

совместимой.

Родительская таблица не может быть удалена пока существуют дочерние. Так же столбцы и CHECK-ограничения дочерних таблиц не могут быть удалены или изменены, если они унаследованы от любой родительской таблицы. Единственным способом удалить таблицу и всех ее наследников является использование опции `CASCADE`.

`ALTER TABLE` обеспечивает распространение любых изменений столбцов и CHECK-ограничений вниз по иерархии наследования. Вместе с тем, удаление столбца или ограничение родительской таблицы возможно только с опцией `CASCADE`. `ALTER TABLE` следует тем же правилам соединения столбцов или отклонения операций, что и `CREATE TABLE`.

Относительно применения прав доступа принято следующее: получение данных из родительской таблицы автоматически предоставляет доступ к дочерним таблицам без дополнительной проверки прав доступа. Это предотвращает появление в родительской таблице данных, дублирующих данные в дочерних. В то же время при прямом доступе к дочерним таблицам, доступ предоставляется только при наличии соответствующих прав доступа.

2.8.1. Предостережения

Следует отметить, что не все команды SQL могут работать с иерархиями наследования. Команды, которые используются для запросов данных, изменения данных или изменения схем (например, `SELECT`, `UPDATE`, `DELETE`, большинство вариантов `ALTER TABLE`, но не `INSERT` или `ALTER TABLE ... RENAME`) обычно по умолчанию включают дочерние таблицы и поддерживают нотацию `ONLY` для их исключения. Команды, которые обслуживают базу данных и выполняют тонкие настройки (например, `REINDEX`, `VACUUM`) обычно работают только с отдельными, физическими таблицами и не поддерживают рекурсивную обработку иерархий наследования. Конкретное поведение каждой отдельной команды приведено в ее описании.

Серьезное ограничение наследования заключается в том, что индексы (включая ограничения уникальности) и ссылочные ограничения целостности применяются только к самой таблице, а не ко всем ее дочерним таблицам. Это одинаково верно в отношении обоих сторон ссылочного ограничения целостности, как к ссылающейся, так и к той, на которую ссылаются. В приложении к рассмотренному примеру это означает:

1) если на `cities.name` установлено ограничение уникальности или первичный ключ, это не предотвратит в таблице `capitals` появления строки с таким же именем города. В действительности по умолчанию таблица `capitals` не имеет ограничений уникальности, но даже при создании подобных ограничений это не предотвратит дублирование по отношению к содержимому таблицы `cities`;
2) если определить, что `cities.name` ссылается (`REFERENCES`) на другую таблицу,

это ограничение не будет автоматически распространено на `capitals`. В этом случае следует добавить ограничение в таблицу `capitals`;

3) если в какой-либо таблице определено ограничение `REFERENCES cities(name)`, то такая таблица сможет содержать только имена городов, но не столиц.

2.9. Горизонтальное разбиение таблиц

PostgreSQL поддерживает базовое разбиение таблиц. Данный раздел описывает зачем и как реализовать разбиение как часть проектирования базы данных.

2.9.1. Общие сведения

Горизонтальное разбиение относится к разбиению, когда логически одна большая таблица разбивается на несколько более малых частей. Разбиение может обеспечить следующие преимущества:

1) скорость выполнения запросов может быть значительно увеличена в некоторых ситуациях (когда большинство часто запрашиваемых строк таблицы расположены в одной части или малом количестве частей). Разбиение выступает заменой ведущего столбца индексов, уменьшая размер индекса и делающий его более применимым, т. к. наиболее часто используемая его часть полностью вмещается в оперативной памяти;

2) в случаях выборки или изменения большого (в процентном отношении) количества данных в одной части производительность может быть увеличена за счет применения последовательного сканирования вместо использования индекса или случайного чтения, разбросанного по всей таблице;

3) массовые загрузки или удаления могут быть совершены добавлением или удалением частей, если подобное было предусмотрено при проектировании разбиения. `ALTER TABLE NO INHERIT` и последующее `DROP TABLE` выполняется значительно быстрее, чем массовые операции. Так же это позволяет избежать затрат на выполнение операции `VACUUM`, необходимой после массового удаления (`DELETE`);

4) редко используемые данные (архивные) могут быть помещены на дешевое и медленное устройство хранения.

Преимущества обычно достигаются, только когда разбиваемая таблица действительно очень большая. Точное определение, разбиение какой таблицы может дать выигрыш, зависит от конкретного приложения, хотя основным правилом является превышение размера таблицы объема физической памяти сервера БД.

PostgreSQL поддерживает горизонтальное разбиение механизмом наследования (см. 2.8). Каждое разбиение должно быть создано как дочерняя таблица одной родительской

таблицы. Родительская таблица сама обычно пуста; она существует только для предоставления самого набора данных.

В PostgreSQL могут быть реализованы следующие формы разбиения таблиц:

- по диапазонам — таблица разбивается на диапазоны, определяемые ключевым столбцом или набором столбцов, при этом данные в разных частях не пересекаются. Например, диапазон дат или диапазон типовых бизнес-объектов;
- списочное — таблица разбивается путем явного указания списка ключевых значений для каждой части.

2.9.2. Реализация разбиения

Для того чтобы определить разбиение, необходимо выполнить следующее:

1) создать родительскую (главную) таблицу, от которой будут наследоваться все части разбиения. Эта таблица не содержит данных. На ней не должно быть задано никаких ограничений проверки, если не планируется их применение одинаково для всех частей. Так же не имеет смысла определять на ней ограничения уникальности или индексы;

2) создать некоторое количество дочерних таблиц, каждая из которых должна быть унаследована от родительской. Обычно при создании подобных таблиц в них не добавляют новых столбцов к тем, что унаследованы от родительской. Такие дочерние таблицы считаются частями разбиения, хотя они представляют собой обычные таблицы PostgreSQL;

3) добавить в таблицы ограничения проверки, которые определяют допустимые значения ключевых полей для каждой таблицы. Типичным примером может быть:

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

Необходимо, что бы проверки гарантировали отсутствие пересечений по значениям ключевых полей между разными частями разбиения. Общая ошибка определения диапазонов следующая:

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

4) для каждой части создать индекс по ключевым полям, так же как и другие запланированные индексы. (Индекс по ключу разбиения не обязателен, но во многих ситуациях может быть полезен. Если планируется обеспечить уникальность ключа разбиения, необходимо создать ограничение уникальности или первичный ключ по ключу разбиения для каждой части.)

5) при необходимости определить триггер или правило, перенаправляющее добавляемые данные в родительскую таблицу в соответствующую часть;

6) убедиться, что конфигурационный параметр `constraint_exclusion` установлен в `postgresql.conf`. В противном случае запросы не будут оптимизированы должным образом.

Например, рассмотрим задачу разработать БД для компании, производящей мороженое. Компания измеряет максимальную температуру так же, как и продажи мороженого в каждом регионе. В общем, необходимо иметь следующую таблицу:

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

Известно, что большинство запросов затрагивают данные за последнюю неделю, месяц или квартал, причем таблица в основном используется для подготовки управленческих отчетов. Для уменьшения количества хранящихся данных, решено хранить данные только за последние 3 года. В начале каждого месяца данные за самый давний месяц удаляются.

В данной ситуации можно использовать горизонтальное разбиение:

1) родительская таблица `measurement` уже объявлена;

2) создаются таблицы для каждого месяца:

```
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);
```

Каждая часть представляет собой полноценную таблицу с собственными правами доступа, но наследует определение родительской, что решает одну из задач: удаление данных. Единственное, что требуется каждый месяц, это выполнить операцию `DROP TABLE` над самой давней дочерней таблицей и создать новую для данных нового месяца;

3) определить непересекающиеся ограничения на таблицы:

```
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) определить индексы на ключевых полях:

```
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) создать соответствующий триггер к родительской таблице `measurement`, чтобы операцией `INSERT INTO` данные помещались в соответствующую таблицу. Если данные должны помещаться только в последнюю часть, можно использовать следующую простую триггерную функцию:

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

```
END;
$$
LANGUAGE plpgsql;
```

После создания триггерной функции требуется создаеть и сам триггер, ее вызывающий:

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

Необходимо переопределять триггерную функцию каждый месяц, чтобы она всегда действовала на текущую часть. В тоже время сам триггер не требует переопределения.

При желании автоматического определения сервером части, в которую должны быть добавлены данные, можно использовать более сложную триггерную функцию, например:

```
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;
```

Само определение триггера остается без изменений. Каждое условие в проверках `IF` должно совпадать с соответствующим проверочным ограничением для указанной части.

Поскольку эта функция более сложная, чем для одного месяца, она не требует обновления каждый месяц до тех пор, пока не потребуется добавление новых ветвей условий.

Примечание. Лучше начинать проверку в более новой части, если большинство операций добавления осуществляется в нее.

Для полноценного горизонтального разбиения таблиц требуется некоторое количество кода (DDL). В приведенном примере было необходимо создавать новую часть каждый месяц, хотя более разумно — создать скрипт, который генерировал бы требуемый код (DDL) автоматически.

2.9.3. Управление разбиением

Обычно набор частей разбиения, установленный при начальном определении таблиц, не планируется статичным. В общем случае требуется удалять старые части и периодически добавлять новые для новых данных. Одним из наиболее важных преимуществ разбиения является то, что оно обеспечивает возможность выполнения операций, затрагивающих большое количество данных, мгновенным изменением структуры разбиения вместо их физического перемещения.

Наиболее простым способом удаления старых данных является удаление части, которая больше не нужна:

```
DROP TABLE measurement_y2006m02;
```

Это выполняется значительно быстрее, чем удаление миллионов записей из таблицы, потому что при этом не выполняется отдельное удаление каждой записи.

Другой способ — удаление части из разбиваемой таблицы, оставляя доступ к данным со своими правами доступа:

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

Это позволяет в дальнейшем выполнять операции над данными до их удаления. Например, чтобы сделать резервную копию данных с использованием `COPY`, `pg_dump` или подобными средствами. Так же это может быть удобным для агрегирования данных в более компактный формат, дополнительной обработки или для создания отчетов.

Похожим образом возможно добавление новой части для поддержки новых данных. Можно создать пустую часть к разбиваемой таблице:

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

Иногда более подходящим является создание новой таблицы отдельно, вне структуры разбиения, и объявление ее как части позднее. Это позволяет данным быть загруженными, проверенными и подготовленными, прежде чем они появятся в разбиваемой

таблице:

```
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'
-- возможно, какие-то еще действия по подготовке данных
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

2.9.4. Разбиения и исключение по ограничениям целостности

Исключение по ограничениям целостности (constraint exclusion) является технологией оптимизации запросов, которая улучшает производительность для разбитых таблиц. Например:

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

Без исключения по ограничениям целостности, данный выше запрос будет сканировать каждое разбиение таблицы `measurement`. При разрешении использовать исключение по ограничениям целостности планировщик запросов исследует каждое ограничение в каждой части разбиения и попытается убедиться, что часть не должна сканироваться, т. к. не содержит строк, удовлетворяющих условиям, приведенным в конструкции `WHERE` запроса. Если планировщик может доказать это, он исключает эту часть из плана запроса.

Можно использовать команду `EXPLAIN` для просмотра разницы между планом запроса с включенным конфигурационным параметром `constraint_exclusion` и планом с выключенным. Обычным планом для подобной таблицы является:

```
SET constraint_exclusion = off;
EXPLAIN SELECT count(*) FROM measurement
  WHERE logdate >= DATE '2008-01-01';
QUERY PLAN
------------------------------------------------------------------------
Aggregate (cost=158.66..158.68 rows=1 width=0)
-> Append (cost=0.00..151.88 rows=2715 width=0)
    -> Seq Scan on measurement (cost=0.00..30.38 rows=543 width=0)
       Filter: (logdate >= '2008-01-01'::date)
    -> Seq Scan on measurement_y2006m02 measurement (cost=0.00..30.38
                                                     rows=543 width=0)
       Filter: (logdate >= '2008-01-01'::date)
    -> Seq Scan on measurement_y2006m03 measurement (cost=0.00..30.38
                                                     rows=543 width=0)
```

```
      Filter: (logdate >= '2008-01-01'::date)
  ...
   -> Seq Scan on measurement_y2007m12 measurement (cost=0.00..30.38
                                                    rows=543 width=0)
      Filter: (logdate >= '2008-01-01'::date)
   -> Seq Scan on measurement_y2008m01 measurement (cost=0.00..30.38
                                                    rows=543 width=0)
      Filter: (logdate >= '2008-01-01'::date)
```

Некоторые или даже все части могут потребовать индексированного чтения вместо последовательного просмотра всей таблицы, но для приведенного запроса нет необходимости просматривать части, содержащие старые данные. При включении `constraint_exclusion` план выполнения указанного запроса сокращается:

```
SET constraint_exclusion = on;
EXPLAIN SELECT count(*) FROM measurement
  WHERE logdate >= DATE '2008-01-01';
QUERY PLAN
------------------------------------------------------------------------
Aggregate (cost=63.47..63.48 rows=1 width=0)
-> Append (cost=0.00..60.75 rows=1086 width=0)
    ->Seq Scan on measurement (cost=0.00..30.38 rows=543 width=0)
      Filter: (logdate >= '2008-01-01'::date)
    ->Seq Scan on measurement_y2008m01 measurement (cost=0.00..30.38
                                                    rows=543 width=0)
      Filter: (logdate >= '2008-01-01'::date)
```

Исключение по ограничениям целостности поддерживает только CHECK-ограничения и не зависит от наличия индексов. Следовательно, нет необходимости определять индекс на ключевых полях. Необходимо ли создавать индекс для данного раздела или нет, зависит от того ожидается ли, что запросы которые сканируют разбиение будут обычно сканировать большую его часть, или только маленькую часть. Индекс будет полезен в последнем случае, но не в первом.

По умолчанию (и рекомендуемой) является настройка `constraint_exclusion` которая имеет значение отличное и от `on`, и от `off`; промежуточная настройка называется `partition` и заставляет данную технологию включаться только для запросов, которые работают с таблицами-разбиениями. Настройка `on` заставляет планировщик проверяет CHECK-ограничения во всех запросах, даже простых, что не дает преимуществ.

2.9.5. Альтернативные методы разбиения

Другим подходом обеспечения перенаправления вставки данных в соответствующую часть является задания правила на родительскую таблицу вместо создания триггера. 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.*);
```

Правило требует намного больше затрат, чем триггер, но эти затраты требуются один раз на запрос, а не на каждую строку. Таким образом, этот метод может быть более выгодным для массовых операций. Но в большинстве случаев, триггер более производителен.

Команда `COPY` игнорирует правила. При ее использовании данные помещаются строго в указанную часть. С другой стороны, команда `COPY` вызывает нормальное срабатывание триггеров, так что возможность вставки через родительскую таблицу в этом случае сохраняется.

Другой недостаток использования правила заключается в том, что невозможно корректно обработать ошибку недостаточности покрытия диапазонов, в этом случае данные просто добавятся в родительскую таблицу.

Разбиение может быть реализовано и с помощью выражения `UNION ALL` вместо наследования. Например:

```
CREATE VIEW measurement AS
           SELECT * FROM measurement_y2006m02
UNION ALL SELECT * FROM measurement_y2006m03
...
UNION ALL SELECT * FROM measurement_y2007m11
UNION ALL SELECT * FROM measurement_y2007m12
UNION ALL SELECT * FROM measurement_y2008m01;
```

Однако это потребует пересоздания представления каждый раз при добавлении или удалении части разбиения. На практике рекомендуется вместо этого использовать наследование (см. 2.8).

2.9.6. Предостережения

Предостережения при использовании разбиения:

1) не существует возможности автоматически проверить, что все CHECK-ограничения взаимно исключающие. Более безопасно запрограммировать создание и удаление частей, чем создавать их руками;

2) представленная схема создания разбиения подразумевает, что ключ, по которому осуществляется разбиение, не изменяется, или, по крайней мере, это изменение не требует перенесения строки между частями. В противном случае попытка выполнить `UPDATE` приведет к ошибке при выполнении проверок. При необходимости обеспечить выполнение подобных операций можно использовать соответствующим образом написанные триггеры, но это усложнит управление структурой разбиения;

3) при выполнении команд `VACUUM` и `ANALYZE` вручную следует помнить, что их необходимо выполнить для каждой части отдельно. Команда типа:

```
ANALYZE measurement;
```

будет выполнена только для родительской таблицы.

Следующие предостережения касаются исключения по ограничениям:

1) исключения по ограничениям действуют только тогда, когда условная часть запроса `WHERE` содержит константы. Параметризованные запросы не оптимизируются, т. к. планировщик не может предсказать, в какую часть попадет условие во время исполнения. По этой причине следует избегать *изменяемых* (non-immutable) функций типа `CURRENT_DATE`;

2) использовать простые проверки разбиения, иначе планировщик не сможет доказать, что часть не должна просматриваться. Должны использоваться простые операторы сравнения или простые проверки диапазонов. Хорошим правилом является использование в условиях сравнения только ключевых полей разбиения с константами, используя B-tree операторы индексирования;

3) все ограничения во всех частях просматриваются при исключениях по ограничениям, так что большое количество разбиений соответственно увеличивает и план запроса. Разбиение мастер-таблицы, использующее эту технологию будет хорошо работать предположительно с более чем сотней разбиений, но не стоит пытаться использовать тысячи разбиений.

2.10. Внешние данные

PostgreSQL реализует часть спецификации SQL/MED, которые позволяют, используя обычные SQL запросы, получить доступ к данным, которые находятся за пределами PostgreSQL. Такие данные называются *внешними данными*. (Не следует путать это название с внешними ключами, которые являются одним из видов ограничений целостности внутри

СУБД.)

Доступ к внешним данным осуществляется при помощи *обработчика внешних данных* (`foreign data wrapper`). Обработчик внешних данных — это некая библиотека, которая может устанавливать соединение с внешними источниками данных, скрывая подробности подключения к источнику данных и извлечения данных из него. В модуле `contrib` есть обработчик внешних данных, который может читать обычные файлы с данными, расположенными на сервере. Другие виды внешних обработчиков данных можно найти как отдельные продукты. Если не существует подходящего обработчика внешних данных, он может быть разработан отдельно.

Чтобы получить доступ к внешним данным, необходимо создать объект *внешний сервер* (`foreign server`), который определяет как подключаться к конкретному источнику данных в соответствии со списком опций, используемых конкретным обработчиком внешних данных. Затем требуется создать одну или более *внешних таблиц* (`foreign tables`), которые определяют структуру внешних данных. Внешняя таблица может быть использована в запросах как самая обычная таблица, но она не хранится сервером PostgreSQL. Таким образом, когда она используется, PostgreSQL запрашивает обработчик внешних данных, чтобы получить данные из внешнего источника.

Доступ к внешним данным может требовать авторизации на внешнем источнике данных. Эта информация может быть предоставлен с помощью механизма `user mapping`, который может предоставлять дополнительные опции, основанные на текущей роли PostgreSQL.

Дополнительная информация приведена в описании команд `CREATE FOREIGN DATA WRAPPER`, `CREATE SERVER`, `CREATE USER MAPPING` и `CREATE FOREIGN TABLE`.

Примечание. До версии PostgreSQL 9.6, внешние таблицы доступны только для чтения.

2.11. Другие объекты баз данных

Таблицы, рассмотренные ранее, являются центральным понятием любой реляционной БД, поскольку обеспечивают собственно хранение данных. Кроме них существует множество дополнительных объектов, которые позволяют обеспечить гибкое управление данными. К таким объектам относятся:

- виды (представления);
- функции, операторы;
- типы данных, домены;
- триггеры и правила модификации запросов.

2.12. Зависимости между объектами базы данных

При создании сложной структуры БД неявно создается сеть зависимостей между различными ее объектами: множеством таблиц, видов, триггеров, функций и т. д. Например, таблица с внешним ключом зависит от таблицы, на которую она ссылается.

Для гарантии целостности структуры БД PostgreSQL автоматически отслеживает зависимости между объектами и не позволяет удалить какой-либо объект, если существует другой объект, зависящий от него. Попытка такого удаления приведет к появлению сообщения об ошибке вида:

```
DROP TABLE products;

NOTICE:  constraint orders_product_no_fkey on table orders depends on /
table products
ERROR:  cannot drop table products because other objects depend on it
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

что означает:

```
DROP TABLE products;

ЗАМЕЧАНИЕ:  ограничение orders_product_no_fkey для таблицы orders зависит /
от таблицы products
ОШИБКА:  не могу удалить таблицу products, потому что другие объекты зависят /
от нее
СОВЕТ:  Используйте DROP ... CASCADE для удаления и зависящего объекта.
```

Тем не менее, существует возможность удалить такой объект с одновременным удалением всех объектов, зависящих от него с помощью флага `CASCADE` команды удаления:

```
DROP TABLE products CASCADE;
```

при этом все зависящие от `products` объекты будут удалены. В данном случае, команда не удалит таблицу `orders`, она только удалит ограничение внешнего ключа.

Все команды удаления PostgreSQL поддерживают это ключевое слово. Вместо флага `CASCADE` можно указать флаг `RESTRICT`, который соответствует поведению команды удаления по умолчанию.

Примечание. SQL-стандарт требует всегда указывать `CASCADE` или `RESCTRICT` как часть команды удаления, однако ни одна СУБД не следует этой рекомендации. При этом соответствие поведению по умолчанию тому или иному сценарию зависит от конкретной системы.

3. РАБОТА С ДАННЫМИ

Ранее было описано создание таблиц и других структур хранения данных. В данной главе приводится описание процедур вставки, модификации и удаления данных. Операции по извлечению данных из БД приведены в следующей главе.

3.1. Вставка данных

Сразу после создания таблицы, она не содержит никаких данных, и первой задачей перед использованием БД является наполнение ее данными. Данные вставляются по одной строке за раз. Существует возможность вставлять данные по несколько строк, но не существует способа вставить меньше одной строки. Даже если известно только несколько значений полей строки, она создается целиком.

Для создания новой строки используется команда `INSERT`. Эта команда требует имя таблицы и значение для каждого столбца таблицы. Например, рассмотрим таблицу `products` из раздела 2:

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

Для вставки строки можно использовать, например, такую команду:

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

Значения данных перечисляются в том порядке, в котором следуют столбцы в таблице и разделяются запятыми. Обычно значения данных являются литералами (константами), но возможны и скалярные выражения.

Показанный выше синтаксис не всегда удобен, поскольку требует знания порядка столбцов в таблице. Так же возможно использование задаваемого явно списка столбцов. Например, обе следующих команды в итоге сработают точно также, как и рассмотренная ранее:

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

Если не известны все значения для строки данных, можно опустить некоторые из них. При этом соответствующий столбец будет заполнен значением по умолчанию:

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

Вторая форма является расширением PostgreSQL. При этом заполнение столб-

цов идет слева, согласно указанным значениям, а оставшиеся столбцы будут заполнены значениями по умолчанию.

Использование значения по умолчанию может быть явно указано как для конкретного столбца, так и для всей строки:

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

Так же допустима вставка нескольких строк одной командой:

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

Примечание. Для единовременной вставки большого количества данных применяется команда `COPY`. Она не столь гибкая как `INSERT`, но более эффективная с точки зрения производительности.

3.2. Модификация данных

Модификация данных, которые уже находятся в БД называется «обновлением». Возможно обновление отдельной строки, всех строк в таблице или их подмножества. Каждый столбец в строке может быть обновлен отдельно, при этом значения других изменены не будут.

Для выполнения обновления необходимо знать:

1) имя таблицы и столбца для обновления;
2) новое значение столбца;
3) информацию о том, какую строку (и) необходимо обновить.

Несмотря на то, что SQL не поддерживает такую возможность, обычно (согласно 2.4), для каждой строки предоставляется уникальный идентификатор. Поскольку не всегда возможно прямое указание строки для обновления, предусмотрена возможность задания условий, согласно которым будут выбираться строки для обновления. Только при наличии в таблице первичного ключа существует возможность надежно ссылаться на отдельные строки, путем создания условия, которое использует первичный ключ. Графические средства работы с БД используют это для обеспечения обновления отдельных строк.

Например, следующая команда обновляет все продукты, у которых значение поля цены (`price`) равно 5, на новое значение `price`, которое равно 10:

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

Может случиться, что в результате работы этой команды будут обновлены нуль, одна или несколько строк. Это не является ошибкой, т. к. команда может не найти ни одной

строки, совпадающей с условием.

Команда начинается с ключевого слова UPDATE, за которым следует имя таблицы. Имя таблицы может быть указано с названием схемы, в противном случае оно ищется в текущем пути. Затем идет ключевое слово SET, за которым следует имя столбца, знак равно и новое значение для этого столбца. Новое значение столбца может быть любым скалярным выражением, а не только константой. Например, для увеличения цен на все продукты на 10%, может быть использована команда:

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

Выражение для нового значения может также ссылаться на существующее значение в строке. Если опускается конструкция WHERE, будут обновлены все строки в таблице. Если она указана, то обновлены будут только те строки, которые удовлетворяют условию, заданному в WHERE. Знак = в предложении SET используется для присваивания, в то время как в предложении WHERE — для сравнения, но такое поведение не создает неоднозначностей. Условие WHERE необязательно должно быть сравнением. Доступны и многие другие операторы (см. раздел 6). Необходимо только, чтобы выражение возвращало логический результат типа boolean.

Обновление более чем одного столбца в команде UPDATE выполняется путем перечисления более чем одного присваивания в предложении SET. Например:

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

3.3. Удаление данных

Также как и в случае добавления данных, удалять из таблицы можно только всю строку. Поскольку SQL не поддерживает прямое указание строки для выполнения операций, удаление строк может быть выполнено только через задание условий, при соблюдении которых они будут удалены. В случае наличия у таблицы первичного ключа имеется возможность четкой идентификации удаляемых строк путем выбора соответствующих условий. Так же возможно удаление строк, удовлетворяющих условиям, или всех строк таблицы за раз.

Для удаления строк используется команда DELETE, синтаксис которой похож на команду UPDATE. Например, удаление всех строк из таблицы products, в которых значение поля price равно 10, выполняется следующей командой:

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

При выполнении команды:

```
DELETE FROM products;
```

в таблице будут удалены все строки!

4. ЗАПРОСЫ

В разделе приведены основные сведения о способах получения данных из БД.

4.1. Общие сведения

Процесс извлечения или команда извлечения данных из БД называется запросом. В SQL для выполнения запроса используется команда SELECT. В самой общей форме ее синтаксис выглядит так:

```
[WITH with_queries] SELECT select_list
  FROM table_expression [sort_specification]
```

В следующих подразделах подробно описывается список выборки, табличное выражение и спецификация сортировки. Запросы с WITH обсуждаются в конце, так как они являются дополнительной возможностью SQL.

Простейшим примером этой команды является:

```
SELECT * FROM table1;
```

Если существует таблица table1, эта команда выведет данные всех ее строк и столбцов. Конкретная форма вывода, при этом, будет зависеть от используемого клиентского приложения. Определение списка выборки select_list, как * означает, что будут выведены все столбцы заданного табличного выражения table_expression, которое может использоваться для выбора подмножества доступных столбцов и/или обеспечить вывод результатов вычислений на их основе. Например, если table1 имеет столбцы с именами a, b и c, то можно выполнить запрос:

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

(при этом подразумевается, что столбцы b и c содержат числовые данные). Более подробно это описано в 4.3.

FROM table1 является простейшим примером table_expression, которое содержит всего одну таблицу. Вообще table_expression может быть сложной конструкцией из таблиц, объединений и подзапросов. Допустимо так же опустить table_expression и использовать механизм запросов как простейший калькулятор:

```
SELECT 3 * 4;
```

Это еще более удобно, если выражение в select_list возвращает значения. Например, таким способом возможен вызов функций:

```
SELECT random();
```

4.2. Табличные выражения

Результатом вычисления *табличного выражения* является таблица. Табличное выражение содержит предложение FROM, за которой могут следовать предложения WHERE, GROUP BY и HAVING. Простое табличное выражение просто указывает на какую-либо таблицу на диске, так называемую базовую таблицу, но для модификации и комбинирования

базовых таблиц различными способами могут быть использованы более сложные выражения.

Необязательные предложения WHERE, GROUP BY и HAVING описывают порядок трансформаций, осуществляемых над таблицей, полученной в результате выполнения конструкции FROM. В результате всех этих трансформаций формируется виртуальная таблица, строки которой обрабатываются в соответствии со списком выборки для вычисления результирующих столбцов и строк запроса.

4.2.1. FROM

Конструкция FROM создает виртуальную таблицу из одной или более таблиц, заданных в виде списка, разделенного запятыми:

```
FROM table_reference [, table_reference [, ...]]
```

Ссылкой на таблицу table_reference может быть имя таблицы (возможно полное) или результат выполнения подзапроса, объединения (JOIN) или их сложной комбинации. Если в конструкции FROM указано более одной таблицы, то выполняется их кросс-объединение (то есть, образуется декартово произведение строк; см. ниже). Результатом FROM является промежуточная виртуальная таблица, которая может затем быть преобразована условиями GROUP BY и HAVING в результат выражения таблицы в целом.

Когда ссылка на таблицу ссылается на таблицу, которая является родительской в некоторой табличной иерархии, то, если не указано ключевое слово ONLY, выбираются строки не только этой таблицы, но и всех ее наследников. Однако в список столбцов полученной в результате этого таблицы будут входить только столбцы таблицы, имя которой указано в ссылке на таблицу, а столбцы, добавленные в наследниках, будут проигнорированы.

Вместо указания ключевого слова ONLY, существует возможность указать символ * после имени таблицы для явного указания включения в результат строк дочерних таблиц. Указание * не является необходимым, т. к. это поведение используется по умолчанию (если это не изменено с помощью конфигурационного параметра sql_inheritance). В то же время указание * может быть удобно для более наглядного выделения этого варианта запроса.

4.2.1.1. Соединенные таблицы

Соединенной таблицей называется «виртуальная таблица», формируемая из строк двух таблиц (реальных или виртуальных) в соответствии с правилами указанного типа соединения. Доступны внутренний (INNER), внешний (OUTER) и перекрестное (CROSS) типы соединений (JOIN). Общий синтакс соединений таблиц следующий:

```
T1 join_type T2 [ join_condition ]
```

Объединения всех типов могут быть соединены вместе; быть вложенными: один или оба T1 и T2 могут быть соединениями. Скобки могут быть использованы в JOIN для

указания порядка соединения. При отсутствии скобок JOIN исполняется слева направо.

Примечание. Объединение таблиц с использованием оператора , не является тем же самым, что и объединение таблиц без него. Например, T1 CROSS JOIN T2 INNER JOIN T3 не то же самое, что T1, T2 INNER JOIN T3, потому что на условие может ссылаться T1 в первом случае, но не во втором.

Типы соединений

Перекрестное соединение (cross join):

```
t1 CROSS JOIN t2
```

Результирующая таблица будет содержать все возможные комбинации строк таблиц t1 и t2 (декартово произведение), состоящие из всех столбцов таблицы t1, за которыми следуют все столбцы таблицы t2. Если эти таблицы имеют m и n строк, соответственно, то результирующая таблица будет иметь m*n строк.

FROM T1 CROSS JOIN T2 эквивалентно FROM T1, T2. Также это эквивалентно FROM T1 INNER JOIN T2 ON TRUE (см. ниже).

Возможные соединения:

```
t1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN t2
  ON boolean_expression
t1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN t2
  USING ( join column list )
t1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN t2
```

Ключевые слова INNER и OUTER необязательны в любой из форм. По умолчанию подразумевается INNER. Ключевые слова LEFT, RIGHT и FULL предполагают внешнее (OUTER) соединение.

Условие объединения определяется выражениями ON и USING или неявно — ключевым словом NATURAL. Оно определяет, какие строки из объединяемых таблиц будут считаться «совпавшими».

Условие ON является наиболее общим видом условия объединения. Оно представляет собой логическое выражение того же вида, которое используется в выражении WHERE. Строки из таблиц t1 и t2, для которых это выражение имеет значение true считаются совпавшими.

Выражение USING является сокращенной формой записи выражения ON специального вида. Оно представляет собой список столбцов, общих для объединяемых таблиц, разделенных запятыми, и требует равенства значений из таблиц t1 и t2 для каждого из этих столбцов. Как следствие, в результирующую таблицу попадает только по одному столбцу для каждой такой пары, за которыми следуют все остальные столбцы таблиц. Таким образом, выражение:

```
USING (a, b, c)
```

эквивалентно выражению:

```
ON (t1.a = t2.a AND t1.b = t2.b AND t1.c = t2.c)
```

с тем исключением, что если используется выражение ON, то в результирующей таблице будет по два столбца a, b и c, тогда как для USING их будет только по одному (и они будут выданы первыми, если используется SELECT *).

Ключевое слово NATURAL является сокращенной формой выражения USING со списком всех общих столбцов таблиц t1 и t2. Так же как и для USING для таких столбцов в результирующей таблице появится только по одному из них. Если общих колонок нет, поведение NATURAL аналогично CROSS JOIN.

Возможными типами таких объединений являются:

- INNER JOIN — для каждой строки из таблицы t1 в результирующую таблицу попадут те ее комбинации со строками из таблицы t2, для которых выполняется условие соединения;
- LEFT OUTER JOIN — сначала выполняется INNER JOIN для таблиц t1 и t2 с указанным условием. Затем результирующая таблица дополняется теми строками из таблицы t1, для которых условие соединения не было выполнено ни с одной из строк из таблицы t2. При этом соответствующие таблице t2 столбцы заполняются NULL-значениями. Таким образом, результирующая таблица включает в себя, по крайней мере, одну строку для каждой строки из таблицы t1;
- RIGHT OUTER JOIN — сначала выполняется INNER JOIN для таблиц t1 и t2 с указанным условием. Затем результирующая таблица дополняется теми строками из таблицы t2, для которых условие соединения не было выполнено ни с одной из строк из таблицы t1. При этом соответствующие таблице t1 столбцы заполняются NULL-значениями. Таким образом, результирующая таблица включает в себя, по крайней мере, одну строку для каждой строки из таблицы t2;
- FULL OUTER JOIN — сначала выполняется INNER JOIN для таблиц t1 и t2 с указанным условием. Затем результирующая таблица дополняется теми строками из таблицы t1, для которых условие соединения не было выполнено ни с одной из строк из таблицы t2. При этом соответствующие таблице t2 столбцы заполняются NULL-значениями. Затем результирующая таблица дополняется теми строками из таблицы t2, для которых условие соединения не было выполнено ни с одной из строк из таблицы t1. При этом соответствующие таблице t1 столбцы заполняются NULL-значениями. Таким образом, результирующая таблица включает в себя, по крайней мере, одну строку для каждой строки из обеих таблиц.

Соединения всех типов можно записывать в виде цепочки соединений либо в виде вложенных соединений, т. е. каждая из таблиц t1 и t2 может представлять собой соедине-

ние таблиц. Круглые скобки могут быть использованы вокруг соединений для управления порядком их выполнения. Если скобки не используются, то соединения вкладываются слева направо.

Поясним все вышесказанное на примере. Имеются таблица `t1`:

```
num | name
 ---+---
   1|a
   2|b
   3|c
```

и таблица `t2`:

```
num |value
 ---+---
   1|xxx
   3|yyy
   5|zzz
```

В результате выполнения различных соединений получится:

```
=> 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);
```

```
\begin{verbatim*}
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, указываемое после ключевого слова ON необязательно должно относиться непосредственно к соединению. Это может быть полезно для ряда запросов, но требует большой осторожности при применении. Например:

```
=> 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)
```

В тоже время, если поместить ограничение в предложение WHERE, результат будет иным:

```
=> 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 перед соединением, в то время как ограничение, помещенное в предложение WHERE обрабатывается после соединения.

4.2.1.2. Псевдонимы таблиц и столбцов

Таблице или сложному табличному выражению (table_reference) можно дать временное имя, используемое на следующем шаге построения результирующей таблицы. Такое имя называется «псевдонимом таблицы» (alias). Оно определяется следующим образом:

```
FROM table_reference AS alias
```

или:

```
FROM table_reference alias
```

Ключевое слово AS не является обязательным. Псевдонимом при этом может быть произвольный идентификатор.

Типичным примером использования табличных псевдонимов является сокращение длинных имен таблиц, для того чтобы сделать выражение более коротким и, следовательно, более читабельным:

```
SELECT * FROM some_very_long_table_name s
  JOIN another_fairly_long_name a ON s.id = a.num;
```

Псевдоним становится новым именем таблицы в пределах текущего запроса. После его объявления сослаться на таблицу по ее прежнему имени невозможно. Таким образом, следующий запрос недопустим:

```
SELECT * FROM my_table AS m WHERE my_table.a > 5; -- неверно
```

Псевдонимы таблиц часто являются удобным средством сокращения записи, однако иногда их используют при объединении таблицы с ней самой:

```
SELECT * FROM people AS mother
  JOIN people AS child ON mother.id = child.mother_id;
```

Кроме этого псевдонимы необходимы в случае использования подзапросов (4.2.1.3).

Круглые скобки используются для разрешения неоднозначности. В приведенном ниже примере первое выражение назначает псевдоним b второму вхождению my_table, тогда как второе назначает псевдоним результату объединения:

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

Другим способом использования табличных псевдонимов является задание временных имен для столбцов таблицы:

```
FROM table_reference [AS] alias ( column1 [, column2 [, ...]] )
```

Если псевдонимов столбцов задано меньше, чем столбцов исходной таблицы, то оставшиеся столбцы не переименовываются. Этот способ использования псевдонимов может быть полезен при выполнении объединений таблиц с ними самими или для именования подзапросов.

Когда псевдоним таблицы применяется к результату объединения с использованием любой из описанных форм, то он скрывает оригинальные имена таблиц (или их псевдонимы), участвующие в этом выражении. Например:

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

является допустимой формой запроса, а форма:

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

недопустима. Табличный псевдоним a не виден вне псевдонима c.

4.2.1.3. Подзапросы

Подзапросы всегда должны быть заключены в круглые скобки и всегда должны иметь псевдоним (см. 4.2.1.2). Например:

```
FROM (SELECT * FROM table1) AS alias_name
```

Этот пример эквивалентен:

```
FROM table1 AS alias_name
```

В подзапросах могут использоваться группировки и агрегирование.

Подзапросом может выступать и список значений VALUES:

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
  AS names(first, last)
```

При этом требуется обязательное указания псевдонима таблицы. Назначение псевдонимов полям VALUES необязательно, но является хорошей практикой. Подробности см. 4.7.

4.2.1.4. Табличные функции

Табличными функциями называются функции, результатом работы которых является набор строк, состоящих как из базовых, так и составных типов данных. Подобные функции используются как таблицы, представления и подзапросы в конструкции FROM запроса. Возвращаемые функцией столбцы могут быть использованы в конструкциях SELECT, JOIN или WHERE, как и столбцы других объектов.

Если функция возвращает базовый тип данных, единственный возвращаемый столбец называется как сама функция. В случае составных типов возвращаемые столбцы называются, как соответствующие атрибуты типа.

Табличным функциям в конструкции FROM могут быть назначены псевдонимы, хотя это необязательно. При отсутствии псевдонима в качестве имени результирующей таблицы используется имя функции:

```
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 удобно объявлять табличные функции, как возвращающие разный набор столбцов в зависимости от того, как они были вызваны. Для этого функция должна быть объявлена, как возвращающая псевдотип `record`. При использовании такой функции в запросе ожидаемый состав возвращаемой строки должен быть указан в самом запросе, чтобы система могла знать, как разобрать и спланировать запрос.

```
SELECT *
  FROM dblink('dbname=mydb', 'select proname, prosrc from pg_proc')
    AS t1(proname name, prosrc text)
WHERE proname LIKE 'bytea%';
```

Функция `dblink` выполняет запрос на удаленном сервере. Она объявлена, как возвращающая тип `record`, т. к. может быть использована для выполнения любых запросов. Действительный набор столбцов должен быть указан в запросе, чтобы анализатор знал, например, как развернуть `*`.

4.2.1.5. `LATERAL` подзапросы

Примечание. `LATERAL` подзапросы доступны, начиная с версии PostgreSQL 9.6.

Подзапросы, появляющиеся в предложении `FROM`, могут быть предварены ключевым словом `LATERAL`. Это дает им возможность ссылаться на столбцы, введенные предыдущими предложениями `FROM` (без ключевого слова `LATERAL` каждый подзапрос анализируется независимо, и следовательно не может ссылаться на другие элементы предложения `FROM`).

Ключевое слово `LATERAL` может быть указано перед табличными функциями, но это не является обязательным; аргументы функции в любом случае могут ссылаться на столбцы, введенные предыдущими предложениями `FROM`.

Элемент с ключевым словом `LATERAL` может появляется на верхнем уровне списка `FROM`, или внутри дерева соединений. В последнем случае он может ссылаться на любой элемент слева в предложении `JOIN`, правее которого он расположен.

Когда элемент списка `FROM` содержит перекрестные `LATERAL` ссылки, процесс обработки выглядит следующим образом: для каждой строки элемента `FROM`, предоставляющего требуемые столбцы, или набора строк элементов `FROM`, предоставляющих столбцы, на основании предоставленных значений столбцов строки или набора строк вычисляется `LATERAL` элемент. Результирующая строка(строки) соединяется обычном способом со строками, на основании которых она вычислена. Это повторяется для всех строк или наборов строк

таблиц, предоставляющих требуемые столбцы.

Простейшим примером использования LATERAL является:

```
SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE bar.id = foo.bar_id) ss;
```

Приведенный пример не слишком полезен, поскольку приводит к тому же результату, что и более привычный запрос:

```
SELECT * FROM foo, bar WHERE bar.id = foo.bar_id;
```

LATERAL в первую очередь полезен, когда используемые столбцы необходимы для вычисления строк, которые должны быть присоединены. Основным применением является предоставление значения аргументу функции, возвращающей набор строк. К примеру, если функция vertices(polygon) возвращает набор вершин многоугольника, то следующим запросом можно определить близко лежащие вершины многоугольников:

```
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 же запрос может быть записан иначе:

```
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 еще несколькими эквивалентными способами (как было уже сказано, ключевое слово LATERAL в этом примере необязательно, но добавлено для наглядности).

Зачастую удобно использовать левое соединение (LEFT JOIN) с LATERAL подзапросом, что бы исходная строка появлялась в результирующей выборке, даже если LATERAL подзапрос не возвращает для нее результата. Например, если get_product_names() возвращает наименование товаров производителя, но некоторые производители в настоящее время товаров не производят, их можно отобрать следующим запросом:

```
SELECT m.name
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true
WHERE pname IS NULL;
```

4.2.2. WHERE

Синтаксис конструкции WHERE:

```
WHERE search_condition
```

где *условием поиска* (search_condition) является любое выражение (см. 1.2), возвращающее логическое значение (true или false).

После обработки конструкции FROM каждая строка, получившаяся в виртуальной

таблице, проверяется на соответствие условию поиска. Если результатом является `true`, то строка остается в результирующей таблице, в противном случае (если результатом является `false` или NULL-значение) строка отбрасывается. Обычно условие поиска ссылается на несколько столбцов обрабатываемой таблицы, однако это не является обязательным условием.

В некоторых случаях выражения объединения в конструкции `FROM` можно переписать с использованием условия в конструкции `WHERE`. Например, приведенные ниже табличные выражения эквивалентны:

```
FROM a, b WHERE a.id = b.id AND b.val > 5

FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5

FROM a NATURAL JOIN b WHERE b.val > 5
```

Запросы с использованием соединений в конструкции `FROM` не столь переносимы (поскольку не везде реализованы, хотя и регламентируются стандартом SQL), как запросы с использованием конструкции `WHERE`. Для внешних (`OUTER`) объединений альтернативы нет и они должны осуществляться в конструкции `FROM`. Конструкциям `ON` и `USING` во внешнем соединении нет эквивалентного условия в конструкции `WHERE`, поскольку они определяют не только удаление строк из конечного результата, но и их добавление.

Вот несколько примеров использования конструкции `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 EXIST (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

где fdt — это таблица, полученная в результате выполнения конструкции FROM. Строки, которые не соответствуют условию поиска в конструкции WHERE, удаляются из fdt. Следует обратить внимание на использование скалярных подзапросов в качестве вычисляемых выражений. Так же как и прочие запросы, подзапросы могут содержать сложные табличные выражения. Важно отметить, как используются ссылки на fdt из подзапросов. Указание столбца c1 как fdt.c1 в подзапросе необходимо только тогда, когда это имя конфликтует с именами столбцов в его собственной виртуальной таблице, однако для ясности рекомендуется всегда указывать ссылку в полной форме.

4.2.3. GROUP BY и HAVING

После фильтрации строк виртуальной таблицы с помощью условия поиска конструкции WHERE получившаяся таблица может быть сгруппирована с помощью конструкции GROUP BY, строки получившейся при этом таблицы в дальнейшем могут быть отфильтрованы с помощью конструкции HAVING:

```
SELECT select_list
  FROM ...
  [WHERE ...]
  GROUP BY grouping_column_reference [, grouping_column_reference]...
```

Конструкция GROUP BY используется для группировки строк таблицы, которые содержат одинаковые значения во всех перечисленных в ней столбцах. Порядок, в котором перечислены столбцы, не существенен. Конструкция действует так, что каждая группа строк, имеющих совпадающие значения в указанных столбцах, будет представлена одной строкой. Это осуществляется для удаления повторений и (или) вычисления агрегатных значений для таких групп. 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 нельзя использовать форму:

```
SELECT * FROM test1 GROUP BY x
```

так как столбец y не содержит одиночных значений, которые могут быть ассоциированы с каждой группой. Столбцы, по которым осуществляется группировка, могут быть указаны в списке выборки (select_list), поскольку для каждой группы содержат одно значение.

Как правило, если таблица группируется, то столбцы, которые не используются для группировки, доступны в списке выборки только в агрегатных выражениях. Например:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;

 x | sum
---+---
 a |   4
 b |   5
 c |   2
(3 rows)
```

где sum() — это агрегатная функция, которая вычисляет сумму значений столбца y в пределах группы. Подробнее агрегирующие функции рассмотрены в 6.20.

Группировка без использования агрегирования является эффективным способом получения уникальных значений столбцов. Так же это может быть достигнуто применением конструкции DISTINCT (см. 4.3.3).

Другой пример: вычисление общей суммы продаж по каждому виду товара:

```
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 столбцы product_id, p.name и p.price должны быть указаны в конструкции GROUP BY, т. к. они появляются в списке выборки вне агрегатных функций. Столбец s.units не заносится в список GROUP BY, т. к. он используется только в агрегатном выражении (sum()), которое и вычисляет суммы продаж по товару. Для каждого товара, запрос возвращает суммарную строку о всех его продажах.

В стандарте SQL конструкция GROUP BY может группировать только столбцы исходной таблицы, однако PostgreSQL допускает также группировку столбцов из списка выборки. Допустима также группировка по значениям выражений вместо простых наименований столбцов.

Если таблица сгруппирована с использованием конструкции GROUP BY, но при этом интересны только некоторые группы, для отбора нужных групп может использоваться

конструкция HAVING, работающая как конструкция WHERE:

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_expression
```

В выражении конструкции HAVING могут быть использованы ссылки как на столбцы с результатами группировки, так и на исходные столбцы. Например:

```
=> 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)
```

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 этом примере в конструкции WHERE отбираются строки по столбцу, который не входит в список группируемых, а в конструкции HAVING итоговая таблица ограничивается только группами, общая продажная стоимость которых превышает 5000. Таким образом, агрегатные выражения не обязаны быть одинаковыми во всех частях запроса.

Если запрос содержит вызовы агрегатный функций, но не содержит предложение GROUP BY, группировка все-равно будет осуществлена: результатом будет одна сгруппированная строка (или не будет никаких строк, если данная сгруппированная строка будет исключена конструкцией HAVING). Тоже самое будет, если запрос содержит конструкцию HAVING, даже без каких-либо вызовов агрегатных функций или конструкции GROUP BY.

4.2.4. Обработка оконной функции

Если запрос содержит любые оконные функции (см. 1.2.8), названные функции вычисляются после выполнения любых группировок, агрегирования и фильтрации HAVING. Таким образом, если запрос использует любые агрегирования, GROUP BY или HAVING, то строки, выданные оконными функциями, являются группами строк вместо оригинальных

строк таблицы из FROM/WHERE.

Когда используется несколько оконных функций, все оконные функции, имеющие синтаксически эквивалентные выражения PARTITION BY и ORDER BY в их определениях окон, гарантировано вычисляются за один проход данных. Следовательно, они выдают одинаковый порядок сортировки, даже если ORDER BY не уникально определяет сортировку. Однако подобные гарантии отсутствуют в случае оконных функций, использующих различные спецификации PARTITION BY и ORDER BY. В таких случаях сортировка осуществляется между проходами при вычислении оконных функций, и сортировка не гарантирует эквивалентности результатов сортировки результату, полученному с ORDER BY.

Оконные функции всегда требуют предварительно отсортированных данных. Следовательно, результаты выполнения запроса должны быть упорядочены в соответствии с одним из выражений PARTITION BY/ORDER BY, указанных в оконных функциях. Однако не рекомендуется на это полагаться. Необходимо использовать заключительное выражение ORDER BY для обеспечения гарантии сортировки результата правильным образом.

4.3. Списки выборки SELECT

Табличное выражение в команде SELECT формирует промежуточную виртуальную таблицу комбинированием таблиц, видов, фильтрацией строк, группировкой и т. д. Эта таблица в итоге обрабатывается с помощью *списка выборки*. Этот список определяет, какие *столбцы* промежуточной таблицы будут представлены в результате.

4.3.1. Элементы списка выборки SELECT

Простейшим видом списка выборки является *, который обеспечивает вывод всех столбцов, сформированных табличным выражением. В остальных случаях список выборки представляет собой разделенный запятыми список вычисляемых выражений (см. 1.2). Например, это может быть список имен столбцов:

```
SELECT a, b, c FROM ...
```

Столбцы с именами a, b и c могут быть как непосредственно именами столбцов таблиц, указанных в конструкции FROM, так и псевдонимами. Пространство имен, доступное в списке выборки, является тем же самым, что и в конструкции WHERE, если не используется группировка, или тем же самым, что в конструкции HAVING.

Если несколько таблиц содержат столбцы с одинаковым именем, должны быть указаны и имена соответствующих таблиц:

```
SELECT tbl1.a, tbl2.b, tbl1.c FROM ...
```

При выполнении запроса к нескольким таблицам символ * так же может быть использован для выбора всех столбцов конкретной таблицы:

```
SELECT tbl1.*, tbl2.a FROM ...
```

Если в списке выборки используется какое-либо вычисляемое выражение, то оно

добавляет новый виртуальный столбец в итоговую таблицу. Это выражение вычисляется один раз для каждой строки с ее значениями, подставляемыми вместо ссылок на столбцы. Однако выражение не обязано иметь какие-либо ссылки на столбцы в конструкции FROM, они могут быть, например, и константными арифметическими выражениями.

4.3.2. Имена столбцов

Элементам списка выборки может быть присвоено новое имя, используемое при дальнейшей обработке. Под «дальнейшей обработкой» понимается необязательная сортировка или клиентское приложение (например, для отображения заголовка таблицы при выводе). Например:

```
SELECT a AS value, b + c AS sum FROM ...
```

Если для столбца имя не задано, то система присвоит ему имя по умолчанию. Для простых ссылок на столбцы это имя совпадает с именем указанного столбца. Для вызовов функций им является имя функции. Для сложных выражений система будет генерировать имена автоматически.

Ключевое слово AS является обязательным только в случае совпадения задаваемого нового имени столбца с ключевыми словами PostgerSQL. Во избежание совпадения имени столбца с ключевым словом, имя столбца может быть заключено в двойные кавычки. Например, VALUE является ключевым словом, так что следующий запрос неверен:

```
SELECT a value, b + c AS sum FROM ...
```

верным будет:

```
SELECT a "value", b + c AS sum FROM ...
```

Для того чтобы избежать совпадения имен с какими-нибудь новыми ключевыми словам, которые могут появиться в будущем, рекомендуется всегда использовать AS или заключать имена в двойные кавычки.

Примечание. Приведенный способ именования столбцов отличается от рассмотренного в конструкции FROM (см. 4.2.1.2). Фактически синтаксис позволяет задавать новые имена столбцов дважды, но в качестве результирующих выбираются указанные именно в списке выборки.

4.3.3. DISTINCT

После обработки списка выборки получившаяся таблица может быть дополнительно обработана с целью исключения строк с повторяющимися значениями. Для этого используется ключевое слово DISTINCT, записанное сразу же после ключевого слова SELECT:

```
SELECT DISTINCT select_list ...
```

(вместо ключевого слова DISTINCT может быть указано ALL, что соответствует действию по умолчанию для возврата все строк).

При этом две строки считаются различными, если они различаются значением, по

крайней мере, в одном столбце. При этом сравнении NULL-значения считаются эквивалентными.

В то же время, существует возможность указать произвольное выражение, по которому будет определяться эквивалентность строк:

```
SELECT DISTINCT ON (expression [, expression ...]) select_list ...
```

где `expression` — это произвольное скалярное выражение, вычисляемое для всех строк. Наборы строк, для которых все выражения являются эквивалентными, также считаются эквивалентными, и только первая строка набора попадает в результирующую выборку. Первая строка набора непредсказуема до тех пор, пока результат запроса не отсортирован по достаточному количеству столбцов для обеспечения однозначного их порядка до применения фильтра `DISTINCT`. (Обработка `DISTINCT ON` осуществляется после выполнения сортировки `ORDER BY`.)

Конструкция `DISTINCT ON` не является частью SQL-стандарта и иногда считается нежелательной, поскольку может приводить к непредсказуемым результатам. Вместо нее применяются конструкции `GROUP BY` и подзапросы в конструкции `FROM`.

4.4. Комбинирование запросов

Результаты двух запросов могут быть скомбинированы с помощью набора операций объединения, пересечения и вычитания:

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

При этом запросы `query1` и `query2` могут содержать любые ранее рассмотренные конструкции. Операции комбинирования могут быть вложены или выполнены цепочкой. Например:

```
query1 UNION query2 UNION query3
```

что в действительности означает:

```
(query1 UNION query2) UNION query3
```

`UNION` добавляет к строкам запроса `query1` строки запроса `query2`. (При этом не гарантируется именно этот порядок появления строк в итоговой таблице.) Более того, при этом исключаются повторяющиеся строки, так же как и при `DISTINCT`, если не задано ключевое слово `ALL`.

`INTERSECT` возвращает все строки, которые присутствуют одновременно и в запросе `query1`, и в `query2`. При отсутствии указания `INTERSECT ALL` дублирующиеся строки исключаются.

`EXCEPT` возвращает все строки результата запроса `query1`, которых нет в результате запроса `query2`. (Иногда это называется «разностью запросов».) Так же исключаются

дублирующиеся строки при отсутствии указания EXCEPT ALL.

Для корректного вычисления объединения, пересечения и вычитания двух запросов оба запроса должны быть *совместимы*, т. е. они должны возвращать одинаковое количество столбцов и одноименные столбцы должны иметь совместимые типы данных.

4.5. Сортировка строк

После формирования итоговой таблицы (и после обработки списка выборки), она может быть отсортирована. Если сортировка не используется, то строки возвращаются в неопределенном порядке. Порядок возврата строк в этом случае зависит от порядка просмотра таблиц, типов использованных объединений и физического расположения данных на диске, т. е. факторов, на которые нельзя положиться. Гарантированный порядок строк может быть обеспечен только применением явной сортировки.

Конструкция ORDER BY определяет этот порядок:

```
SELECT select_list
  FROM table_expression
  ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
         [, sort_expression2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

Выражением сортировки (sort_expression) может быть любое выражение, допустимое для списка выборки, например:

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

Если определен более чем один столбец для сортировки, то каждый следующий столбец используется для сортировки строк в пределах эквивалентности строк, заданной предыдущими столбцами. Для каждого столбца, указанного в спецификации сортировки, может быть задан порядок сортировки ASC (в порядке увеличения) и DECS (в порядке уменьшения). По умолчанию используется сортировка в порядке увеличения (ASC). При сортировке в порядке увеличения меньшие значения помещаются вперед, где смысл «меньший» определяется в терминах оператора <. Аналогично сортировка в порядке уменьшения определяется оператором >.

Примечание. Фактически, PostgreSQL использует класс оператора B-tree по умолчанию для типа данных указанного выражения сортировки, чтобы определить порядок сортировки при использовании ASC и DESC. Обычно типы данных устанавливаются таким образом, чтобы операторы < и > отвечали порядку сортировки, но разработчик пользовательского типа данных может выбрать что-либо другое.

Порядок вывода NULL-значений перед или после определенных (не NULL) может быть задан с помощью опций NULLS FIRST и NULLS LAST, соответственно. По умолчанию NULL-значения считаются больше любых определенных, т. е. по умолчанию для сортировок DESC применяется NULLS FIRST и NULLS LAST в противном случае.

Опции сортировки считаются независимыми для каждого выражения. Например:

```
ORDER BY x, y DESC
```

означает:

```
ORDER BY x ASC, y DESC
```

а не:

```
ORDER BY x DESC, y DESC
```

В качестве *выражения сортировки* могут использоваться как имена результирующих столбцов, так и их порядковые номера:

```
SELECT a+b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

Оба запроса сортируют выборку по первому результирующему столбцу. В тоже время имя отдельного результирующего столбца не может быть частью выражения сортировки, например следующее выражение неверно:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;
```

Это ограничение введено во избежание неоднозначности. Существует некоторая неоднозначность, если элемент выражения сортировки совпадает одновременно с именем результирующего столбца и именем столбца табличного выражения. В любом случае используется результирующий столбец. Это может возникнуть при переименовании с помощью конструкции `AS` результирующего столбца в имя какого-нибудь столбца используемых в запросе таблиц.

Конструкция `ORDER BY` может быть применена к результату выполнения комбинаций `UNION`, `INTERSECT` и `EXCEPT`, но в этом случае допустима только сортировка по именам или порядковым номерам результирующих столбцов, но не по выражениям.

4.6. LIMIT и OFFSET

Ключевые слова `LIMIT` и `OFFSET` позволяют ограничить количество возвращаемых в результате выполнения запроса строк:

```
SELECT select_list
  FROM table_expression
  [ ORDER BY ... ]
  [ LIMIT { number | ALL } ] [ OFFSET number ]
```

Если задано количество `number` для ключевого слова `LIMIT`, то в результирующей таблице будет не более, чем заданное количество строк (но может оказаться меньше, если результатом выполнения запроса является меньшее количество строк). Использование конструкции `LIMIT ALL` эквивалентно неиспользованию конструкции `LIMIT` вообще.

`OFFSET` определяет количество строк, которые будут пропущены и не попадут в конечный результат. `OFFSET 0` эквивалентно неиспользованию конструкции `OFFSET`. Если используются обе конструкции: и `LIMIT`, и `OFFSET`, то сначала будет пропущено количество

строк, указанное в конструкции OFFSET, а затем будет выведено количество строк, заданное в конструкции LIMIT.

При использовании конструкции LIMIT необходимо использовать конструкцию ORDER BY для определения порядка, иначе нельзя предсказать, какие строки попадут в результирующую выборку.

Оптимизатор запросов учитывает конструкцию LIMIT при определении плана выполнения запроса, так что можно получить различные планы выполнения запроса в зависимости от конкретных значений, указанных в конструкциях LIMIT и OFFSET. Следовательно, использование различных значений для LIMIT/OFFSET для получения различных результирующих наборов строк, без использования явно заданного порядка сортировки, приводит к противоречивым результатам. Это не является ошибкой, т. к. стандарт SQL не определяет какого-либо порядка получения строк без использования явной сортировки.

Строки, пропущенные конструкцией OFFSET, в действительности все равно обрабатываются сервером, так что использование OFFSET с большими значениями может быть неэффективным.

4.7. Списки VALUES

Конструкция VALUES дает возможность создать «таблицу констант», которая может быть использована в запросе без действительного ее создания и сохранения на диске. Синтаксис:

```
VALUES ( expression [, ...] ) [, ...]
```

Каждый окруженный скобками список значений создает одну строку таблицы. Списки должны содержать одинаковое количество элементов (т. е. количество столбцов таблицы), и соответствующие значения в каждом списке должны быть совместимых типов. Действительный тип данных, ассоциированный с каждым результирующим столбцом, определяется по тем же правилам, что и при выполнении комбинации UNION.

Например:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

возвращает таблицу, состоящую из двух столбцов, и содержащую три строки, что эквивалентно:

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

По умолчанию PostgreSQL присваивает результирующим столбцам таблице, полученной конструкцией VALUES, имена типа column1, column2, и т. д. Имена столбцов не

регламентируются стандартом SQL и в различных СУБД могут быть разными, так что лучше переопределять имена по умолчанию псевдонимами:

```
=> SELECT * FROM (VALUES (1, 'one'), (2, 'two'), (3, 'three')) AS t (num,letter);
num | letter
-----+--------
1 | one
2 | two
3 | three
(3 rows)
```

Синтаксически конструкция VALUES с последующим списком выражений рассматривается как:

```
SELECT select_list FROM table_expression
```

и может быть использована во всех местах запроса, допустимых для конструкции SELECT. Например, как часть комбинации UNION или иметь присоединенные спецификации сортировки и ограничения выборки (ORDER BY, LIMIT с или без OFFSET). Наиболее часто конструкция VALUES используется как источник данных для команды INSERT или в качестве подзапроса.

4.8. Запросы WITH (Общие табличные выражения)

Конструкция WITH предоставляет возможность создавать подзапросы, которые можно в дальнейшем использовать в большом запросе типа SELECT. Подзапросы, создаваемые в конструкции WITH, которые часто называют как *Общие табличные выражения* или CTE (Common Table Expressions), можно рассматривать как временные таблицы, существующие только на момент исполнения основного запроса. Каждый вспомогательный оператор в конструкции WITH может быть одним из SELECT, INSERT, UPDATE или DELETE; а само предложение WITH прикрепляется к первичному оператору, который также может быть одним из SELECT, INSERT, UPDATE или DELETE.

4.8.1. SELECT в WITH

Основным способом использования SELECT в WITH является упрощение сложного запроса путем его разбиения на более простые, например:

```
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;
```

Приведенный запрос отображает суммы продаж по продуктам, но только в регионах с максимальным количеством продаж. Конструкция `WITH` определяет два вспомогательных запроса с именами `regional_sales` и `top_regions`, где вывод `regional_sales` используется в `top_regions`, а вывод `top_regions` используется в первичном запросе `SELECT`. Этот пример может быть написан без использования конструкции `WITH`, но тогда потребуется два уровня вложенных подзапросов. Гораздо легче следовать вышеописанному методу.

Необязательный параметр `RECURSIVE` изменяет конструкцию `WITH` из более-менее обычной, в конструкцию, которая может обеспечить решение задач, невозможных при использования обычного SQL. При использовании `RECURSIVE` конструкция `WITH` может ссылаться на саму себя. В качестве примера рассмотрена арифметическая сумма ряда до 100:

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

Основная форма использования рекурсивного запроса с `WITH` всегда состоит из нерекурсивного элемента, комбинации `UNION` (или `UNION ALL`), затем рекурсивного элемента, где только рекурсивный элемент может ссылаться на результат конструкции `WITH`. При этом рекурсивный запрос выполняется следующим образом:

1) выполняется нерекурсивная часть. Для `UNION` (но не `UNION ALL`) удаляются повторяющиеся строки и результат помещается во временную *рабочую таблицу*;
2) пока рабочая таблица не пуста, повторяются шаги:
    а) выполняется рекурсивная часть, при этом используется текущее содержимое рабочей таблицы. Для `UNION` (но не `UNION ALL`) удаляются повторяющиеся строки и строки, совпадающие с содержимым рабочей таблицы, и результат

помещается во временную *промежуточную таблицу*;

б) содержимое рабочей таблицы заменяется содержимым промежуточной, после чего промежуточная очищается.

Примечание. Строго говоря, данный процесс является итерационным, а не рекурсивным, но RECURSIVE является терминологическим выбором комитета по стандартам SQL.

В приведенном выше примере рабочая таблица на каждом шагу содержала только одну строку, содержащую последовательно значения от 1 до 100. На сотом шаге выборка, соответствующая конструкции WHERE, была пуста, в результате чего вычисления были прерваны.

Рекурсивные запросы обычно используются для работы с иерархическими или древовидными структурами данных. Более полезным примером является выборка всех компонентов продукта как непосредственно в него входящих, так и входящих в его компоненты, если в таблице используется только непосредственное вхождение компонентов:

```
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
```

При использовании рекурсивных запросов надо быть уверенным, что рекурсивная часть запроса в конечном счете вернет пустой набор, или запрос будет выполняться бесконечно. Иногда использование UNION вместо UNION ALL позволяет достичь этого путем удаления строк, дублирующих предыдущие. Однако, не всегда в цикл вовлекаются полностью совпадающие строки, иногда необходимо проверять ограниченный набор полей для определения, была ли строка обработана ранее. Стандартным методом для выхода из этой ситуации применяется массив уже обработанных значений. Например, запрос поиска в графе по полям связи:

```
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 зациклится, если граф содержит циклы по полю связи. Поскольку требуется вывод всего графа, смена `UNION ALL` на `UNION` не предотвратит зацикливание. Вместо этого необходимо определять, достигнута ли строка, которая была уже ранее обработана, следуя по полю связи. Для этого добавляется два столбца `path` и `cycle` в запрос:

```
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 зацикливания, массив значений часто используется для непосредственного запоминания пути, достигнутого при поиске конкретной строки.

В общем случае, для распознавания цикла требуется более одного поля, в этом случае используется массив строк. Например, при сравнении полей `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. Синтаксис `ROW()` можно не использовать только в общем случае, когда для распознавания цикла применяется одно поле. Это позволяет использовать простой типизированный массив вместо массива составного типа, что более эффективно.

2. Алгоритм выполнения рекурсивного запроса выводит результат в ширину дерева поиска. Для вывода результата по глубине поиска необходимо использовать конструкцию `ORDER BY` для указания сортировки по полю `path`, содержащему путь.

Удобно для проверки возможности зацикливания запроса помещать конструкцию `LIMIT` в основной запрос, использующий `WITH`. Например, следующий запрос без указания `LIMIT` представляет собой бесконечный цикл:

```
WITH RECURSIVE t(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

В данном случае это сработает, т. к. PostgreSQL вычисляет только то количество строк `WITH` подзапроса, которое реально выбирается основным запросом. Не рекомендуется использовать этот способ, поскольку другие системы могут обрабатывать это иначе. Так же это может не сработать при указании явной сортировки или использования результата в объединениях с другими таблицами, потому что в таких случаях внешний запрос обычно в любом случае будет пытаться получить вывод от всех запросов `WITH`.

Одним из полезных свойств подзапросов `WITH` является то, что они вычисляются только один раз за время выполнения основного запроса, даже если на них существует более одного обращения в основном запросе. Следовательно, сложные и затратные вычисления, встречающиеся много раз в основном запросе можно вынести в подзапрос `WITH`, чтобы избежать ненужных затрат вычислительных ресурсов. Другим возможным применением является предотвращение множественного вызова функций с побочными эффектами. С другой стороны, оптимизатор имеет меньше возможностей для передачи возможных ограничений из основного запроса в подзапросы `WITH` по сравнению с обычными подзапросами. WITH-подзапросы в основном вычисляются так, как они написаны без учета того, что некоторые из полученных строк могут быть отклонены основным запросом. (Но вычисление WITH-поздапроса может быть остановлено, если в основном запросе есть ограничение на количество обрабатываемых строк.)

Приведенный выше пример показывает использование конструкции `WITH` с `SELECT`,

но она может быть тем же способом использована и с INSERT, UPDATE или DELETE. В этом случае она фактически предоставляет временную таблицу(ы), которая может быть использована в главной команде.

4.8.2. Операторы, изменяющие данные, в WITH

Совместно с конструкцией WITH могут использоваться операторы, изменяющие данные (INSERT, UPDATE или DELETE). Это позволяет выполнять различные операции в одном запросе. 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;
```

Данный запрос фактически перемещает строки из products в products_log. DELETE в WITH удаляет заданные строки из products, возвращая их содержимое с помощью предложения RETURNING; и затем первичный запрос читает выводимые данные и вставляет их в products_log.

Изящество данного выше примера в том, что предложение WITH прикрепляется к INSERT, вместо подзапроса внутри INSERT. Это необходимо, потому что операторы, изменяющие данные разрешены только в выражениях WITH, которые прикрепляются к оператору верхнего уровня. В то же время применяются обычные правила видимости WITH, так что становится возможным работать с выводом WITH из подзапроса.

Изменяющие данные операторы в WITH обычно имеют конструкцию RETURNING как видно в приведенном выше примере. Вывод, осуществляемый конструкцией RETURNING, не является целевой таблицей оператора, изменяющего данные, а формирует временную таблицу, которая может использоваться в оставшейся части запроса. Если оператор, изменяющий данные в WITH не содержит выражение RETURNING, то временная таблица не формирует и ее нельзя использовать в оставшейся части запроса. Тем не менее, такие операторы будут запущены. Например:

```
WITH t AS (
    DELETE FROM foo
)
DELETE FROM bar;
```

Данный пример удалит все строки из таблиц foo и bar. Количество затронутых запросом строк, выданных клиенту, будет включать только строки, удаленные из bar.

Рекурсивные ссылки на себя в операторах, изменяющих данные, не разрешаются. В некоторых случаях возможно обойти это ограничение сославшись на вывод рекурсивного WITH, например:

```
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);
```

Данный запрос удалит все прямые и непрямые компоненты sub_part продукта product.

Операторы, изменяющие данные в WITH выполняются только один раз и всегда до полного завершения, независимо от того, будет ли первичный запрос читать все (или ничего) из выводимого ими. Следует отметить, что указанное поведение отлично от правила исполнения SELECT в WITH: как было сказано ранее, выполнение SELECT осуществляется до тех пор, пока основной запрос запрашивает выводимые данные.

Операторы в WITH запускаются параллельно друг с другом и с основным запросом. Таким образом, когда в WITH используются операторы, изменяющие данные, порядок в котором фактически происходят заданные обновления является непредсказуемым. Все операторы запускаются в том же самом снимке (snapshot), так что они не могут «видеть» все другие изменения целевых таблиц. Это смягчает эффекты непредсказуемости фактического порядка обновления строк, и означает, что данные, возвращаемые RETURNING являются единственным способом связи изменений между разными операторами в WITH и основным запросом. Вот пример этого:

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM products;
```

внешний SELECT должен вернуть первоначальные цены (prices) перед выполнением UPDATE, в то время как в:

```
WITH t AS (
```

```
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM t;
```

внешний SELECT должен вернуть обновленные данные.

Попытки обновить одну и ту же строку дважды в одном операторе не поддерживаются. Происходит только одно изменение, но надежно предсказать каким будет это изменение не легко (а иногда и невозможно). Это же самое касается также удаления строки, которая была уже обновлена в этом же операторе: выполняется только обновление. В основном следует избегать попытки изменения одной строки дважды в одном операторе. В особенности следует избегать таких операторов в WITH, которые могут влиять на строки, которые уже изменены основным или другим таким же оператором. Изменения, производимые такими операторами будут непредсказуемы.

Таблица, используемая как целевая в WITH операторах, изменяющих данные, не должна иметь ни условных правил, ни правил ALSO или INSTEAD, которые расширяются до нескольких операторов.

5. ТИПЫ ДАННЫХ

В PostgreSQL представлен большой список доступных пользователю встроенных типов. Кроме того, с помощью команды `CREATE TYPE` существует возможность добавлять новые типы данных.

В таблице 3 приведены встроенные типы данных общего назначения. Большая часть альтернативных имен, перечисленных в колонке «Псевдонимы», используется в PostgreSQL по историческим причинам. Кроме того, доступны, но не указаны, некоторые типы данных, которые используются для внутренних нужд PostgreSQL или которые устарели.

Таблица 3 – Типы данных

Имя	Псевдонимы	Описание
`bigint`	`int8`	Знаковое восьмибайтное целое число
`bigserial`	`serial8`	Восьмибайтное целое число с автоинкрементом
`bit [ (n) ]`		Битовая строка фиксированной длины
`bit varying [ (n) ]`	`varbit`	Битовая строка переменной длины
`boolean`	`bool`	Логическое значение (`true`/`false`)
`box`		Четырехугольник на плоскости
`bytea`		Двоичные данные («массив байт»)
`character varying [ (n) ]`	`varchar [ (n) ]`	Строка переменной длины
`character [ (n) ]`	`char [ (n) ]`	Строка фиксированной длины
`cidr`		Адрес сети IPv4 или IPv6
`circle`		Круг на плоскости
`date`		Календарная дата (год, месяц, день)
`double precision`	`float8`	Число с плавающей точкой двойной точности
`inet`		Адрес узла IPv4 или IPv6
`integer`	`int, int4`	Знаковое четырехбайтное целое
`interval [ fields ] [ (p) ]`		Промежуток времени
`json`		Данные JSON
`line`		Бесконечная линия на плоскости
`lseg`		Сегмент линии на плоскости
`macaddr`		MAC-адрес
`money`		Денежное значение (в валюте)
`numeric [ (p, s) ]`	`decimal [ (p, s) ]`	Числовое значение с заданной точностью

Окончание таблицы 3

Имя	Псевдонимы	Описание
path		Геометрический путь на плоскости (ломаная)
point		Геометрическая точка на плоскости
polygon		Закрытый геометрический путь на плоскости (полигон)
real	float4	Число с плавающей точкой одинарной точности
smallint	int2	Знаковое двухбайтное целое число
smallserial	serial2	Двухбайтное целое число с автоинкрементом
serial	serial4	Четырехбайтное целое число с автоинкрементом
text		Строка символов переменной длины
time [(p)] [without time zone]		Время дня
time [(p)] with time zone	timetz	Время дня, включая часовой пояс
timestamp [(p)] [without time zone]		Дата и время
timestamp [(p)] with time zone	timestamptz	Дата и время, включая часовой пояс
tsquery		Запрос текстового поиска
tsvector		Документ текстового поиска
txid_snapshot		Снимок ID-транзакции уровня пользователя
uuid		Универсальный уникальный идентификатор
xml		Данные XML

Примечание. В СУБД версии 9.6 могут быть использованы следующие типы:

- jsonb — бинарное представление данных JSON;
- pg_lsn — номер последовательностей лога PostgreSQL.

Примечание. Следующие типы (или вытекающие из них) регламентируются стандартом SQL: bigint, bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (с или без часового пояса), timestamp (с или без часового пояса), xml.

Каждый тип данных имеет внешнее представление, определяемое с помощью функций ввода и вывода. Многие из встроенных типов имеют понятные внешние форматы. Однако некоторые типы являются либо уникальными в PostgreSQL, такие как геометри-

ческие, либо имеют несколько возможных форматов, такие как типы даты и времени. Некоторые функции ввода и вывода являются необратимыми. К ним относятся функции, результат вывода которых может привести к потере точности по сравнению с введенным первоначальным значением.

5.1. Числовые типы

Числовые типы представлены двух-, четырех- и восьмибайтными целыми числами, четырех- и восьмибайтными числами с плавающей точкой и числами с заданной точностью (количеством знаков после запятой). В таблице 4 перечислены доступные числовые типы.

Таблица 4 – Числовые типы

Имя	Размер хранения, байты	Описание	Диапазон
`smallint`	2	Целые числа малого диапазона	От -32768 до +32767
`integer`	4	Обычные целые числа	От -2147483648 до +2147483647
`bigint`	8	Целые числа большого диапазона	От -9223372036854775808 до +9223372036854775807
`decimal`	Переменный	Числа с задаваемой точностью	до 131072 разрядов перед десятичной запятой; до 16383 разрядов после десятичной запятой
`numeric`	Переменный	Числа с задаваемой точностью	до 131072 разрядов перед десятичной запятой; до 16383 разрядов после десятичной запятой
`real`	4	Число с плавающей точкой одинарной точности	От 1E-37 до 1E+37 (точность 6 десятичных разрядов)
`double precision`	8	Число с плавающей точкой двойной точности	От 1E-307 до 1E+308 (точность 15 десятичных разрядов)
`smallserial`	2	Целое число малого диапазона с автоинкрементом	От 1 до 32767
`serial`	4	Целое число с автоинкрементом	От 1 до 2147483647
`bigserial`	8	Большое целое число с автоинкрементом	От 1 до 9223372036854775807

Синтаксис констант для числовых типов приведен в 1.1.2. Числовые типы имеют полный набор соответствующих арифметических операторов и функций (см. раздел 6.

5.1.1. Целочисленные типы

Типы `smallint`, `integer` и `bigint` хранят обыкновенные числа, т. е. числа без дробной части, но разных диапазонов. Попытки сохранить значения, которые выходят за

рамки разрешенного диапазона приведут к ошибке.

Тип integer предлагает баланс между диапазоном хранимых значений, размером и производительностью. Тип smallint обычно используется, когда необходимо рациональное использование дискового пространства. Тип bigint используется, если не хватает диапазона типа integer.

Спецификация SQL определяет только целочисленные типы integer (или int), smallint и bigint. Имена типов int2, int4 и int8 являются расширениями, которые работают и в некоторых других СУБД.

5.1.2. Числа с заданной точностью

Тип данных numeric может хранить числа с очень большим количеством разрядов, и с этим типом могут выполняться точные вычисления. Использование numeric особенно рекомендуется для хранения денежных значений и других величин, для которых требуется точность. Однако выполнение арифметических операций с ним осуществляется значительно медленнее по сравнению с целочисленными типами или типами с плавающей точкой.

При работе с числами с заданной точностью применяются следующие термины: масштаб (scale) — это количество десятичных разрядов в дробной части, справа от десятичной точки. Точность (precision) — это общее количество значимых разрядов во всем числе, т. е. количество разрядов по обе стороны от десятичной точки. Таким образом, число 23.5141 имеет точность 6 и масштаб 4. Целые числа могут быть представлены с использованием масштаба нуль.

Для столбца типа numeric существует возможность настроить и максимальную точность, и максимальный масштаб. Для определения столбца типа numeric используется синтаксис:

```
NUMERIC(precision, scale)
```

Точность должна быть положительным значением, масштаб должен быть либо положительным значением, либо нулем. Можно также использовать и такой синтаксис:

```
NUMERIC(precision)
```

который обусловливает значение масштаба нуль. Если задать NUMERIC без значений точности и масштаба, будет создан столбец, в котором можно хранить значения типа numeric с любыми точностью и масштабом, ограниченными только реализованным в СУБД пределом точности. Столбец такого типа будет приводить вводимые значения к любому отдельному масштабу, в то время как столбцы типа numeric с заданным значением масштаба будут приводить вводимые значениям к этому масштабу. (Стандарт SQL требует установки по умолчанию масштаба 0, т.е. приведение к целому числу. Для большей переносимости следует всегда явно указывать точность и масштаб).

П р и м е ч а н и е. Максимально разрешенная точность, которая явно задается при

объявлении данного типа — 1000; NUMERIC без указания точности говорит об ограничениях, описанных в таблице 4.

Если масштаб какого-либо значения больше, чем заявленный масштаб столбца, то система округлит это значение до указанного количества дробных разрядов. Затем, если количество разрядов слева от десятичной точки превышает заявленную точность минус заявленный масштаб, будет выдано сообщение об ошибке.

Значения numeric физически хранятся без каких-либо дополнительных нулей вначале или в конце. Следовательно, заявленная точность и масштаб столбца являются максимально возможными, но объем хранения для них выделяется не фиксированный. (В этом смысле тип numeric больше похож на тип varchar(n), чем на тип char(n).) Фактически требования, предъявляемые к хранению, – это два байта для каждой группы из четырех десятичных разрядов, плюс дополнительно от трех до восьми байт.

В дополнение к обычным числовым значениям, тип numeric позволяет хранить специальное значение NaN, которое означает «not-a-number» (не число). Любые операции над NaN в качестве результата дают другое значение NaN. Когда это значение пишется как константа в команде SQL, его следует заключить в одинарные кавычки, например:

```
UPDATE table SET x = 'NaN'
```

При вводе строка NaN распознается независимо от регистра букв.

Примечание. В большинстве реализаций концепции «не число», NaN не считается равным какому либо числовому значению (включая NaN). Чтобы разрешить сортировку значениям numeric и использовать индексы, основанные на деревьях, PostgreSQL считает значения NaN равными или больше, чем все не-NaN значения.

Типы decimal и numeric эквивалентны. Оба типа являются частью стандарта SQL.

5.1.3. Типы с плавающей точкой

Типы данных real и double precision являются типами с плавающей точкой (одинарной и двойной точности, соответственно). На практике, эти типы обычно реализуются по стандарту IEEE Standard 754 для Двоичной Арифметики с Плавающей Точкой (соответственно одинарной и двойной точности), согласно поддержке этих типов процессором, операционной системой и компилятором.

Слово «неточный» применительно к этим типам означает, что некоторые значения не могут быть точно преобразованы во внутренний формат этих типов и поэтому они хранятся как приблизительные значения. Таким образом, при попытке получения какого-либо значения после его сохранения возможно обнаружение незначительных отличий. Важно учитывать следующие моменты:

- Если требуется точное хранение и вычисление значений (таких как денежные величины), вместо данных типов следует использовать тип numeric;

- Следует с осторожностью реализовывать вычисления с этими типами для каких-либо важных целей, особенно в случаях возможных граничных моментах (бесконечность, переполнение);
- Сравнение двух значений с плавающей точкой на равенство может работать не так как ожидается.

На большинстве платформ, тип `real` имеет диапазон как минимум от 1E-37 до 1E+37 с точностью по меньшей мере в 6 десятичных разрядов. Тип `double precision` обычно имеет диапазон от 1E-307 до 1E+308 с точностью по меньшей мере в 15 разрядов. Значения, которые слишком велики или слишком малы приведут к ошибке. Если точность вводимых чисел слишком велика, возможно округление. Числа близкие к нулю, которые не могут быть представлены как отличные от нуля, приведут к ошибке переполнения.

Примечание. Конфигурационный параметр `extra_float_digits` задает количество дополнительных значащих знаков, включаемых при конвертировании в текст значения с плавающей запятой. При значении по умолчанию 0, выводимый текст будет одинаков на любой платформе, поддерживаемой PostgreSQL. Увеличение этого параметра более точно отображает сохраненное значение, но может быть непереносимым.

В дополнение к обычным числовым значениям, типы с плавающей точкой имеют некоторые специальные значения:

```
Infinity
-Infinity
NaN
```

Это соответствует специальным значениям стандарта IEEE 754 для «бесконечности», «минус бесконечности» и значению «не-число» соответственно. При записи этих значений как констант в командах SQL они заключаются в одинарные кавычки, например

```
UPDATE table SET x = 'Infinity'
```

При вводе эти строки распознаются независимо от регистра.

Примечание. IEEE754 указывает, что `NaN` не должен быть равен любым другим значениям с плавающей точкой (включая `NaN`). Чтобы разрешить сортировку значений с плавающей точкой и использовать индексы, основанные на деревьях, PostgreSQL считает значения `NaN` равными или больше, чем все не-`NaN`.

PostgreSQL также поддерживает стандартную нотацию `float` и `float(p)` для задания неточных числовых типов. Здесь `p` указывает минимально доступную точность в двоичных разрядах. PostgreSQL принимает значения от `float(1)` до `float(24)` как тип `real` и значения от `float(25)` до `float(53)` как `double precision`. Значения `p`, выходящие за этот допустимый диапазон, приведут к ошибке. Указание `float` без точности означает `double precision`.

5.1.4. Автоинкрементные типы

Типы данных smallserial, serial и bigserial являются нотацией для определения столбцов, которые играют роль уникального идентификатора (свойство, сходное с AUTO_INCREMENT, поддерживается и некоторыми другими СУБД). Конструкция:

```
CREATE TABLE tablename (
  colname SERIAL
);
```

эквивалента конструкции:

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename (
  colname integer NOT NULL DEFAULT nextval('tablename_colname_seq')
);
ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

Таким образом, создается столбец с типом integer и значением по умолчанию, получающимся с помощью генератора последовательности. Ограничение NOT NULL предназначено для того, чтобы исключить возможность появления в столбце значения NULL. (В большинстве случаев может возникнуть необходимость применить ограничения UNIQUE или PRIMARY KEY, чтобы предотвратить появление в столбце дублированных значений. Эти ограничения не устанавливаются автоматически.) Также последовательность отмечается как «принадлежащая» столбцу для ее удаления при удалении столбца таблицы.

Примечание. Поскольку smallserial, serial и bigserial реализованы с помощью генераторов, в последовательности значений столбца могут быть «дыры» или пропуски, даже если не производилось удаления строк. Значение, полученное с генератора, считается использованным, даже если строка с этим значением не была успешно добавлена, что возможно если транзакция, добавлявшая строку, была отменена. См. описание функции nextval() в 6.16.

Ограничения уникальности или первичного ключа для столбца с типом serial требуют явного указания, также как и для других типов данных.

Получение и вставка следующего значения последовательности в столбец с типом serial производится при указании использования значения по умолчанию для этого столбца. Это осуществляется либо пропуском столбца в списке столбцов в операторе INSERT, либо указанием ключевого слова DEFAULT.

Типы с именами serial и serial4 эквиваленты: оба создают столбцы типа integer. Точно также и типы с именами bigserial и serial8, за исключением того, что они создают столбцы типа bigint. Тип bigserial должен использоваться, если предполагается использование более чем 2^{31} идентификаторов в одной таблице. Типы с именами smallserial и serial2 действуют аналогично, но создают столбцы типа smallint.

Последовательность, создаваемая для столбца serial, автоматически удаляется, когда удаляется столбец, которому она принадлежит. Можно удалить эту последовательность без удаления столбца, но это приведет к принудительному удалению выражения по умолчанию для этого столбца.

5.2. Денежные типы

Тип money хранит значения валюты с фиксированной дробной частью (восьмибайтное число, диапазон от -92233720368547758.08 до +92233720368547758.07). Точность дробной части определяется настройкой СУБД lc_monetary. Ввод значений допускается в нескольких разных форматах, включая целые числа и числа с плавающей запятой, также как и значения в типичном для валюты формате, такие как '$1,000.00'. Вывод значений обычно осуществляется в последней форме, но зависит от региональных установок.

Поскольку вывод значений зависит от региональных установок, загрузка данных типа money из одной БД в другую может не работать, если БД имеют разные настройки lc_monetary. Чтобы избежать проблем, перед восстановлением необходимо убедиться, что значение настроек lc_monetary такое же или эквивалентно значению в той БД, где было выполнено сохранение.

Значения типов данных numeric, int и bigint могут быть приведены к типу money. Преобразование из типов данных real и double precision могут быть выполнены путем приведения вначале к типу numeric, например:

```
SELECT '12.34'::float8::numeric::money;
```

Однако, это не рекомендуется. Числа с плавающей запятой не должны использоваться для работы с деньгами из-за возможных ошибок, связанных с округлением.

Значение типа money может быть приведено к numeric без потери точности. Преобразование к другим типам может потенциально привести к потери точности и должно выполняться в два шага:

```
SELECT '52093.89'::money::numeric::float8;
```

Когда значение типа money делится на другое значение типа money, результатом является число двойной точности типа double precision (т.е. просто число, не денежное значение); валютные единицы при делении компенсируют друг друга.

5.3. Символьные типы

В таблице 5 представлены символьные типы общего назначения, доступные в PostgreSQL.

Таблица 5 – Символьные типы

Имя	Описание
`character varying(n), varchar(n)`	Строка переменной длины с ограничением
`character(n), char(n)`	Строка фиксированной длины с заполнением пробелами
`text`	Строка неограниченной длины

Стандарт SQL определяет два первичных символьных типа: `character varying(n)` и `character(n)`, где `n` является положительным целым числом. Оба эти типа могут хранить строки длиной до n символов. Попытка сохранить строку длиннее приведет к ошибке, если только избыточные символы не являются пробелами, в этом случае строка будет усечена до максимальной длины. Если сохраняемая строка короче, чем заданная длина, то значение типа `character` будет дополнено пробелами, а значение типа `character varying` будет сохранено как более короткая строка.

Если осуществляется явное приведение значения типа `character varying(n)` или `character(n)`, то значения с большей длиной будут усечены до n символов без возникновения ошибки.

Нотации `varchar(n)` и `char(n)` являются псевдонимами для `character varying(n)` и `character(n)`, соответственно. Указание `character` без длины эквивалентно `character(1)`. Если `character varying` используется без указания длины, то допускаются строки любой длины. Это является расширением PostgreSQL.

В дополнение, PostgreSQL предоставляет тип `text`, который хранит строки любой длины. Хотя тип `text` отсутствует в стандарте SQL, некоторые другие SQL СУБД также его поддерживают.

Значения типа `character` физически дополняются пробелами до заданной длины, хранятся и отображаются в этом виде. Однако эти дополнительные пробелы считаются семантически незначащими. При сравнении двух значений типа `character` конечные пробелы игнорируются и будут удалены при преобразовании значений типа `character` в любые другие строковые типы. Следует отметить, что конечные пробелы являются семантически значимыми в значениях `character varying` и `text`.

Для хранения короткой строки (до 126 байт) необходим один байт плюс сама строка, которая включает в случае типа `character` заполняющие пробелы. Более длинным строкам необходимо дополнительно четыре байта вместо одного. Длинные строки автоматически сжимаются системой, так что физически места на диске может потребоваться меньше. Очень длинные значения также хранятся во вспомогательных таблицах и не влияют на вы-

сокоскоростной доступ к столбцам с короткими значениями. Максимальный размер строки, которая может быть сохранена, составляет около 1 ГБ. (Максимальное значение, которое можно указать для n в описании типа будет меньше. В кодировках, которые работают с мультибайтными символами, количество символов и количество занимаемых ими байт может различаться, что делает сложным указание точного размера строки. При необходимости использования строки без ограничения длины рекомендуется применять тип text или тип character varying без указания длины, вместо того, чтобы точно указывать предельную длину.)

С точки зрения производительности, СУБД работает с этими тремя типами одинаково, за исключением увеличения размера хранения, при использовании типа с заполнением пробелами и несколькими дополнительными циклами для проверки длины при сохранении в столбцы ограниченной длины. Хотя в некоторых СУБД работа с типом character(n) более производительна, это не относится к PostgreSQL; фактически character(n) обычно самый медленный из трех, из-за дополнительных затрат на его хранение. В большинстве случаев вместо этого типа можно использовать типы text или character varying.

Информация о синтаксисе строковых литералов приведена в 1.1.2.1, а описание операторов и функций в разделе 6. Набор символов БД определяет тот набор символов, который используется для хранения текстовых значений; информация о поддержке наборов символов приведена в 12.3.

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');
ERROR:  value too long for type 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
(1)
```

Функция `char_length` описана в 6.4.

В PostgreSQL существует два других символьных типа с фиксированной длиной, приведенные в таблице 6.

Таблица 6 – Специальные символьные типы

Имя	Размер хранения, байты	Описание
`"char"`	1	Строка переменной длины с ограничением
`name`	256	Внутренний тип для имен объектов

Тип `name` существует только для хранения идентификаторов во внутренних системных каталогах. Его длина определена в 256 байт (255 обычных символа плюс символ окончания строки), но фактически указывается с помощью использования константы `NAMEDATALEN`. Эта длина устанавливается в момент компиляции (и может меняться). Тип `"char"` (в кавычках) отличается от `char(1)` тем, что для хранения его значений используется только один байт. Этот тип предназначен для внутреннего использования в системных каталогах в качестве типа для небольших перечислений.

5.4. Двоичные типы данных

Тип `bytea`, приведенный в таблице 7, предназначен для хранения двоичных строк.

Таблица 7 – Двоичные типы данных

Имя	Размер хранения	Описание
`bytea`	1 или 4 байта плюс сама двоичная строка	Двоичная строка переменной длины

Двоичная строка — это последовательность байт (или октетов). Двоичные строки отличаются от символьных строк двумя характеристиками. Во-первых, двоичные строки позволяют хранить байтовое представление таких символов, как: нуль и других «непечатаемых» символов (символы, представленные кодами в диапазоне от 32 до 126). Символьные строки не позволяют использовать символ нуль, а также любые другие значения и последовательности байт, недопустимые той кодировкой, в которой работает СУБД. Во-вторых, операции с двоичными строками осуществляются с байтами, в то время как операции с символьными строками зависят от региональных установок.

Тип `bytea` поддерживает два внешних формата для ввода и вывода: исторический (экранированный) «escape» формат PostgreSQL и (шестнадцатеричный) «hex» формат. Ввод

данных возможен в любом из них. Формат вывода зависит от конфигурационного параметра `bytea_output` (по умолчанию установлен в «hex»).

Примечание. Формат «hex» был введен в PostgreSQL 9.0; ранние версии и некоторые инструменты этот формат не поддерживают.

Стандарт SQL описывает другие двоичные строковые типы, называемые `BLOB` или `BINARY LARGE OBJECT`. Их формат ввода отличается от `bytea`, но предоставляемые функции и операторы в основном те же.

5.4.1. «hex» формат bytea

Шестнадцатеричный («hex») формат кодирует двоичные данные как два шестнадцатеричных разряда на байт, старший разряд идет первым. Для отличия от «escape» формата, строка предваряется последовательностью `\x`. Требуется экранировать начальную обратную косую черту, дублируя ее, в тех же случаях, в которых требуется двойная обратная косая черта в «escape» формате (см. ниже). Шестнадцатеричные разряды могут быть представлены как в верхнем, так и в нижнем регистре, между парами цифр допускаются пробелы (но не внутри пары разрядов и не в стартовой последовательности `\x`). Шестнадцатеричный формат совместим с широким диапазоном внешних приложений и протоколов, а также он способствует более быстрому преобразованию данных, чем в случае «escape» формата, так что его использование является предпочтительным.

Пример

```
SELECT E'\\xDEADBEEF';
```

5.4.2. «escape» формат bytea

«Escape» формат является традиционным форматом для типа `bytea` в PostgreSQL. Он использует представление двоичной строки как последовательности ASCII символов, где во время конвертации байты, которые не могут быть представлены как ASCII символы, заменяются специальными «escape» последовательностями. Если, с точки зрения приложения, представление байтов как символов имеет смысл, то данное представление может быть подходящим. Но на практике это обычно запутывает, поскольку затрудняет понимание разницы между двоичными строками и символьными строками, к тому же используется особый механизм экранирования, который является несколько громоздким. Таким образом, в большинстве новых приложений, использование данного формата не рекомендуется.

При вводе значений типа `bytea` в «escape» формате байты определенных значений, которые используются как части строчных литералов в каких-либо операторах SQL, должны быть экранированы (все байтовые значения могут быть экранированы). В общем случае, чтобы экранировать вводимое значение, оно преобразуется в трехциферное восьмеричное значение, эквивалентное его десятичному значению, и предваряется символами `\\`. В

таблице 8 приведены символы, которые должны быть экранированы, и альтернативные escape-последовательности там, где это возможно.

Таблица 8 – Экранируемые литеральные значения

Десятичное значение	Описание	Экранирование	Пример	Представление на выходе
0	Нуль	E'\\000'	SELECT E'\\000'::bytea;	\000
39	Одиночная кавычка	"" или E'\\047'	SELECT E'\"::bytea;	'
92	Обратная косая черта	E'\\\\' или E'\\134'	SELECT E'\\\\'::bytea;	\\
От 0 до 31 и от 127 до 255	Непечатаемый символ	E'\\xxx' (восьмеричное значение)	SELECT E'\\001'::bytea;	\001

Требование об экранировании непечатаемых символов варьируется в зависимости от региональных установок. Результатом в каждом примере является один байт, даже если представление на выходе составляет более одного символа.

Причина, по которой используется два символа \ в таблице 8, заключается в том, что вводимый строковый литерал должен пройти на сервере PostgreSQL через две фазы обработки. Первая \ каждой пары интерпретируется для обработчика строкового литерала (включая использование синтаксиса экранирования) как символ экранирования и, таким образом, опускается, оставляя второй символ \ в паре (чтобы избежать этого уровня экранирования, можно использовать строки «доллар–кавычка»). Оставшийся символ \ затем распознается функцией ввода типа bytea, как начинающий трехциферное восьмеричное значение или символ экранирования следующего символа \. Например, строковый литерал, передаваемый на сервер как 'E\\001', принимает вид \001 после обработчика строкового литерала. Далее \001 посылается функции ввода типа bytea, где преобразовывается в одиночный байт с десятичным значением 1. Символ одиночной кавычки для bytea не считается специальным, т. к. он отвечает обычным правилам строковых литералов (см. 1.1.2.1).

Значения bytea при выводе также экранируются. Обычно в виде эквивалентного трехциферного восьмеричного значения, предваряемого символом \. Большинство печатаемых байт выводится в их стандартном представлении, согласно кодировке клиента. Символ с десятичным значением 92 (\) выводится в специальной альтернативной форме. Подробности приведены в таблице 9.

Таблица 9 – Экранированные значения на выводе

Десятичное значение	Описание	Экранирование	Пример	Результат вывода
92	Обратная косая черта	\\	SELECT E'\\134'::bytea;	\\
От 0 до 31 и от 127 до 255	Непечатаемые значения	\xxx (восьмеричное значение)	SELECT E'\\001'::bytea;	\001
От 32 до 126	Печатаемые значения	Представление по кодировке клиента	SELECT E'\\176'::bytea;	~

В зависимости от программы, используемой для работы с PostgreSQL, могут быть доступны дополнительные инструменты для работы с экранированными и неэкранированными строками типа bytea. Например, роль экранирующего символа может также играть символ перевода строки и возврата каретки, если в таковые их автоматически будет преобразовывать интерфейс программы.

5.5. Типы даты/времени

PostgreSQL поддерживает набор SQL-типов даты и времени, представленных в таблице 10. Операции, которые доступны для этих типов данных, описываются в 6.9.

Таблица 10 – Типы даты/времени

Имя	Размер хранения, байты	Описание	Наименьшее значение	Наибольшее значение	Разрешение
timestamp [(p)] [without time zone]	8	Дата и время	4713 BC	294276 AD	1 мкс / 14 знаков
timestamp [(p)] with time zone	8	Дата и время с часовым поясом	4713 BC	294276 AD	1 мкс / 14 знаков
date	4	Только дата	4713 BC	5874897 AD	1 день
time [(p)] [without time zone]	8	Только время	00:00:00	24:00:00	1 мкс / 14 знаков
time [(p)] with time zone	12	Время с часовым поясом	00:00:00+1459	24:00:00-1459	1 мкс / 14 знаков
interval [fields] [(p)]	12	Время в интервалах	-178000000 лет	178000000 лет	1 мкс / 14 знаков

Примечание. Стандарт SQL требует, чтобы написание просто timestamp было эквивалентно timestamp without time zone и PostgreSQL следует стандарту. Дополнительно PostgreSQL допускает использование сокращение timestamptz для timestamp with time zone.

Для типов time, timestamp и interval можно указывать необязательное значение точности p, которое задает количество дробных разрядов в поле секунд. По умолчанию точность в явном виде не указывается. Допустимый диапазон значения p от 0 до 6 для типа timestamp и для типа interval.

Примечание. Если значения timestamp хранятся как восьмибайтные целые числа (по умолчанию), то в пределах всего диапазона значений достигается точность в одну микросекунду. Если значения timestamp хранятся как числа с плавающей запятой двойной точности (в случае выбора устаревшей опции при компиляции), то эффективное ограничение на точность может быть меньше 6. Значения timestamp хранятся как количество секунд перед или после 2000-01-01. В случае когда значения timestamp реализуются с использованием чисел с плавающей точкой, микросекундная точность достигается для дат, которые находятся внутри отрезка, начинающегося с 2000-01-01, но эта точность снижается для дат вне этого отрезка. Значения timestamp, использующие плавающую точку, позволяют работать с большим диапазоном значений, чем указано выше: от 4713 BC до 5874897 AD. Та же опция компиляции определяет, будут ли значения time interval храниться как числа с плавающей точкой или как восьмибайтные целые числа. В случае чисел с плавающей точкой, большие значения interval имеют меньшую точность из-за увеличения размера самого интервала.

Для типов time разрешается диапазон точности p от 0 до 6, когда для хранения значений используется восьмибайтное целое, или от 0 до 10, когда используется число с плавающей точкой.

Тип interval имеет дополнительную опцию, которая должна ограничивать набор сохраняемых полей с помощью одной из следующих фраз:

```
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

Если указаны и поля, и точность, то поля должны включать SECOND, т. к. точность применима только к секундам.

Тип time with time zone определяется стандартом SQL, но в большинстве случаев, комбинация типов date, time, timestamp without time zone и timestamp with time zone предоставляет полный диапазон функциональности для работы с датой и временем, который бы мог потребоваться какому-либо приложению.

Типы abstime и reltime имеют низкую точность и используются для внутренних целей.

5.5.1. Ввод даты/времени

Вводимые значения даты и времени принимаются в любом допустимом формате, включая ISO 8601, SQL-совместимый, традиционный формат Postgres и другие. Для некоторых форматов порядок следования полей месяца, дня и года в значении даты является неоднозначным и для таких форматов поддерживается указание порядка следования этих полей. Установкой параметра DateStyle в значение MDY можно выбрать порядок следования месяц-день-год, в значение DMY — день-месяц-год и в YMD — год-месяц-день.

В PostgreSQL управление вводом значений даты и времени устроено более гибко, чем этого требует стандарт SQL.

Необходимо помнить, что любые вводимые значения даты или времени требуется заключать в одинарные кавычки, как и текстовые строки (см. 1.1.2.7). Стандарт SQL требует следующего синтаксиса:

```
type [ (p) ] 'value'
```

где p является целым числом, соответствующим количеству дробных разрядов во втором поле. Точность может быть задана для типов time, timestamp и interval. Если точность в спецификации константы не указана, то она устанавливается в значение по умолчанию для литеральных значений.

5.5.1.1. Даты

В таблице 11 приведены некоторые формы ввода для типа date.

Таблица 11 – Ввод значений даты

Значение	Описание
1999-01-08	ISO 8601; 8 января в любом режиме (рекомендуемый формат)
January 8, 1999	Значение является однозначным для любого datestyle режима ввода
1/8/1999	8 января в режиме MDY; 1 августа в режиме DMY

Окончание таблицы 11

Значение	Описание
1/18/1999	18 января в режиме MDY; в других режимах значение будет отвергнуто
01/02/03	2 января 2003 года в режиме MDY; 1 февраля 2003 года в режиме DMY; 3 февраля 2001 года в режиме YMD
1999-Jan-08	8 января в любом режиме
Jan-08-1999	
08-Jan-1999	
99-Jan-08	8 января в режиме YMD, иначе ошибка
08-Jan-99	8 января, за исключением режима YMD, в котором будет ошибка
Jan-08-99	
19990108	ISO 8601; 8 января 1999 года в любом режиме
990108	
1999.008	Год и номер дня в году
J2451187	День по Юлианскому календарю
January 8, 99 BC	99 год до н. э.

5.5.1.2. Время

Типами времени являются:

```
time [ (p) ] without time zone
time [ (p) ] with time zone
```

Тип time эквивалентен типу time without time zone.

Правильные вводимые значения для этих типов состоят из времени дня, за которым следует необязательное значение часового пояса (таблицы 12 и 13).Если часовой пояс задается при вводе значений типа time without time zone, то он игнорируется. Также всегда возможно задать дату, но она будет проигнорирована, за исключением случая, когда используется имя часового пояса, который реализует переход на летнее/зимнее время, например America/New_York. В этом случае требуется указание даты для корректного применения летнего или зимнего времени. Соответствующее смещение часового пояса записывается в значение time with time zone.

Таблица 12 – Ввод значений времени

Значение	Описание
04:05:06.789	ISO 8601
04:05:06	
04:05	
040506	
04:05 AM	Тоже, что и 04:05; AM не имеет значения

Окончание таблицы 12

Значение	Описание
04:05 PM	Тоже, что и 16:05; значение часа должно быть <= 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	
04:05-08:00	
040506-08	
04:05:06 PST	Часовой пояс, заданный аббревиатурой
2003-04-12 04:05:06 America/New_York	Часовой пояс, заданный полным именем

Таблица 13 – Ввод значений часового пояса

Значение	Описание
PST	Аббревиатура (Pacific Standard Time)
America/New_York	Полное имя часового пояса
PST8PDT	Спецификация часового пояса в стиле POSIX
-8:00	ISO 8601 смещение для PST
-800	
-8	
zulu	Аббревиатура для UTC
z	Краткая форма zulu

5.5.1.3. Дата и время

Правильные вводимые значения для типа даты и времени состоят из объединенного значения даты и времени, за которым следует необязательное значение часового пояса, за которым следует необязательный суффикс AD или BC. (В качестве альтернативы AD/BC могут указываться перед часовым поясом, но так делать не рекомендуется.) Таким образом:

```
1999-01-08 04:05:06
```

и:

```
1999-01-08 04:05:06 -8:00
```

являются правильными значениями, которые соответствуют стандарту ISO 8601. Также поддерживается расширенный формат стандарта:

```
January 8 04:05:06 1999 PST
```

Стандарт SQL различает литералы типов timestamp without time zone и timestamp with time zone по наличию «+» или «-». Следовательно, значение

```
TIMESTAMP '2004-10-19 10:23:54'
```

имеет тип timestamp without time zone, в то время как значение:

```
TIMESTAMP '2004-10-19 10:23:54+02'
```

имеет тип timestamp with time zone. PostgreSQL никогда не проверяет содержимое строки литерала перед тем, как определить его тип и, таким образом, будет считать оба данных выше примера как timestamp without time zone. Для распознавания литерала как timestamp with time zone необходимо явно задавать правильный для этого типа вид:

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

В литерале, который однозначно будет иметь тип timestamp without timezone, PostgreSQL игнорирует любые указания часового пояса. Таким образом, итоговое значение будет получено из полей даты/времени, которые присутствуют во вводимом значении, и не будет скорректировано по часовому поясу.

Для типа timestamp with time zone значение всегда хранится в UTC. Вводимое значение, которое имеет явно заданный часовой пояс, конвертируется в UTC, используя смещение, соответствующее этому часовому поясу. Если при вводе часовой пояс не указан, то он берется из системного параметра TimeZone и точно также конвертируется в UTC, используя смещение часового пояса, указанное в переменной TimeZone.

Когда выводится значение типа timestamp with time zone, оно всегда конвертируется из UTC в часовой пояс, заданный переменной TimeZone, и отображается как местное время в этом поясе. Чтобы увидеть это же время в другом часовом поясе, требуется или изменить значение переменной TimeZone, или использовать конструкцию AT TIME ZONE (см. 6.9.3).

Преобразования между типами timestamp without time zone и timestamp with time zone обычно означают, что значение timestamp without time zone должно быть принято или выдано как значение местного времени часового пояса, указанного в переменной TimeZone. Ссылка на другой часовой пояс при таком преобразовании может быть указана с помощью конструкции AT TIME ZONE.

5.5.1.4. Специальные значения

PostgreSQL поддерживает дополнительно еще несколько специальных значений ввода даты/времени, показанных в таблице 14.

Таблица 14 – Специальные значения ввода даты и времени

Значение	Типы данных	Описание
epoch	date, timestamp	1970-01-01 00:00:00+00 (нулевое системное время Unix)
infinity	date, timestamp	Позже, чем все другие временные штампы
-infinity	date, timestamp	Раньше, чем все другие временные штампы
now	date, time, timestamp	Время начала текущей транзакции
today	date, timestamp	Полночь текущего дня

Окончание таблицы 14

Значение	Типы данных	Описание
tomorrow	date, timestamp	Полночь завтрашнего дня
yesterday	date, timestamp	Полночь вчерашнего дня
allballs	time	00:00:00.00 UTC

Из этих значений infinity и -infinity специально представлены внутри СУБД и отображаются в таком же виде; другие же являются просто сокращениями, которые будут преобразованы при вводе в обычные значения даты/времени. (Например, в момент обработки now и подобные строки преобразовываются в определенное значение времени.) Все такие значения, когда они используются в командах SQL, должны записываться в одинарных кавычках.

Также получить текущее значение времени для соответствующих типов данных можно используя следующие, совместимые со стандартом SQL функции: CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME, LOCALTIMESTAMP. Последние четыре функции позволяют задавать необязательную точность (см. 6.9.4.) Указанные ключевые слова являются функциями SQL и не распознаются как строки данных.

5.5.2. Вывод даты/времени

Формат вывода значений типов даты/времени может быть установлен в один из четырех стилей: ISO 8601, SQL (Ingres), традиционный Postgres (Unix date) и German, с помощью команды SET datestyle. По умолчанию используется формат ISO. (Стандарт SQL требует использования формата ISO 8601. Название формата вывода «SQL» является исторически сложившимся.) Таблица 15 показывает примеры каждого стиля вывода. Вывод значений типов date и time осуществляется в соответствии с приведенными примерами только как часть даты и времени.

Таблица 15 – Стили вывода даты и времени

Стиль	Описание	Пример
ISO	Стандарт ISO 8601/SQL	1997-12-17 07:37:16-08
SQL	Традиционный стиль	12/17/1997 07:37:16.00 PST
Postgres	Собственный стиль	Wed Dec 17 07:37:16 1997 PST
German	Региональный стиль	17.12.1997 07:37:16.00 PST

Примечание. Стандарт ISO 8601 регламентирует использование символа T в верхнем регистре для разделения даты и времени. PostgreSQL позволяет вводить значения в таком формате, но при выводе вместо T использует пробел для лучшей читабельности и соответствию RFC 3339 и некоторым другим СУБД.

В стилях SQL и Postgres день идет перед месяцем, если был задан порядок полей DMY, в противном случае месяц следует перед днем. В таблице 16 приведен пример.

Таблица 16 – Соглашения по датам

Установка стиля	Формат	Пример
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

Пользователь может выбрать нужные ему стили даты/времени с помощью команды SET datestyle, параметра DateStyle в конфигурационном файле postgresql.conf или с установкой переменной окружения PGDATESTYLE на сервере или клиенте.

Функция форматирования to_char (см. 6.8) также может быть использована как более гибкий способ форматирования выводимых значений даты/времени.

5.5.3. Часовые пояса

PostgreSQL использует БД часовых поясов zoneinfo для получения информации об исторических правилах часовых поясов. Для времени в будущем берутся последние известные правила для данного часового пояса.

Для избежания неоднозначности при использовании часовых поясов следует применять типы даты/времени, которые содержат как дату, так и время. Не рекомендуется использовать тип time with time zone (хотя он поддерживается PostgreSQL для старых приложений и для совместимости со стандартом SQL). PostgreSQL для всех типов, которые содержат только дату или только время, использует локальный часовой пояс.

Все даты, учитывающие часовой пояс и время внутри системы, хранятся приведенными к UTC. Перед тем как значения будут выданы клиенту, они преобразуются в локальное время в соответствии с настройками, указанными в параметре TimeZone.

PostgreSQL позволяет указывать часовые пояса в трех различных формах:

- полное имя часового пояса, например America/New_York. Распознаваемые имена часовых поясов перечислены в pg_timezone_names. PostgreSQL использует для этой цели данные о часовых поясах zoneinfo, так что одни и те же имена распознаются и другими программами;
- аббревиатура имени часового пояса, например PST. Такие аббревиатуры часто задают конкретное смещение от UTC, в отличие от полных имен часовых поясов, которые могут неявно устанавливать также и правила перехода на летнее/зимнее время. Распознаваемые аббревиатуры перечислены в pg_timezone_abbrevs .

Аббревиатура часового пояса не может использоваться при установке конфигурационных параметров TimeZone или log_timezone. Аббревиатуры могут быть использованы при вводе значений даты/времени и с оператором AT TIME ZONE;

- в дополнение к именам часовых поясов и их аббревиатурам, PostgreSQL также может работать и со спецификациями часовых поясов в стиле POSIX в форме STDoffset или STDoffsetDST, где STD — аббревиатура часового пояса, offset — числовое смещение в часах на запад от UTC и DST — необязательная аббревиатура летнего/зимнего времени, установленная на одночасовое опережение заданного смещения. Например, если значение EST5EDT не является уже распознанным именем часового пояса, оно будет принято и будет функционально эквивалентно времени восточного побережья США. При вводе имени часового пояса с переходом на летнее/зимнее время работа с ним будет осуществляться в соответствии с такими же правилами перехода на летнее/зимнее время, которые используются в БД zoneinfo в записи posixrules. При стандартной установке PostgreSQL posixrules — это тоже самое, что и US/Eastern, так что спецификации часовых поясов в стиле POSIX следуют правилам перехода на летнее/зимнее время, применяемым в США. При необходимости это можно изменить заменой файла posixrules.

Есть концептуальное и практическое отличие между аббревиатурами и полными именами: аббревиатуры всегда предоставляют постоянное смещение от UTC, в то время как полные имена подразумевают локальное правило перехода на летнее/зимнее время и, таким образом, имеют два возможных смещения от UTC.

Использование часовых поясов в стиле POSIX требует осторожности, поскольку данная возможность позволяет задавать некорректные значения, которые будут приняты без сообщений об ошибках, т. к. не производится проверка аббревиатур. Например, SET TIMEZONE TO FOOBAR0 будет работать, но фактически система будет продолжать использовать данное значение как аббревиатуру для UTC. Кроме того, необходимо помнить, что в именах часовых поясов POSIX положительные смещения используются для поясов к западу от Гринвича. Во всем остальном PostgreSQL следует соглашению ISO 8601: положительные смещения часовых поясов используются для поясов к востоку от Гринвича.

Имена часовых поясов распознаются независимо от регистра букв.

Ни полные имена, ни аббревиатуры не являются жестко встроенными в сервер; они берутся из конфигурационных файлов, хранящихся в подкаталогах .../share/timezone/ и .../share/timezonesets/ относительно каталога установки.

Конфигурационный параметр TimeZone может быть установлен в файле postgresql.conf или любым другим стандартным способом, описанным в разделе 8. Также существует несколько специальных способов его установки:

1) SQL команда SET TIME ZONE устанавливает часовой пояс для текущей сессии. Эта команда является альтернативой команды SET TIMEZONE TO, которая имеет более соответствующий стандарту SQL синтаксис;

2) Переменная окружения PGTZ применяется для неявного выполнения команды SET TIME ZONE приложениями, которые используют библиотеку libpq.

5.5.4. Ввод значений интервала

Значения типа interval могу быть записаны следующим образом:

```
[@] quantity unit [quantity unit...] [direction]
```

где quantity (длительность) является числом (возможно со знаком); unit (элемент) может принимать значения microsecond (микросекунда), millisecond (миллисекунда), second (секунда), minute (минута), hour (час), day (день), week (неделя), month (месяц), year (год), decade (декада), century (век), millennium (тысячелетие) или аббревиатуры одного из этих значений; direction (направление) может принимать значения ago (назад) или не указываться. Знак (@) также является необязательным. В соответствии с указанным знаком будет добавлено необходимое количество различных элементов. ago изменяет знак у всех полей. Указанный синтаксис также используется при выводе значений типа interval, если переменная IntervalStyle установлена в postgres_verbose.

Длительность в днях, часах, минутах и секундах может быть указана без явных указаний соответствующих элементов. Например, '1 12:59:10' читается также как и '1 day 12 hours 59 min 10 sec'. Комбинация лет и месяцев может быть также задана с использованием знака черточки; например, '200-10' читается также как и '200 years 10 months'. (Эти укороченные формы фактически являются единственными из тех, что разрешает стандарт SQL и используются для вывода, когда значение IntervalStyle установлено в sql_standard.)

Значения типа interval могут также записываться по стандарту ISO 8601, используя либо формат с указателями, либо альтернативный формат.

Формат с указателями выглядит так:

```
P quantity unit [ quantity unit ...] [ T [ quantity unit ...]]
```

Строка должна начинаться с P и может включать T, которые указывают на элементы времени дня. Доступные аббревиатуры элементов приведены в таблице 17. Элементы могут быть опущены и могут быть указаны в любом порядке, но элементы меньшие, чем день должны указываться после T. В особенности это касается M, смысл которого зависит от того, будет ли он указан до или после T.

Таблица 17 – Аббревиатуры элементов интервалов по стандарту ISO 8601

Аббревиатура	Пояснение
Y	Годы
M	Месяцы (в части, касающейся даты)
W	Недели
D	Дни
H	Часы
M	Минуты (в части, касающейся времени)
S	Секунды

В альтернативном формате:

```
P [ years-months-days ] [ T hours:minutes:seconds ]
```

строка должна начинаться с P, а T отделяет в интервале дату от времени. Значения задаются цифрами.

При записи констант-интервалов по спецификации с полями или при задании значения столбца интервала, который создан по спецификации с полями, интерпретация длительностей без явного указания элемента зависит от ранее указанных в спецификации полей. Например, `INTERVAL '1' YEAR` читается как 1 год, в то время как `INTERVAL '1'` означает 1 секунду. Кроме того, значения полей «справа» от наименее значимого поля, разрешенного спецификацией полей, молча отбрасываются. Например, в результате написания `INTERVAL '1 day 2:03:04' HOUR TO MINUTE` будет отброшено поле секунд, но не поле дня.

В соответствии со стандартом SQL все поля в значении интервала должны иметь один и тот же знак, так что начальный знак «минус» применяется ко всем полям; например знак «минус» в значении `'-1 2:03:04'` применяется и к дням, и к часам/минутам/секундам. PostgreSQL позволяет полям иметь разные знаки и считает каждое поле в текстовом представлении как независящее от знака, так что часть «час/минута/секунда» рассматривается в данном выше примере как положительная. Если значение переменной `IntervalStyle` установлено в `sql_standard`, то начальный знак применяется ко всем полям (а не только к тем, где нет дополнительного знака). В противном случае, используется традиционная для PostgreSQL интерпретация. Чтобы избежать путаницы, рекомендуется явно указывать знак для каждого поля, если какое-либо поле является отрицательным.

Внутри значения `interval` хранятся как месяцы, дни и секунды. Так сделано, потому что количество дней в месяце разное, а продолжительность дня может быть 23 или 25 часов, если используется переход на летнее/зимнее время. Поля «месяцы» и «дни» являются целыми, а поле «секунды» может храниться в виде дроби. Поскольку интервалы

обычно создаются из строковых констант или как разность значений timestamp, данный метод хранения хорошо работает в большинстве случаев. Для коррекции дней и часов, которые перекрывают их нормальные диапазоны, доступны функции justify_days и justify_hours.

В подробном формате ввода и в некоторых полях компактных форматов значения могут быть указаны в виде дробей; например '1.5 week' или '01:02:03.45'. Такой ввод для хранения преобразуется в соответствующее количество месяцев, дней и секунд. Преобразование выполняется по следующим правилам: 1 месяц = 30 дням, 1 день = 24 часам. Например, '1.5 month' будет 1 месяц и 15 дней. Только секунды будут показываться при выводе в виде дроби.

В таблице 18 приведены примеры допустимого ввода значений interval.

Таблица 18 – Ввод значений интервала

Пример	Описание
1-2	Формат по стандарту SQL: 1 год 2 месяца
3 4:05:06	Формат по стандарту 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 указателями ISO 8601: то же значение, что и выше
P0001-02-03T04:05:06	Альтернативный формат ISO 8601: то же значение, что и выше

5.5.5. Вывод значений интервала

Формат вывода значений типа interval может быть одним из четырех стилей: sql_standard, postgres, postgres_verbose или iso_8601, в зависимости от использования команды SET intervalstyle. По умолчанию установлен формат postgres. В таблице 19 приведены примеры каждого стиля вывода.

Таблица 19 – Вывод значений интервала

Спецификация стиля	Интервал (год–месяц)	Интервал (день–время)	Смешанный интервал
sql_standard	1-2	3 4:05:06	-1-2 +3 -4:05:06
postgres	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
postgres_verbose	@ 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
iso_8601	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

Стиль sql_standard выводит строковые значения, соответствующие стандарту

SQL, если значения интервала соответствуют ограничениям стандарта (либо только год–месяц, либо только день–время, без смешивания положительных и отрицательных компонентов). В противном случае, вывод выглядит как стандартный литерал «год–месяц», за которым следует литерал «день–время» с явным указанием знаков, добавляемых во избежание путницы на смешанных по знакам интервалах.

Вывод стиля `postgres` совпадает с выводом, который был в PostgreSQL до версии 8.4, при установке параметра `DateStyle` в `ISO`.

Вывод стиля `postgres_verbose` совпадает с выводом PostgreSQL до версии 8.4, при установке параметра `DateStyle` в `не-ISO`.

Вывод стиля `iso_8601` совпадает с форматом с указателями.

5.6. Логический тип

PostgreSQL предоставляет стандартный SQL-тип `boolean`, для которого принято два значения: `true` (истина) или `false` (ложь). Третье состояние `unknown` (неизвестно) представляется SQL значением `NULL`.

Таблица 20 – Логический тип

Имя	Размер хранения	Описание
`boolean`	1 байт	`true` (истина) или `false` (ложь)

Допустимы следующие литеральные значения для `true`:

```
TRUE
't'
'true'
'y'
'yes'
'on'
'1'
```

Для значения `false` могут быть использованы такие значения, как:

```
FALSE
'f'
'false'
'n'
'no'
'off'
'0'
```

Пробелы в начале и в конце игнорируются, регистр букв значения не имеет. Предпочтительным (и соответствующим стандарту SQL) является использование ключевых слов

TRUE и FALSE.

Пример

```
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
```

В примере видно, что значения boolean выводятся с использованием букв t и f.

5.7. Перечисления

Перечисления (enum) — типы данных, которые включают в себя статические, предопределенные списки упорядоченных значений. Они эквивалентны типам enum в некоторых языках программирования. В качестве примера типа enum можно привести дни недели или список состояний для каких-либо данных.

5.7.1. Объявление перечислений

Типы enum создаются с помощью команды CREATE TYPE, например:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Однажды созданный тип enum может использоваться в таблицах и при определении функций точно также, как и другие типы:

Пример

```
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)
```

5.7.2. Порядок значений

Порядок значений в типе `enum` точно указывается при объявлении типа. Для перечислений поддерживаются все стандартные операции сравнения и агрегатные функции.

Пример

```
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)
```

5.7.3. Безопасность типа

Перечисление является полностью отдельным типом данных и не может быть неявно приведено или сравнимо с другими.

Пример

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (
  num_weeks int,
  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:  недопустимое значение для enum happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood = holidays.happiness;
ОШИБКА:  не найден оператор: mood = happiness
```

Если требуется обеспечить приведение типа или сравнение, необходимо либо написать специальный оператор, либо использовать явное приведение типа.

Пример

Сравнение разных перечислений, путем приведения к тексту:

```
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood::text = holidays.happiness::text;
 name | num_weeks
------+-----------
 Moe  |          4
(1 row)
```

5.7.4. Детали реализации

Тип `enum` занимает четыре байта на диске. Длина значения (текстовой метки) ограничена настройкой `NAMEDATALEN`.

Значения перечислений зависят от регистра букв, так что `'happy'` не совпадает по значению с `'HAPPY'`. Пробелы в значениях также учитываются.

Соответствие внутренних значений перечисления текстовым меткам хранится в системном каталоге `enum`.

5.8. Геометрические типы

Геометрические типы данных представлены двухмерными пространственными объектами. Наиболее фундаментальный тип — точка, является основой для всех остальных типов. В таблице 21 приведены доступные в PostgreSQL геометрические типы данных.

Таблица 21 – Геометрические типы

Имя	Размер хранения, байты	Описание	Представление
`point`	16	Точка на плоскости	`(x,y)`
`line`	32	Бесконечная линия	`((x1,y1),(x2,y2))`

Окончание таблицы 21

Имя	Размер хранения, байты	Описание	Представление
lseg	32	Конечный сегмент линии (отрезок)	((x1,y1),(x2,y2))
box	32	Четырехугольник	((x1,y1),(x2,y2))
path	16+16n	Закрытый путь (аналог полигона)	((x1,y1),...)
path	16+16n	Открытый путь	[(x1,y1),...]
polygon	40+16n	Полигон (аналог закрытого пути)	((x1,y1),...)
circle	24	Круг	<(x,y),r>

Для выполнения различных геометрических операций, таких как масштабирование, преобразование, поворот и определение пересечений, доступен набор функций и операторов, описанных в 6.11.

5.8.1. Точки

Точки являются фундаментальными двухмерными строительными блоками для геометрических типов. Значения типа point задаются с помощью следующего синтаксиса:

```
( x , y )
  x , y
```

где x и y соответствующие координаты в форме чисел с плавающей точкой.

При выводе точек используется первый вариант синтаксиса.

5.8.2. Линии

Примечание. Данный тип может быть полноценно использован только в СУБД версии 9.6.

Линии представляют собой коэффициенты линейного уравнения Ax + By + C = 0, где коэффициенты A и B не равны одновременно нулю. Для ввода/вывода используется следующий синтаксис:

```
{ A, B, C }
```

В качестве альтернативы может быть использована одна из следующих форм записи:

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

где (x1,y1) и (x2,y2) — две точки линии.

5.8.3. Сегменты линий

Сегменты линии представляются парой точек. Значения типа lseg задаются с помощью следующего синтаксиса:

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
```

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

где `(x1,y1)` и `(x2,y2)` являются конечными точками сегмента линии.

При выводе сегментов линий используется первый вариант синтаксиса.

5.8.4. Четырехугольники

Боксы представляются парой точек, которые являются противоположными углами бокса. Значения типа `box` задаются с помощью следующего синтаксиса:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

где `(x1,y1)` и `(x2,y2)` являются любыми двумя противоположными углами бокса.

При выводе значений типа `box` используется второй вариант синтаксиса.

Порядок углов при вводе корректируется так, чтобы хранить сперва верхний правый угол, а затем левый нижний угол.

5.8.5. Пути

Пути представляются списками входящих в них точек. Пути могут быть открытыми, где первая и последняя точки списка не соединяются, или закрытыми, где первая и последняя точки соединены. Значения типа `path` задаются с помощью следующего синтаксиса:

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

где точки являются конечными точками сегментов линий, из которых состоит путь. Квадратные скобки показывают, что это открытый путь, а круглые скобки, что это закрытый путь. Если внешние скобки опущены, как в третьем и четвертом вариантах синтаксиса, это означает закрытый путь.

Вывод значений типа `path` осуществляется в виде первого и второго вариантов синтаксиса.

5.8.6. Полигоны

Полигоны представляются списками точек (вершин полигона). Полигоны могут рассматриваться как эквиваленты закрытого пути, но хранятся по-другому и имеют свой собственный набор функций. Значения типа `polygon` задаются с помощью следующего синтаксиса:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
```

```
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1   , ... ,   xn , yn )
  x1 , y1   , ... ,   xn , yn
```

где точки являются конечными точками сегментов линий, которые образовывают область полигона.

Вывод значений типа `polygon` осуществляется в виде первого варианта синтаксиса.

5.8.7. Круги

Круги представляются точкой, которая является центром круга, и радиусом. Значения типа `circle` задаются с помощью следующего синтаксиса:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

где точка `(x,y)` является центром круга, а `r` — радиусом.

Вывод значений типа `circle` осуществляется в виде первого варианта синтаксиса.

5.9. Типы для представления сетевых адресов

В таблице 22 приведены типы данных для хранения адресов IPv4, IPv6 и MAC, предлагаемые PostgreSQL.

Таблица 22 – Типы для представления сетевых адресов

Имя	Размер хранения, байты	Описание
`cidr`	7 или 19	IPv4 и IPv6 (сети)
`inet`	7 или 19	IPv4 и IPv6 (узлы и сети)
`macaddr`	6	MAC-адреса

При сортировке типов данных `inet` и `cidr` адреса IPv4 всегда будут находится перед адресами IPv6, включая адреса IPv4, инкапсулированные или отображенные в адреса IPv6, такие как `::10.2.3.4` или `::ffff:10.4.3.2`.

5.9.1. `inet`

Значениями типа `inet` могут быть IPv4- или IPv6-адреса узлов, а также подсети, в которые входят эти адреса — все в одном поле. Подсети представляются количеством бит адреса узла, которые представляют адрес сети («netmask» (сетевую маску)). Если значение сетевой маски равно 32, а адрес — IPv4, то значение задает одиночный узел. В IPv6 длина адреса равна 128 битам, так что все 128 бит задают уникальный адрес узла. При необходимости задания только сети требуется использовать вместо типа `inet` тип `cidr`.

Формат ввода для этого типа выглядит так:

address/y

где address — это IPv4- или IPv6-адрес, а y — количество бит в сетевой маске. Если часть /y отсутствует, то сетевая маска считается равной 32 для IPv4 и 128 — для IPv6, задавая только один узел. При выводе, если часть /y задает только один узел, она не отображается.

5.9.2. cidr

Значениями типа cidr могут быть спецификации сетей IPv4 и IPv6. Форматы ввода и вывода соответствуют соглашениям CIDR. Формат для задания сетей следующий:

address/y

где address — это сетевое представление IPv4- или IPv6-адреса, а y — количество бит в сетевой маске. Если часть y опущена, она вычисляется с помощью старой классификации номеров сетей в тех случаях, когда это возможно, исходя из введенного значения адреса. Является ошибкой задание сетевого адреса, который содержит установленными те же биты, что и значение сетевой маски справа.

В таблице 23 приведены примеры.

Таблица 23 – Ввод значений типа cidr

Ввод	Вывод	abbrev(cidr)
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

5.9.3. Сравнение типов inet и cidr

Важнейшее отличие между типами данных inet и cidr состоит в том, что inet допускает значения с ненулевыми битами в правой части сетевой маски, в то время как cidr нет.

Примечание. Для вывода значений inet или cidr в других форматах предназначены функции host, text и abbrev.

5.9.4. macaddr

Значениями типа macaddr могут быть MAC-адреса, например аппаратные адреса Ethernet-карт (также MAC-адреса используются и для других целей). Ввод значений допускается в нескольких следующих форматах:

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'08002b010203'
```

Все приведенные примеры задают один и тот же адрес. Для цифр допускаются значения как в верхнем, так и в нижнем регистрах от a до f. Вывод всегда осуществляется в первом из указанных форматов.

Стандарт IEEE 802-2001 задает вторую из показанных выше форм (с переносами) как каноническую форму для MAC адресов и задает первую из показанных выше форм (с двоеточиями) как бит-зарезервированную нотацию, таким образом 08-00-2b-01-02-03 = 01:00:4D:08:04:0C. Это соглашение в настоящее время игнорируется и является справедливым только для устаревших сетевых протоколов (таких как Token Ring). PostgreSQL не отвергает бит-зарезервированную нотацию и всегда принимает форматы, используя канонический порядок LSB.

Оставшиеся четыре входных формата не соответствуют каким-либо стандартам.

5.10. Типы битовых строк

Битовые строки — это строки, которые состоят из единиц и нулей. Они могут использоваться для хранения и визуализации битовых масок. В SQL существует два битовых типа: bit(n) и bit varying(n), где n является положительным целым числом.

Тип данных bit должен точно соответствовать длине n; попытка записи более короткого или более длинного значения битовой строки приведет к ошибке. Данные типа bit varying имеют переменную длину, максимальное значение которой n; более длинные строки будут отвергнуты. Написание bit без длины эквивалентно bit(1), в то время как bit varying без указания длины означает битовую строку неограниченной длины.

Примечание. Если производится явное приведение значения битовой строки к типу `bit(n)`, то это значение будет или усечено, или дополнено справа нулями до точной длины в `n` без генерации ошибки. Похожим образом, если производится явное приведение значения битовой строки к типу `bit varying(n)`, то это значение будет усечено справа, если оно длиной больше, чем `n` бит.

Информацию о синтаксисе битово-строчных константах см. в 1.1.2.5. Доступные битово-логические операторы и функции приведены в 6.6.

Пример

Использование 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 2 битовой строки не совпадает с типом bit(3)
INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;
  a  |  b
-----+-----
 101 | 00
 100 | 101
```

Значение битовой строки требует 1 байт для каждой группы из 8 битов плюс 5 или 8 байт дополнительно, в зависимости от длины данной строки (но длинные значения могут быть сжаты или помещены во вспомогательное место хранения).

5.11. Типы текстового поиска

PostgreSQL предоставляет два типа данных, которые разработаны для поддержки полнотекстового поиска. Поиск производится с помощью коллекции *документов* на обычном языке, в которых ищутся те документы, которые лучшим образом совпадают с *запросом*. Тип `tsvector` представляет собой документ в подходящей для поиска форме, а тип `tsquery` представляет собой некий запрос.

5.11.1. `tsvector`

Значение `tsvector` представляет собой сортированный список разных *лексем*, которые являются разными формами одного и того же слова, представленными в *нормализованном* виде. Сортировка и удаление дублирующих значений выполняются автоматически при вводе, например:

```
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 лексем, содержащих пробелы или знаки пунктуации, используются кавычки:

```
SELECT $$the lexeme '    ' contains spaces$$::tsvector;
                  tsvector
-------------------------------------------
 '    ' 'contains' 'lexeme' 'spaces' 'the'
```

В данном ниже и следующем примерах используются экранированные `$` строковые литералы, чтобы избежать путаницы с наличием двойных кавычек внутри литералов. Одинарные кавычки и символы `\`, используемые в литералах, должны вводится дважды:

```
SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;
                     tsvector
-----------------------------------------------
 'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

Для лексем можно указать целое число, указывающее позицию:

```
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 показывает местоположение слова в документе. Информация о позиции может быть использована для вычисления близких (находящихся рядом) слов (proximity ranking). Значения позиций могут находиться в диапазоне от 1 до 16383; большие значения приравниваются к 16383. Дублированные позиции для той же лексемы отбрасываются.

Лексемам, у которых указаны позиции, может затем быть назначен *вес*, который может иметь значение `A`, `B`, `C` или `D` (значение по умолчанию и при выводе не показывается):

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
-----------------------------
 'a':1A 'cat':5 'fat':2B,4C
```

Вес обычно используется для отражения структуры документа, например, чтобы отличить слова в заголовке от слов в теле документа. Функция определения ранга при поиске текста может назначать разные приоритеты для разных весов.

Это является важным для понимания того, что сам тип `tsvector` не выполняет никакой нормализации, он просто содержит слова, которые нормализуются нужным образом для того или иного применения. Например:

```
select 'The Fat Rats'::tsvector;
```

```
       tsvector
--------------------
 'Fat' 'Rats' 'The'
```

Для большинства приложений, выполняющих поиск текста на английском языке, данные выше слова не должны считаться нормализованными, но `tsvector` этого не знает. Текст «сырого» документа для нормализации слов в необходимый для поиска вид обычно обрабатывается функцией `to_tsvector`:

```
SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
 'fat':2 'rat':3
```

5.11.2. `tsquery`

Значение `tsquery` представляет собой набор предназначенных для поиска лексем, комбинированных логическими операторами `&` (И), `|` (ИЛИ) и `!` (НЕ). Для группировки операторов могут быть использованы круглые скобки:

```
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'
```

Без учета скобок наибольший приоритет имеет `!` (НЕ), затем `&` (И) и, наконец, `|` (ИЛИ).

Дополнительно лексемам в `tsquery` может назначаться одна и более букв с весом, которые ограничивают при поиске совпадение этих лексем только с лексемами в `tsvector`, имеющими тот же вес:

```
SELECT 'fat:ab & cat'::tsquery;
     tsquery
-----------------
 'fat':AB & 'cat'
```

Также лексемам в tsquery может быть назначен специальный символ *, чтобы указать совпадение по префиксу:

```
SELECT 'super:*'::tsquery;
  tsquery
-----------
 'super':*
```

Данный запрос будет совпадать с любым словом в tsvector, которое начинается на «super». Следует отметить, что префиксы сперва обрабатываются конфигурациями поиска текста, что означает, что данное сравнение возвращает истину:

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' );
?column?
----------
t
(1 row)
```

потому, что postgres после обработки становится postgr:

```
SELECT to_tsquery('postgres:*');
to_tsquery
------------
'postgr':*
(1 row)
```

что затем приводит к совпадению с postgraduate.

Правила экранирования для лексем, точно такие же как было описано выше для лексем в tsvector, любая необходимая нормализация слов должна выполняться перед помещением их в тип tsquery. Для выполнения такой нормализации применяется функция to_tsquery:

```
SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
```

5.12. Тип UUID

Тип данных uuid хранит универсальные уникальные идентификаторы (UUID) как определено RFC 4122, ISO/IEC 9834-8:2005 и смежными стандартами. (Некоторые системы называют этот тип данных как глобальный уникальный идентификатор или GUID .) UUID представляет собой 128-битный идентификатор, который генерируется с помощью алгоритма, призванного обеспечить максимально возможную уникальность генерируемого идентификатора от других генерируемых идентификаторов, использующих этот же алгоритм. Таким образом, для распределенных систем эти идентификаторы предоставляют более

высокую гарантию уникальности, чем генераторы последовательностей, которые уникальны только внутри одной БД.

UUID записывается как последовательность шестнадцатеричных цифр в нижнем регистре в нескольких группах, разделенных переносами, обычно группа из восьми цифр, за которой следует три группы по четыре цифры, за которыми следует группа из 12 цифр, т. е. всего 32 цифры, представленные 128 битами. Вид UUID в стандартной форме выглядит следующим образом:

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

PostgreSQL также позволяет следующие альтернативные формы ввода: использование цифр в верхнем регистре, заключение стандартного формата в фигурные скобки, допущение отсутствия всех или некоторых переносов и добавление переносов после любой группы из четырех разрядов. Например:

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

Вывод значений данного типа всегда осуществляется в стандартной форме.

PostgreSQL предоставляет для `uuid` функции хранения и сравнения, но ядро БД не включает никаких функций для генерации UUID, поскольку нет единого алгоритма, подходящего для всех приложений. Дополнительный модуль `contrib/uuid-ossp` предоставляет функции, которые реализуют несколько алгоритмов получения подобных уникальных идентификаторов. С другой стороны, UUID может генерироваться клиентскими приложениями или другим библиотеками, вызываемыми через функции на стороне сервера.

Примечание. В СУБД версии 9.6 расширение `pgcrypto` также содержит функции для генерации случайных UUID.

5.13. Тип XML

Тип данных `xml` может быть использован для хранения данных в формате XML. Преимущество хранения данных в данном типе по сравнению с хранением в поле типа `text` состоит в том, что при вводе значений происходит проверка форматирования, а также в том, что для этого типа существует набор функций для выполнения операций над этими данными (см. 6.14). Использование этого типа данных требует указания при компиляции `configure --with-libxml`.

Тип `xml` может хранить форматированные документы, а также фрагменты, которые могут иметь более чем один элемент верхнего уровня или символьный узел. Выражение `xmlvalue IS DOCUMENT` может быть использовано для определения, является ли конкрет-

ное значение xml полным документом или же только фрагментом.

5.13.1. Создание XML значений

Для создания значения xml из символьных данных используется функция xmlparse:

```
XMLPARSE ( { DOCUMENT | CONTENT } значение)
```

Пример

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title>
  <chapter>...</chapter></book>')
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

Указанный синтаксис является единственным способом конвертации символьных строк в XML-значения, соответствующим стандарту SQL. В тоже время существует специфичный для PostgreSQL синтаксис:

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

Тип xml не производит проверку вводимых значений согласно включаемому описанию типа документа (DTD), даже когда входное значение задает какой-нибудь DTD. Также в настоящий момент не существует встроенной поддержки проверку с другими языками XML разметки, такими как XML Schema.

Обратная операция получения значений строки символов из xml использует функцию xmlserialize:

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } value AS type )
```

где type может принимать одно из значений: character, character varying или text (или псевдоним на один из этих типов). В соответствии со стандартом SQL, это единственный способ преобразования из типа xml в один из символьных типов, но PostgreSQL также разрешает простое приведение типов.

При приведении значений символьных строк к xml и обратно без использования, соответственно, XMLPARSE или XMLSERIALIZE выбор DOCUMENT или CONTENT определяется с помощью параметра настройки сессии XML option, который можно установить, используя стандартную команду:

```
SET XML OPTION { DOCUMENT | CONTENT };
```

или команду в стиле синтаксиса PostgreSQL:

```
SET xmloption TO { DOCUMENT | CONTENT };
```

По умолчанию устанавливается CONTENT, так что допускаются все формы данных XML.

Примечание. С опцией XML option, включенной по умолчанию, напрямую не осуществляется приведение символьных строк к типу xml, если они содержат декларацию типа документа, потому что определение фрагмента содержимого XML не позволяет этого. В

таких случаях следует либо использовать XMLPARSE, либо изменить значение XML option.

5.13.2. Управление кодировкой

Необходимо проявлять осторожность при работе с кодировками на клиенте, сервере и в данных XML, передаваемых от клиента к серверу и обратно. При выполнении запросов на сервере и получении данных клиентом в текстовом режиме (что является нормальным режимом) PostgreSQL конвертирует все символьные данные, передаваемые между клиентом и сервером, в соответствующую кодировку (см. 12.3). Конвертирование включает строковое представление данных XML. Это означает, что объявления кодировки, содержащиеся в данных XML, могут стать неправильными, т. к. символьные данные конвертируются в другие кодировки при обмене этими данными между клиентом и сервером, а встроенное описание кодировки при этом не меняется. При таком поведении описание кодировки, содержащееся в строке символов, которая вводится в тип xml, игнорируется, и содержимое всегда считается в текущей кодировке сервера. Таким образом, чтобы сделать процесс ввода корректным, символьные строки в данных XML должны передаваться клиентом в текущей кодировке клиента. Именно клиент ответственен либо за конвертирование документа в текущую кодировку клиента перед отправкой на сервер, либо за согласование кодировок. При выводе значения типа xml не имеют каких-либо описаний кодировки, и клиенты должны привести кодировку в соответствие со своей.

Когда для передачи параметров запроса сервер использует бинарный режим и результаты запроса отправляются обратно клиенту, не выполняется никаких преобразований кодировок. В данном случае будет использовано описание кодировки в данных XML, а если оно отсутствует, будет считаться, что данные представлены в кодировке UTF-8 (как требует стандарт XML; PostgreSQL не поддерживает UTF-16). При выводе в данные будет добавлено описание той кодировки, которая задана клиентом, за исключением случая, когда клиент работает в UTF-8, в этом случае описание кодировки будет опущено.

Обработка данных XML в PostgreSQL будет менее подвержена ошибкам и более эффективна, если кодировка данных, кодировка клиента и сервера будут одинаковы. Поскольку внутри XML данные обрабатываются в UTF-8, эта обработка будет наиболее эффективной, если кодировка сервера также будет UTF-8.

ВНИМАНИЕ! Некоторые относящиеся к XML функции могут не работать для всех не ASCII данных, когда кодировка сервера отлична от UTF-8. Например, функция xpath().

5.13.3. Доступ к XML значениям

Тип данных xml не предоставляет никаких операторов сравнения. Это связано с тем, что для данных XML не существует задекларированных и универсальных алгоритмов сравнения. Из этого следует, что не существует возможности получить строки с помощью

сравнения значений столбцов типа `xml` с каким-либо искомым значением. Значения XML обычно сопровождаются отдельным ключевым полем, таким как `ID`. Альтернативным решением для сравнения значений XML может служить их конвертация в символьные строки перед выполнением сравнения, но важно помнить, что сравнение символьных строк мало применимо для сравнения XML.

Поскольку операторов сравнения для типа данных `xml` не существует, также невозможно напрямую создать индекс для столбцов этого типа данных. Если очень необходим быстрый поиск в данных XML, возможным решением является приведение выражения к типу символьной строки и индексирование этих строк или индексирование XPath-выражения. Фактический запрос должен быть составлен так, чтобы производить поиск по индексированному выражению.

Для ускорения поиска в XML данных также может быть использована функциональность полнотекстового поиска в PostgreSQL.

5.14. Типы JSON

Тип данных JSON используются для хранения JSON (JavaScript Object Notation) данных, как указано в RFC 7159. Такие данные также могут быть сохранены в виде текста, но тип данных JSON гарантирует, что каждое значение хранится в соответствии с правилами JSON.

В PostgreSQL есть два типа данных JSON: `json` и `jsonb`. Они принимают *почти* одинаковые наборы значений в качестве входных. Основная практическая разница состоит в эффективности. Тип `json` хранит точную копию входного текста и при его использовании вызываются функции его обработки; в то время как `jsonb` хранит данные в двоичном формате, что делает его немного медленнее при изменении, но значительно быстрее при чтении, так при повторной обработке не требуется вызов этих функций. `jsonb` также поддерживает индексацию, которая может быть значительным преимуществом по сравнением с `json`.

Т.к. тип `json` хранит точную копию входного текста, он будет сохранять незначительные пробелы между токенами, а также порядок ключей в пределах объектов `json`. Кроме того, если объект `json` содержит ключ несколько раз, все пары ключ-значение будут сохраняться (функции обработки позволяют получить последнее его значение). Напротив, `jsonb` не сохраняет лишние пробелы, не сохраняют порядок ключей объектов, а также не хранит дубликаты ключей объектов. Если дубликаты ключей задаются на инициализации, то сохраняется только последнее значение ключа.

Если приложению необходимо хранить данные в формате JSON, то рекомендуется хранить эти данные в формате `jsonb`, если не требуется порядок ключей объекта.

СУБД PostgreSQL требуется хотя бы один символ для определения кодировки базы

данных. Поэтому невозможно типам JSON жестко следовать спецификации JSON, если кодировка базы данных не является UTF8. Если в JSON содержатся символы, которые не могут быть представлены в кодировке базы данных, произойдет сбой.

RFC 7159 позволяет JSON строкам содержать управляющие последовательности Unicode, обозначаемые `\uxxxx`. В функции ввода для типа `json` Unicode последовательности могут быть использованы независимо от кодировки базы данных, и они проверяются только на синтаксическую корректность (т.е., что четыре шестнадцатеричные цифры следуют за `\u`). При этом функция ввода типа `jsonb` строже: она запрещает использовать последовательности Unicode для не ASCI-символов (идентификатор которых больше `U+007F`), если кодировка базы данных не является UTF8. Тип `jsonb` не допускает `\u0000` (т.к. этот символ не может представлен в PostgreSQL ни одним текстовым типом), и требует, что любое использование суррогатных пар Unicode для обозначения символов, неподчиняющихся Basic Multilingual Plane, должно быть допустимым. Допустимые последовательности Unicode конвертируются в эквивалентные ASCII или UTF8 последовательности; это включает в себя хранение суррогатных пар в одном символе.

Примечание. Многие функции обработки JSON, описанные в разделе (см. 6.15) будут конвертировать Unicode последовательности символов и, следовательно, будут вызывать те же типы ошибок описанных выше, даже если входные типы `json` или `jsonb`

При преобразовании JSON (текста) в `jsonb`, примитивные типы, описанные в RFC 7159 фактически отображаются на родных типах PostgreSQL (см. таблицу 24). Таким образом, есть некоторые незначительные дополнительные ограничения на то, что является допустимыми типами данных `jsonb`. Ввиду реализации `jsonb` будет считать недопустимыми значения числовых типов данных, которые находятся вне диапазона типа числовых типов PostgreSQL, а JSON не будет. Такие реализации определенных ограничений разрешено RFC 7159. Тем не менее, на практике такие проблемы гораздо чаще встречаются в других реализациях, поэтому принято представлять численный тип JSON как IEEE 754 с двойной точностью с плавающей точкой (которая допустима в RFC 7159). При использовании JSON в качестве формата обмена с такими системами, существует вероятность потери точности данных.

И наоборот, как отмечалось, в таблице есть некоторые незначительные ограничения на формат ввода в JSON примитивных типов, которые не относятся к соответствующим типам PostgreSQL.

Таблица 24 – Примитивные типы JSON и соответсвующие им типы PostgreSQL

Примитивный тип JSON	Соответствующий тип PostgreSQL	Замечание
string	text	u0000 не разрешен, как и все не ASCII последовательности символов Unicode, если кодировка базы данных не является UTF8
number	numeric	Nan и infinity не являются допустимыми
boolean	boolean	Разрешены только значения true и false в нижнем регистре
null	(отсутствует)	ключевое слово NULL имеет иное назначение

5.14.1. Синтакс ввода-вывода типа JSON

Синтакс ввода-вывода для типа данных JSON указан в RFC 7159.

Следующие запросы являются корректными для типов json и jsonb:

```
 -- Простые скалярные/примитивные значения
 -- Примитивными значениями могут быть числами, строки, заключенные в кавычки,
 -- true, false или null
SELECT '5'::json;

 -- Массив из 0 или более элементов (элементы не должны быть все одного типа)
SELECT '[1, 2, "foo", null]'::json;

 -- Объект, содержащий пары ключ-значение
 -- Замечание: ключи объекта должны быть всегда заключены в кавычки
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;

 -- Массивы и объекты могут быть вложены произвольным образом
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

Как отмечалось ранее, если на вход функции ввода типа json подается строка, то она будет выведена на печать без какой-либо дополнительной обработки, а jsonb не сохраняет семантически-незначительные детали, такие как пробелы. Например, обратите внимание на различия здесь:

```
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)
```

Семантически незначительная деталь: в `jsonb` числовые типы будут печататься в соответствии с его поведением. На практике это означает, что число введенных с `E` нотацией будут напечатаны 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)
```

Тем не менее, `jsonb` сохранит последние нули, как показано в этом примере, даже если они являются семантически незначимыми.

5.14.2. Эффективная разработка документов JSON

Представление данных в виде JSON могут быть значительно более гибким, чем в традиционной реляционной модели данных, которое является очевидной в условиях, когда требуется больше гибкости. Возможно сосуществование этих подходов и дополнение друг друга в рамках одного приложения. Тем не менее, даже для приложений, где гибкость желаема, все же рекомендуется, что документы JSON имели несколько фиксированную структуру. Структура является обычно нефиксированной (хотя соблюдения некоторых бизнес-правил декларативно можно придерживаться), но с известной структурой становится проще писать запросы, которые эффективно суммируют набор “документов” (datums) в таблице.

Данные JSON является объектом контроля, как и любой другой тип данных, когда хранятся в таблице. Хотя хранение больших документов возможно, имейте в виду, что любое обновление активирует блокировку записи таблицу. Рассмотрим ограничения на JSON документы на приемлемых размерах документов для того, чтобы уменьшить блокировки по обновлению между транзакциями. В идеале JSON документы должны представляет собой атомарный объект, такой, что бизнес-правила не могут разделить документ на более мелкие объекты, которые могут быть изменены независимо друг от друга.

5.14.3. Содержание и существование в jsonb

Проверка *на содержание* является важной особенностью jsonb. Для типа json нет аналогичных проверок. Тесты на содержание заключается в том, содержится ли один jsonb объект в другом. Эти примеры возвращают истину, везде, где не оговорено иное:

```
 -- Простые скалярные/примитивные значения
 -- могут содержаться только в совпадении их типов:
SELECT '"foo"'::jsonb @> '"foo"'::jsonb;

 -- Массив справа содержится в массиве слева:
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;

 -- Сортировка массива не важна, поэтому это также возвращает true:
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;

 -- Дупликаты элементов в массиве также не влияют:
SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;

 -- Объект справа содержится в объекте слева:
SELECT '{"product": "PostgreSQL", "version": 9.6,
 "jsonb":true}'::jsonb @> '{"version":9.6}'::jsonb;

 -- Массив справа не содержится в массиве слева,
 -- хотя в объект слева вложен в него:
SELECT '[1, 2, [1, 3]]'::jsonb @>
       '[1, 3]'::jsonb;  -- ожидается false

 -- Изменяем уровень вложенности объекта
 -- Объект содержится в контейнере слева:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;

 -- Уровни вложенности не соответсвуют друг другу
 -- Ожидается false
SELECT '{"foo": {"bar": "baz"}}'::jsonb @>
       '{"bar": "baz"}'::jsonb;  -- ожидается false
```

Общий принцип заключается в том, что объект должен соответствовать структуре и данным после мысленного отбрасывания некоторых элементов массива. Но помните, что порядок элементов массива не является существенным при выполнении этой операции и дублирование его элементов рассматривается только один раз.

```
 -- Этот массив содержит примитивное значение строки:
SELECT '["foo", "bar"]'::jsonb @> '"bar"'::jsonb;

 -- Это исключение здесь не работает -- ожидается "не содержит"
SELECT '"bar"'::jsonb @> '["bar"]'::jsonb;  -- ожидается false
```

В `jsonb` есть оператор *существования*, который является разновидностью содержания: она проверяет есть ли строка (представленная как тип `text`) в качестве ключа объекта или элемента массива на верхнем уровне `jsonb`. Эти примеры возвращают `true`, если не указано иное:

```
 -- Строка существует, как элемент массива:
SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';

 -- Строка существует, как ключ объекта:
SELECT '{"foo": "bar"}'::jsonb ? 'foo';

 -- Значение объекта не существует:
SELECT '{"foo": "bar"}'::jsonb ? 'bar';  -- ожидается false

 -- Как и в случае с содержанием,
 -- существование должно иметь место на верхнем уровне:
SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar'; -- ожидается false

 -- Строка существует, если она совпадает с примитивной строкой JSON:
SELECT '"foo"'::jsonb ? 'foo';
```

Объекты JSON лучше подходят для проверки массивов на содержание или существование элементов в случае, если они содержат много ключей или элементов, потому что в отличие от массивов они внутренне оптимизированы для поиска, в то время как для поиска элемента в массиве используется линейный метод.

Различные операторы содержания и существования, а также и все остальные операторы JSON и функции, описаны в разделе 6.15.

5.14.4. Индексирование `jsonb`

Индексы GIN могут быть использованы для эффективного поиска ключей или пар ключ-значение при большом количестве `jsonb` документов (datums). Предоставляются два “класса операторов”, предоставляющие различные копмпромиссы в производительности и гибкости.

Операторный класс GIN по умолчанию для типа `jsonb` поддерживает запросы с операторами `@>`, `?`, `?&` и `?|`. (Подробнее ознакомиться с семантикой и приминением этих

оператором можно ознакомиться в таблице 68.) Пример создания индекса с этим классом операторов:

```
CREATE INDEX idxgin ON api USING gin (jdoc);
```

Операторный класс GIN `jsonb_path_ops` поддерживает индексирование только оператора `@>`. Пример создания индекса с использованием этого операторного класса:

```
CREATE INDEX idxginp ON api USING gin (jdoc jsonb_path_ops);
```

Рассмотрим пример таблицы, в которой хранятся в формате JSON документы, извлеченные из веб-службы сторонних с документальным определением его схемы. Типичный документ:

```
{
    "guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",
    "name": "Angela Barton",
    "is_active": true,
    "company": "Magnafone",
    "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 храним эти документы в таблице `api` в столбце `jdoc`типа `jsonb`. Если индекс GIN создан на этом столбце, то запросы наподобие следующих создадут индекс:

```
 -- Найти все документы, в которых ключ "company" имеет значение "Magnafone"
SELECT jdoc->'guid', jdoc->'name' FROM api
WHERE jdoc @> '{"company": "Magnafone"}';
```

Однако, индекс не может быть использован в запросах наподобие следующего, потому что хотя и оператор `?` поддерживает индексы, ему не может обратиться к индексируемому столбцу `jdoc` напрямую:

```
 -- Найти все документы, в которых ключ "tags"
 -- содержит ключ или массив элементов "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ? 'qui';
```

Тем не менее, изменив выражение для индекса, запрос выше может использовать индекс. Если запрос для конкретных элементов ключей `"tags"` общий, то можно определить следующий индекс:

```
CREATE INDEX idxgintags ON api USING gin ((jdoc -> 'tags'));
```

Теперь условие `WHERE jdoc -> 'tags' ? 'qui'` будет говорить приложению использовать индексируемый оператор `?` для индексации выражения `jdoc -> 'tags'`.

Другой подход к запросам – использование оператора содержания, например:

```
-- Найти все документы, в которых ключ "tags" содержит массив элементов "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"tags": ["qui"]}';
```

Простой индекс GIN на столбце `jdoc` может поддерживаться этим запросом. Но заметьте, что этот индекс будет хранить копии каждого ключа и каждого значения столбца `jdoc`, в то время как простой индекс из предыдущего примера будет хранить только найденные данные из ключа `tags`. Хотя определение простого индекса более гибкое (он поддерживает запросы с любым ключом), целевой запрос будет меньше и быстрее искать данные, чем простой индекс.

Хотя операторный класс `jsonb_path_ops` поддерживает запросы только с оператором `@>`, он имеет значительные преимущества в производительности над операторным классом `jsonb_ops`. Индекс `jsonb_path_ops` обычно меньше, чем `jsonb_ops` индекс при одинаковых данных и специфика поиска лучше, особенно когда запросы содержат ключи, которые появляются часто в данных. Поэтому поисковые операции, как правило, лучше, чем с операторным классом по умолчанию.

Техническое различие между `jsonb_ops` и `jsonb_path_ops` GIN индексами состоит в том, что первый создает независимые элементы индекса для каждого ключа и значения в данных, в то время как последний создает индексируемые элементы для каждого значения (здесь термин "значение"включает в себя элементы массива, хотя в терминологии JSON иногда считается элементы массива, отличные от значений в пределах объектов). В принципе, каждый элемент `jsonb_path_ops` индекса – это хэш значения и ключа(ей), ведущего к нему. Например, для индексации `{"foo": {"bar": "baz"}}` будет создан элемент, включающий в себя `foo`, `bar` и `baz` в значении хэша. Таким образом, запрос на содержание ищет этой структуры может привести к крайне специфичному индексному поиску; но нет никакого способа, чтобы понять, есть ли ключ `foo`. С другой стороны, индекс `jsonb_ops` будет создавать три элемента индекса, представляющих `foo`, `bar` и `baz` отдельно; при исполнении запроса на сожержание, индекс будет искать строки, содержащие все три эти элемента. В то время как индексы GIN могут выполнять поиск довольно эффективно, он будет меньше и медленнее, чем поиск эквивалентных `jsonb_path_ops`, особенно если есть очень большое количество строк, содержащих какой-либо одной из трех элементов по индексу.

Недостатком такого подхода является то, что `jsonb_path_ops` не создает никаких записей индекса для JSON структуры, не содержащей каких-либо данных, таких как

{"A": {}}. Если потребуется поиск документов, содержащих такую структуру, то это потребует полного индексного сканирования, которое довольно медленно. jsonb_path_ops поэтому плохо подходит для приложений, которые часто выполняют такие поиски.

jsonb также поддерживает индексы btree и hash. Они обычно полезны только если важно проверить соответствие полным JSON документам. Упорядочевание индексом btree для данных jsonb редко представляет интерес, но для полноты приведено следующее:

```
Объект > Массив > Логический тип > Численный тип > Строка > Null
Объект с n парами > объект с n - 1 парами
Массив с n элементами > массив с n - 1 элементами
```

Объекты с одинаковым числом пар сравниваются в порядке:

```
ключ-1, значение-1, ключ-2 ...
```

Отметим, что ключи объектов сравниваются в порядке их хранения; в частности, так короткие ключи хранятся до более длинных ключей, это может привести к результатам, которые могут быть неинтуитивными, таких как:

```
{ "aa": 1, "c": 1} > {"b": 1, "d": 1}
```

Аналогично, массивы с одинаковыми номерами элементов сравниваются в следующем порядке:

```
элемент-1, элемент-2 ...
```

Примитивные значения JSON сравниваются с использованием правил типов PostgreSQL, которые используются для описания этого примитивного типа. Строки сравниваются с использованием сопоставления по умолчанию.

5.15. Массивы

PostgreSQL поддерживает возможность определения типа столбца таблицы, как многомерного массива переменной длины. При этом элементами массива могут быть встроенные стандартные типы PostgreSQL, типы, определяемые пользователем, типы перечислений и составные типы.

5.15.1. Объявление массива

Для демонстрации использования типа массива рассмотрим следующую таблицу:

```
CREATE TABLE sal_emp (
  name             text,
  pay_by_quarter   integer[],
  schedule         text[][]
);
```

Видно, что столбец типа массива объявляется добавлением квадратных скобок после наименования типа элемента массива. Приведенная выше команда создает таблицу sal_emp, состоящую из столбца типа text (name), одномерного массива типа integer

(pay_by_quarter), представляющего собой поквартальный оклад сотрудника, и двумерного массива типа text schedule), содержащего его понедельный список дел.

Синтаксис команды CREATE TABLE позволяет также задавать и точное значение количества элементов в каждом измерении массива, например:

```
CREATE TABLE tictactoe (
  squares   integer[3][3]
);
```

Ограничения на количество элементов измерений массива не учитываются, и массивы с таким объявлением ведут себя так же, как и массивы с не заданными ограничениям.

А также не учитывается и задание размерности массивов. Массив обычных элементов определенного типа считается столбцом этого типа, независимо от размера и количества измерений. Таким образом, задание этих значений в команде CREATE TABLE следует рассматривать только в целях документирования. Это не оказывает влияние на работу системы.

Альтернативный синтаксис, соответствующий SQL-стандарту, для объявления одномерных массивов выглядит следующим образом (на примере объявления столбца pay_by_quarter):

```
pay_by_quarter  integer ARRAY[4],
```

Или без указания количества элементов:

```
pay_by_quarter  integer ARRAY,
```

5.15.2. Ввод значений элементов массива

Значения элементов массива в виде литеральной константы записываются внутри фигурных скобок и разделяются запятыми. Каждый элемент массива может быть заключен в двойные кавычки, более того, это необходимо в случае, когда в значении элемента встречаются запятые или фигурные скобки. В общем случае синтаксис объявления значений массива в виде константы выглядит так:

```
'{ val1 delim val2 delim ... }'
```

где delim является разделительным символом для заданного типа, как указано в pg_type. Для типов, которые по умолчанию поддерживает PostgreSQL, только тип box использует в качестве разделителя точку с запятой (;), тогда как остальные — запятую (,). Каждый val может представлять собой константу элемента массива или, в свою очередь, подмассив для многомерных массивов. Например:

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

В данном случае константа представляет собой двумерный массив, состоящий из трех измерений по три элемента типа integer в каждом.

Для задания неопределенного значения элементу массива используется ключевое слово NULL (может использоваться написание как в нижнем, так и в верхнем регистре). При необходимости задания действительного символьного значения NULL требуется заключать

значение в двойные кавычки.

Указанный тип констант массива является только одним из общих типов констант (см. 1.1.2.7). Константы изначально рассматриваются как текстовые строки, подаваемые на вход функций преобразования вводимых значений массива, при этом может потребоваться явная спецификация типа.

Рассмотрим некоторые INSERT-команды:

```
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 выполнения выглядит следующим образом:

```
SELECT * FROM sal_emp;
 name  |      pay_by_quarter       |                  schedule
-------+---------------------------+-----------------------------------------
 Bill  | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
 Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

Многомерные массивы обязаны иметь одинаковое количество элементов в каждом измерении. В противном случае генерируется ошибка:

```
INSERT INTO sal_emp
  VALUES ('Bill',
  '{10000, 10000, 10000, 10000}',
  '{{"meeting", "lunch"}, {"meeting"}}');
ОШИБКА: для многомерных массивов должны задаваться выражения с
 соответствующими размерностями
```

Также может быть использован конструктор 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']]);
```

Элементы массивов являются обычными константами или выражениями SQL; к примеру, строковые литералы заключаются в одинарные кавычки, в отличие от их заключения в двойные кавычки, когда они являются значениями элементов массива. Конструктор `ARRAY` описан в 1.2.12.

5.15.3. Доступ к массивам

Рассмотрим доступ к отдельному элементу массива на примере запроса имен сотрудников, чьи выплаты изменились во втором квартале:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];
 name
-------
 Carol
(1 row)
```

Порядковый номер элемента массива указывается внутри квадратных скобок. По умолчанию PostgreSQL использует нумерацию элементов массива, начиная с 1. Следовательно, массив из `n` элементов начинается элементом `array[1]` и заканчивается элементом `array[n]`.

Следующий запрос возвращает выплаты сотрудникам в третьем квартале:

```
SELECT pay_by_quarter[3] FROM sal_emp;
 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

Также возможен доступ к произвольному прямоугольному срезу массива или подмассива. Срез определяется нижней и верхней границами, разделенными двоеточием. Например, запрос первого элемента списка дел `Billa` для первых двух дней недели:

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';
        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

Если хотя бы одно измерение определено как срез, т. е. содержит двоеточие, все остальные измерения также рассматриваются как срезы. Любое измерение, заданное без указания двоеточия, рассматривается как срез от первого элемента до указанного. Например, `[2]` рассматривается как `[1:2]`, как приведено ниже:

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';
                    schedule
-----------------------------------------
 {{meeting,lunch},{training,presentation}}
(1 row)
```

Рекомендуется использовать одинаковый синтаксис для всех измерений, т. е. для всех измерений явно задавать срезы, например `[1:2][1:1]`, а не `[2][1:1]`.

Обращение к массиву возвращает `NULL`, когда пуст массив, или любое из запрашиваемых измерений. Также возвращается NULL-значение, если запрашивается элемент за границами массива (в этом случае ошибка не возникает). К примеру, поскольку текущая размерность столбца `schedule` составляет `[1:3][1:2]`, обращение к элементу `schedule[3][3]` возвратит `NULL`. Также не возникает ошибки, а возвращается NULL-значение при указании неверного количества измерений.

Обращение к массиву для выборки среза также возвращает NULL-значение, когда пуст массив, или любое из запрашиваемых измерений. В то же время, если запрашивается срез, целиком выходящий за границы массива, вместо NULL-значения возвращается пустой массив (с нулевым количество измерений). (Это отличается от поведения при запросе одного элемента.) Если запрашиваемый срез частично попадает на границы массива, результат обрезается по реальной части массива.

Для получения текущих границ массива предназначена функция `array_dims`:

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';
 array_dims
------------
 [1:2][1:2]
(1 row)
```

Функция `array_dims` возвращает результат в текстовом виде, что удобно для чтения, но не удобно для использования в коде программ. Также для получения верхней и нижней границ конкретного измерения предназначены функции `array_upper` и `array_lower`, соответственно:

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';
 array_upper
-------------
           2
box(1 row)
```

Функция `array_length` позволяет получить количество элементов указанного измерения:

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';
```

```
 array_length
--------------
            2
(1 row)
```

Функция cardinality возвращает общее число элементов в массиве по всем измерениями массива. Это фактически число строк вызова функции untest:

```
SELECT cardinality(schedule) FROM sal_emp WHERE name = 'Carol';
 cardinality
-------------
           4
(1 row)
```

Примечание. Данная функция может быть использована только в СУБД версии 9.6.

5.15.4. Модификация массивов

Значение типа массив может быть заменено полностью:

```
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 задать значение конкретному элементу:

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
  WHERE name = 'Bill';
```

или срезу:

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
  WHERE name = 'Carol';
```

Существующий массив может быть увеличен заданием новых элементов. Позиции между существующими значениями и вновь задаваемым заполняются NULL-значением. Например, если массив myarray содержит четыре элемента, то он будет содержать шесть значений при задании значения элементу myarray[6], причем элемент myarray[5] будет содержать NULL-значение. Подобное поведение возможно только для одномерных массивов.

Индексированное задание значений позволяет создать массив, нумерация элементов которого может начинаться не с 1. Например, задание значения элементу массива myarray[-2:7] создает массив с элементами, нумерующимися с -2 до 7.

Также новое значение массива может быть получено конкатенацией (||):

```
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 добавлять одиночный элемент к началу или концу одномерного массива. Также возможна конкатенация двух N-мерных массивов или N-мерного и N+1-мерного массивов.

В случае добавления одного элемента в начало или конец существующего одномерного массива результирующий массив будет иметь ту же нижнюю границу нумерации элементов, что и исходный. 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)
```

При конкатенации двух массивов с одинаковым количеством измерений результирующий массив имеет нижние границы измерений левого операнда. Результирующий массив включает в себя элементы левого операнда, после которого следуют элементы правого:

```
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)
```

В том же случае, когда N-мерный массив добавляется в начало или конец N+1-мерного массива, результат получается, аналогичный одномерному варианту, т. к. N-мерный массив является элементом N+1-мерного массива. Например:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [1:3][1:2]
(1 row)
```

Массив также может быть создан с использованием функций array_prepend, array_append или array_cat. Первые две функции поддерживают работу только с одномерными массивами, тогда как array_cat может работать и с многомерными. Следует отметить, что описанный ранее оператор конкатенации предпочтительнее, чем прямой вызов указанных функций. Фактически, эти функции в первую очередь существуют для реализации оператора конкатенации. В то же время, прямой их вызов может быть полезен в случае создания определенных пользователем агрегатов. Примеры использования:

```
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}}
```

5.15.5. Поиск в массивах

Для поиска в массиве необходимо проверить каждое его значение. Если известен размер массива, поиск может быть выполнен вручную:

```
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 для больших массивов это может быть неудобно, а при отсутствии сведений о размере вообще невозможно. Альтернативный способ рассмотрен в 6.23. Предыдущий запрос может быть записан иначе:

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

Также существует возможность найти строки, в которых все значения элементов равны 10000 следующим образом:

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

Или используя функцию `generate_subscripts` (см. таблицу 81):

```
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;
```

Этот и другие операторы для работы с массивами описаны далее в 6.18.

Примечание. Массивы не являются наборами значений, и попытки поиска в них могут говорить о неверной структуре БД. В этом случае лучше рассмотреть вместо массива отдельную таблицу, содержащую строки, представляющие собой элемент массива. Такое решение более удобно для поиска при большом количестве элементов.

5.15.6. Синтаксис входных и выходных значений массива

Внешнее представление массива, содержащего элементы, в текстовом виде состоит из представлений его элементов в соответствии с правилами конвертации ввода/вывода для типа элементов и некоторых символов, определяющих структуру массива. Символы,

определяющие структуру массива, включают фигурные скобки, обрамляющие элемент массива, и символ-разделитель, разделяющий элементы одного измерения. Символом-разделителем обычно является запятая, но может быть любым другим: определяется параметром `typdelim` для элемента массива. Для типов, которые по умолчанию поддерживает PostgreSQL, только тип `box` использует в качестве разделителя точку с запятой (`;`), тогда как остальные — запятую (`,`). В многомерном массиве каждое измерение (строка, плоскость, куб и т. п.) имеет свой собственный уровень фигурных скобок, причем элементы одного уровня должны разделяться запятыми.

Процедура вывода массива обрамляет двойными кавычками элементы, если они представляют собой пустые строки, содержат фигурные скобки, символы-разделители, двойные кавычки, символы `\` или пробелы, или совпадают с ключевым словом `NULL`. Двойные кавычки или символы `\`, входящие в элемент массива, должны быть экранированы символами `\`. Для числовых типов данных справедливо полагать отсутствие двойных кавычек, тогда как тестовые типы данных могут как содержать их, так и не содержать.

По умолчанию нижняя граница измерений массива устанавливается в 1. Для представления массивов с иным значением нижней границы значение диапазона измерения должно быть задано явно перед выводом содержимого массива. Это выполняется указанием в квадратных скобках нижней и верхней границ измерения через двоеточие. За явным указанием размера массива следует символ равенства, 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 явное указание размерности массива, только если в одном или более измерений нижняя граница отличается от 1.

В случае написания в качестве значения элемента `NULL` (в любом регистре), элемент считается содержащим NULL-значение. Присутствие любых кавычек или символа `\` отменяет это и позволяет задать строковое значение `NULL`. Также для обратной совместимости с версиями PostgreSQL ранее 8.2, конфигурационный параметр `array_nulls` может быть установлен в выключенное состояние `off` для предотвращение распознавания `NULL` в качестве NULL-значения.

Как было показано ранее, при задании значения массива возможно обрамление двойными кавычками каждого элемента массива. Это необходимо, если значение элемента может привести к неоднозначности разбора элементов массива. Например, элементы,

содержащие фигурные скобки, запятые (или разделители для указанного типа данных), кавычки, символы \, пробелы до или после, должны быть заключены в кавычки. Пустые строки и NULL-значения - аналогично. При необходимости указания двойных кавычек или символов обратной косой черты в заключенных в двойные кавычки значениях элементов массива следует использовать синтаксис экранирования строк, или предварять их символами \. Также для избежания указания кавычек, может быть использование экранирование символом \ для защиты всех символов, которые в противном случае могут быть восприняты как синтаксис массива.

Допустимо написание пробелов перед открывающейся или после закрывающейся скобок. Также пробелы могут присутствовать до или после отдельных строк значений. Во всех указанных случаях символы пробелов игнорируются. В тоже время пробелы внутри заключенных в двойные кавычки элементов или находящиеся между непробельными символами внутри элемента не игнорируются.

Примечание. SQL-команда сначала интерпретируется как строковый литерал, а только затем как массив. В связи с этим требуется удвоение символа \. Например, для вставки символьного значения элемента массива, содержащего символ \ и двойную кавычку, необходимо указать:

```
INSERT ... VALUES (E'{"\\\\","\\""}');
```

Лексический процессор удалит один уровень \, и на вход процедуры разбора массива на значения попадет строка: `{"\\", "\""}`. И после этого на вход процедурам ввода конкретных элементов попадут символы \ и " (если при этом элементами массива являются типы, функции ввода которых, в свою очередь, специально обрабатывают символ \, например `bytea`, то для представления в результате всего одной \ понадобится изначально записать восемь символов обратной косой черты). Можно избежать необходимости удвоения символов \ экранированием с помощью символа `$` (см. 1.2.4).

Примечание. Конструктор `ARRAY` (см. 1.2.12) часто удобнее использовать, чем литеральный синтаксис задания элементов массива в SQL-командах, т. к. отдельный элемент конструкции `ARRAY` указывается в том же виде, как когда он не является частью массива.

5.16. Составные типы

Составной тип описывает структуру строки или записи; в сущности он представляет собой список имен полей и их типов данных. PostgreSQL позволяет использовать значения составного типа таким же образом, как и значения простых типов.

5.16.1. Объявление составного типа

Примеры объявления 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 объявления похож на конструкцию `CREATE TABLE`, за исключением того, что может использоваться только объявление имен полей и их типов без каких-либо объявлений ограничений целостности (типа `NOT NULL`). Важно отметить обязательность наличия ключевого слова `AS`, в случае его отсутствия система предположит, что имелся в виду совершенно другой вид команды `CREATE TYPE` и выдаст ошибку об устаревшем синтаксисе.

После определения типа возможно его использование для создания таблиц:

```
CREATE TABLE on_hand (
  item      inventory_item,
  count     integer
);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

или функций:

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric
AS 'SELECT $1.price * $2' LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;
```

При создании таблицы также автоматически создается и соответствующий составной тип, представляющий строку таблицы. Например, после создания таблицы:

```
CREATE TABLE inventory_item (
  name            text,
  supplier_id     integer REFERENCES suppliers,
  price           numeric CHECK (price > 0)
);
```

одноименный составной тип `inventory_item` может быть использован так же, как и рассмотренные ранее. Существует важное ограничение подобной реализации, т. к. с составным типом не ассоциируются никакие ограничения целостности; ограничения целостности, присутствующие в определении таблицы, *не применяются* к значениям составного типа за

ее пределами. (Частично обойти указанное ограничение возможно, используя домены в качестве членов составного типа.)

5.16.2. Ввод значений составного типа

При написании составного значения в виде литеральной константы значения ее полей заключаются в общие скобки и разделяются запятыми. Значение каждого поля может быть заключено в двойные кавычки (и обязано быть в них заключено, если содержит символы запятой или скобок). Таким образом, в общем виде формат константы выглядит следующим образом:

```
'( val1 , val2 , ... )'
```

Например:

```
'("fuzzy dice",42,1.99)'
```

который является допустимым значением составного типа `inventory_item`, рассмотренного выше. При необходимости задать значению поля NULL-значение в его позиции не указывается никаких символов. К примеру, в следующей записи NULL-значение задается третьему полю:

```
'("fuzzy dice",42,)'
```

Для задания пустой строки вместо NULL-значения используются двойные кавычки:

```
'("",42,)'
```

В этом примере первое поле задается пустой строкой, а третье — NULL-значением.

(Указанные константы являются частным случаем констант общего вида (см. 1.1.2.7). Константы изначально рассматриваются как строки и только после этого подаются на вход процедуры обработки входных значений составного типа. При этом может потребоваться и явное задание типа значения.)

Также для создания значений составного типа может использоваться выражение `ROW`. В большинстве случаев его использование проще литерального синтаксиса, поскольку дает возможность не задумываться о большом количестве вложенных двойных кавычек. Ниже приведены примеры задания составных значений этим способом:

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

Ключевое слово `ROW` является необязательным для составных типов, состоящих более чем из одного поля, т. е. предыдущие выражения можно записать проще:

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

Синтаксис выражений `ROW` приведен в 1.2.13.

5.16.3. Доступ к составному типу

Для доступа к конкретному полю столбца составного типа достаточно записать его имя через точку так же, как и при выборе столбца из таблицы. Это совпадает с выбором

столбца из таблицы, поэтому необходимо использование скобок. Рассмотрим попытку выбора поля столбца ранее рассмотренной таблицы on_hand:

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

Приведенный пример вызовет ошибку, т. к. анализатор в соответствии с синтаксисом SQL воспринимает item как имя таблицы, а не поля. Правильный запрос должен быть написан следующим образом:

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

или при указании имени таблицы (например в многотабличных запросах):

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

При этом окруженный скобками объект корректно интерпретируется как обращение к соответствующему столбцу, после чего выбирается значение указанного поля.

Похожий синтаксический подход применяется и для выбора поля составного значения. Например, для выбора только одного поля из результата функции, возвращающей составной тип, необходимо использовать запись следующего вида:

```
SELECT (my_func(...)).field FROM ...
```

Без скобок указанный запрос вызвал бы синтаксическую ошибку.

5.16.4. Модификация составного типа

Рассмотрим несколько примеров правильного синтаксиса вставки и обновления столбцов составного типа. Вставка или обновление значений столбцов целиком:

```
INSERT INTO mytab (complex_col) VALUES((1.1,2.2));

UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

В первом запросе, в отличие от второго, опущено выражение ROW, хотя допустимо его указание в обоих случая.

Возможно обновление значения отдельного поля столбца составного типа:

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;
```

Нет необходимости заключать в скобки имя столбца, следующего непосредственно за ключевым словом SET, в то же время это обязательно при обращении к полям столбцов таблиц справа от оператора присваивания.

Также обращения к полям могут быть использованы и в команде INSERT:

```
INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);
```

В случае когда для заполнения указываются не все поля столбца, остальные заполняются NULL-значением.

5.16.5. Синтаксис входных и выходных значений составного типа

Внешнее представление значения составного типа, состоящего из элементов, в текстовом виде интерпретируется в соответствии с правилами конвертации ввода/вывода для типа элементов, и некоторых символов, определяющих структуру составного типа.

Символы, определяющие структуру составного типа, включают скобки (`(` и `)`), обрамляющие все составное значение и запятую (`,`) разделяющую поля. Пробелы снаружи составного значения игнорируются, в то время как внутри они считаются частью значения поля и могут учитываться или не учитываться в зависимости от правил преобразования конкретного типа поля. Например:

```
'(  42)'
```

Пробелы будут проигнорированы в случае типа поля `integer`, и учтены, если тип поля — `text`.

Как было показано ранее, при задании значения составного типа возможно обрамление двойными кавычками каждого поля. Это необходимо, если значение поля может привести к неоднозначности разбора полей составного типа. Например, поля, содержащие скобки, запятые, кавычки или символы `\`, должны быть заключены в кавычки. При необходимости указания двойных кавычек или символов обратной косой черты в заключенных в двойные кавычки значениях полей они должны предваряться символом обратной косой черты. (Для представления символа двойных кавычек может быть использовано и их удвоение, по аналогии с правилами удвоения одинарных кавычек в литеральных строках SQL.) Также возможно экранирование символом `\` всех символов, совпадающих с используемыми для синтаксиса структуры составных значений.

Полностью пустое поле (при отсутствии символов, запятых или скобок) представляет NULL-значение. Для задания пустой строки вместо NULL-значения используются `""`.

Процедура вывода значений составного типа заключает в двойные кавычки значения отдельных полей, если они представляют собой пустые строки или содержат в себе скобки, запятые, двойные кавычки, символы `\` или пробелы. (Необходимости заключать в двойные кавычки пробельного пространства нет, но это повышает четкость записи.) Двойные кавычки и символы `\`, входящие в значение поля, должны быть удвоены.

Примечание. SQL-команда сначала интерпретируется как строковый литерал, а только затем как составное значение. В связи с этим требуется удвоение символа `\`. Например, для вставки символьного значения поля составного типа, содержащего символ `\` и двойную кавычку, необходимо указать:

```
INSERT ... VALUES (E'("\\"\\\\")');
```

Лексический процессор удалит один уровень `\`, и на вход процедуры разбора составного значения попадет строка: `("\"\\")`. И после этого на вход процедуре ввода конкретного поля попадут символы `"\`. (Если при этом полями составного типа являются типы, функции ввода которых, в свою очередь, специально обрабатывают символ `\`, например, `bytea`, то для представления в результате всего одной `\` понадобится изначально записать восемь символов `\`.) Можно избежать необходимости удвоения символов `\` с помощью

символа $ (см. 1.2.4).

Примечание. Конструктор ROW удобнее использовать, чем литеральный синтаксис задания составных значений в SQL-командах, т. к. отдельный элемент конструкции ROW указывается в том же виде, как когда он не является частью составного значения.

5.17. Типы диапазонов

Типы диапазонов — типы данных, представляющие диапазон значений определенного типа (называемого *подтипом* диапазона). Например, диапазон даты и времени может использоваться для представления диапазона времени разервирования переговорной комнаты. В этом случае типом диапазона будет tsrange (сокращение от «timestamp range»), а подтипом — timestamp. Подтип должен иметь глобальный порядок для определения места значения внутри, до или после диапазона значений. Типы диапазонов полезны, поскольку они представляют множество значений в одном значении диапазона, и концепция перекрывающихся диапазонов достаточно ясна. Использование диапазонов времен и дат для организации расписаний является наиболее понятным примером, но также могут быть полезны и диапазоны цен, инструментальных измерений и т. п.

5.17.1. Втроенные типы диапазонов

PostgreSQL имеет следующие встроенные типы диапазонов:

- int4range — диапазон значений integer;
- int8range — диапазон значений bigint;
- numrange — диапазон значений numeric;
- tsrange — диапазон значений timestamp without time zone;
- tstzrange — диапазон значений timestamp with time zone;
- daterange — диапазон значений date.

Дополнительно пользователем может быть создан собственный тип диапазона с помощью команды CREATE TYPE.

5.17.2. Примеры

Создание и наполнение:

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');
```

Включение:

```
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);
```

Проверка на пустоту:

```
SELECT isempty(numrange(1, 5));
```

5.17.3. Включающие и исключающие границы

Каждый непустой диапазон имеет две границы, нижнюю и верхнюю. Все точки между ними включаются в диапазон. Включающие границы подразумевают включение в диапазон самих границ, тогда как исключающие границы не включаются в диапазон.

В тестовом представлении диапазона, включающая нижняя граница обозначается символом `'['`, а исключающая нижняя — `'('`. Аналогично включающая верхняя обозначается символом `']'`, а исключающая верхняя — `'('`.

Функции `lower_inc` и `upper_inc` проверяют тип включения нижней и верхней границы диапазона соответственно.

5.17.4. Бесконечные (неограниченные) диапазоны

Нижняя граница может не указываться, что означает включение в диапазон всех точек, меньших верхней границы. Аналогичным образом может не указываться и верхняя граница, тогда включаются все точки, большие нижней границы. В случае отсутствия обеих границ — все значения типа элемента диапазона считаются входящими в диапазон.

Это эквивалентно восприятию нижней границы как «минус бесконечности», а верхней как «плюс бесконечности». Важно понимать, что эти бесконечные значения не являются значениями типа элемента диапазона и никогда не включаются в диапазон. (Таким образом не существует такого понятия как бесконечная включающая граница — при попытке задать такую, она автоматически конвертируется в исключающую границу.)

Также некоторые типы элементов имеют нотацию «infinity», но в этом случае это именно значение, рассматриваемое механизмом диапазонов. Например, в диапазонах даты и времени `[today,]` означает тоже самое, что и `[today,)`, тогда как `[today,infinity]` означает не то же, что и `[today,infinity)` — последнее исключает специальное `timestamp` значение `infinity`.

Функции `lower_inf` и `upper_inf` проверяют на бесконечность нижнюю и верхнюю границы диапазона соответственно.

5.17.5. Ввод/вывод диапазонов

Ввод значений диапазона должен следовать одному из следующих шаблонов:

```
(lower-bound ,upper-bound )
(lower-bound ,upper-bound ]
[lower-bound ,upper-bound )
```

```
[lower-bound ,upper-bound ]
empty
```

Круглые и квадратные скобки показывают тип включения/исключения нижних и верхних границ, как было описано ранее. Последний шаблон `empty` представляет пустой диапазон (диапазон, не включающий ни одной точки).

Нижняя граница может быть как строкой, подходящей для ввода значений подтипа, или отсутствовать, что означает отсутствие нижней границы. То же справедливо и в отношении верхней границы.

Каждое граничное значение может быть заключено в двойные кавычки. Это необходимо, если значение границы содержит круглые или квадратные скобки, запятые, кавычки или символы \, так как они могут быть восприняты как часть синтаксиса описания диапазона. (Для представления символа двойных кавычек может быть использовано и их удвоение, по аналогии с правилами удвоения одинарных кавычек в литеральных строках SQL.) Также возможно экранирование символом \ всех символов, совпадающих с используемыми для синтаксиса описания диапазона. Для задания в качестве значения границ пустой строки следует писать `""`, так как без указания этого граница считается бесконечной.

Допустимо написание пробелов перед или после значений диапазона, в тоже время пробелы между скобками воспринимаются как часть значений нижней или верхней границ. (В зависимости от типа элемента это может как иметь, так и не иметь значения.)

Примечание. Приведенные правила схожи с правилами задания значений полей для литералов составного типа. См. 5.16.5.

Примеры:

1. Включает 3, не включает 7 и включает все точки между:

```
SELECT '[3,7)'::int4range;
```

2. Не включает 3 и 7, но включает все точки между:

```
SELECT '(3,7)'::int4range;
```

3. Включает только 4:

```
SELECT '[4,4]'::int4range;
```

4. Не включает ничего (будет нормализовано к `'empty'`):

```
SELECT '[4,4)'::int4range;
```

5.17.6. Создание диапазонов

Каждый тип диапазона имеет соответствующую функцию-конструктор с тем же именем, что и сам тип диапазона. Использование подобной конструирующей функции более удобно, чем задание диапазона с помощью литеральной константы, поскольку не требует дополнительного заключения в кавычки значений границ. Конструирующая функция принимает два или три аргумента. Форма вызова с двумя аргументами создает диапазон

стандартного вида (нижняя граница включающая, верхняя — исключающая), тогда как форма вызова с тремя аргументами создает диапазон с типом включения границ, заданным третьим аргументом. При этом, третий аргумент может быть одним из `'()'`, `'(]'`, `'[)'` или `'[]'`.

Примеры:

1. Полная форма: нижняя граница, верхняя граница и текстовый аргумент, задающий тип включения границ:

```
SELECT numrange(1.0, 14.0, '(]');
```

2. Не задан третий аргумент, по умолчанию принимается `'[)'`:

```
SELECT numrange(1.0, 14.0);
```

3. Не смотря на то, что указано `'(]'`, при отображении значение будет конвертировано в каноническую форму, поскольку `int8range` является дискретным типом диапазона (см. ниже):

```
SELECT int8range(1, 14, '(]');
```

4. Использование `NULL` для задания отсутствия той или иной границы:

```
SELECT numrange(NULL, 2.2);
```

5.17.7. Типы дискретных диапазонов

Дискретным диапазоном является диапазон, тип элемента которого обладает четко заданным «шагом», например `integer` и `date`. В таких типах два элемента называются смежными, если межу ними не содержится других значений. Это отличает их от непрерывных диапазонов, в которых всегда (или почти всегда) существует возможность между любыми двумя выбранными значениями различить другие. Например, диапазон типа `numeric` является непрерывным, так же как и диапазон типа `timestamp`. (Не смотря на то, что `timestamp` обладает ограниченной точностью и теоретически может считаться дискретным, лучше считать его непрерывным, так как размер шага обычно не важен.)

Другим способом отличить дискретный тип диапазона является наличие четких понятий «следующего» и «предыдущего» значения для каждого значения элемента. Зная это, существует возможность перехода между включающей и исключающей формой представления границ путем выбора следующего или предыдущего значения вместо указанных изначально. Например, `integer` типы диапазонов `[4,8]` и `(3,9)` обозначают одинаковый набор значений, но это не будет верным для диапазонов типа `numeric`.

Тип дискретного диапазона должен иметь функцию приведения к каноническому виду, учитывающую шаг для типа элемента. Функция приведения к каноническому виду отвечает за приведение эквивалентных значений типа диапазона к идентичному представлению, в частности включающие и исключающие границы. При отсутствии заданной функции приведения к каноническому виду, диапазоны с различным форматом границ будут рассматриваться как не равные, даже если они могли представлять в действительности одинаковый

набор значений.

Встроенные типы диапазонов `int4range`, `int8range` и `daterange` используют каноническую форму с включающей нижней границей и исключающей верхней, т. е. `[)`. Заданные пользователем диапазоны могут использовать иные формы.

5.17.8. Определение новых типов диапазонов

Пользователь может создать свой собственный тип диапазона. Основанием для этого может являться необходимость использования диапазона с подтипом, для которого не существует встроенного типа диапазона. Например, создание нового типа диапазона для подтипа `float8`:

```
CREATE TYPE floatrange AS RANGE (
    subtype = float8,
    subtype_diff = float8mi
);

SELECT '[1.234, 5.678]'::floatrange;
```

Поскольку `float8` не имеет различимого «шага», в данном примере не создается функция приведения к каноническому виду.

В случае, если подтип считается имеющим дискретные значения, команда `CREATE TYPE` должна задавать функцию приведения к каноническому виду. Функция приведения к каноническому виду принимает на вход значение диапазона и должна возвращать эквивалентное значение диапазона, которое может иметь другие границы или формат. Каноническое представление двух диапазонов, содержащих одинаковый набор значений, например `integer` диапазонов `[1, 7]` и `[1, 8)`, должно быть идентичным. Неважно какое именно представление выбрано в качестве канонического, поскольку два эквивалентных значения с разным форматированием всегда приводятся к одному значению с одним форматированием. Дополнительно к регулированию включающих/исключающих границ, функция приведения к каноническому виду может округлять значения границ, в случае, когда требуемый шаг больше того, который способен хранить подтип. Например, `timestamp` диапазон может быть определен, как обладающий шагом в один час, в этом случае функции приведения к каноническому виду необходимо округлять значения границ к значению, кратному часу, или возможно вызывать ошибку.

При определении своих собственных типов диапазонов допускается задание различных B-tree классов операторов или сопоставления для подтипа для изменения порядка сортировки, определяющего какие именно значения попадают в данный диапазон.

Дополнительно, любой тип диапазона, который подразумевает использование GiST или SP-GiST индексов, должен определять функцию разности `subtype_diff` для подтипа

(без определения такой функции индекс может работать значительно менее эффективно). Функция разности принимает два значения подтипа и возвращает их разность (т. е. X-Y) как значение типа float8. В приведенном выше примере может быть использована обычная функция, лежащая в основе оператора - для типа float8; для других подтипов может потребоваться приведение типов. Также может требоваться творческий подход к представлению разности в виде числа. Для максимально возможной степени соответствия функция subtype_diff должна учитывать порядок сортировки применяемых класса операторов и сопоставления; в любом случае результат должен быть положительным, если первый аргумент больше второго согласно порядку сортировки.

Информация о создании типов диапазонов приведена в описании команды CREATE TYPE.

5.17.9. Индексирование

Для столбцов, использующих типы диапазонов, могут быть заданы GiST или SP-GiST индексы. Например, для создания GiST индекса необходимо выполнить:

```
CREATE INDEX reservation_idx ON reservation USING gist (during);
```

GiST и SP-GiST могут ускорять запросы, включающие следующие операторы работы с диапазонами: =, &&, <@, @>,<<, >>, -|-, &< и &> (см. 73).

Также для столбцов, использующих типы диапазонов, могут быть заданы и B-tree и hash индексы. Для этих типов индексов единственным значимым оператором для работы с диапазонами является =. Существуют B-tree порядки сортировки для значений типов диапазонов, соответствующие операторам < и >, но они скорее приблизительны и мало применимы. Поддержка B-tree и hash для типов диапазонов в первую очередь предназначена для возможности сортировки и хеширования в запросах, а не для создания индексов.

5.17.10. Ограничения на диапазонах

Естественный вид ограничения UNIQUE, используемый для скалярных значений, не применим для типов диапазонов. Вместо него более подходящим является ограничение исключения (см. описание CREATE TABLE ... CONSTRAINT ... EXCLUDE). Ограничения исключения позволяют определить ограничение по «не пересечению» для типов диапазонов, например:

```
CREATE TABLE reservation (
    during tsrange,
    EXCLUDE USING gist (during WITH &&)
);
```

Это ограничение предотвращает одновременное появление в таблице пересекающихся значений:

```
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)');
ERROR: конфликтующее значение ключа нарушает ограничение-исключения
 "reservation_during_excl"
DETAIL: 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")).
```

Существует возможность с помощью расширения `btree_gist` определить ограничение исключения для скалярных типов данных, использование которого совместно с ограничением исключения дли диапазона позволяет достичь большей гибкости. Например, после установки расширения `btree_gist` следующее ограничение отвергнет пересекающиеся диапазоны только, если совпадают номера переговорных 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)');
ERROR:  конфликтующее значение ключа нарушает ограничение-исключения
 "room_reservation_room_during_excl"
DETAIL:  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
```

5.18. Типы идентификаторов объектов

Идентификаторы объектов (OID) используются внутри PostgreSQL как первичные ключи для разных системных таблиц. OID не добавляются к создаваемым пользователями таблицам, если только при создании таблицы не было указано `WITH OIDS` или если переменная окружения `default_with_oids` не была установлена в значение `true`. Для представления идентификатора объекта используется тип `oid`. Существует несколько различных типов-псевдонимов для типа `oid`: `regproc`, `regprocedure`, `regoper`, `regoperator`, `regclass`, `regtype`, `regconfig` и `regdictionary` (таблица 25).

Таблица 25 – Типы OID

Имя	Указывает на	Описание	Пример
`oid`	все что угодно	Цифровой идентификатор объекта	`564182`
`regproc`	`pg_proc`	Имя функции	`sum`
`regprocedure`	`pg_proc`	Функция с типами аргументов	`sum(int4)`
`regoper`	`pg_operator`	Имя оператора	`+`
`regoperator`	`pg_operator`	Оператор с типами аргумента	`*(integer,integer)` или `-(NONE,integer)`
`regclass`	`pg_class`	Имя таблицы	`pg_type`
`regtype`	`pg_type`	Имя типа данных	`integer`
`regconfig`	`pg_ts_config`	Конфигурация текстового поиска	`english`
`regdictionary`	`pg_ts_dict`	Словарь текстового поиска	`simple`

Тип `oid` реализован как беззнаковое четырехбайтное целое число. Таким образом, значение этого типа не настолько велико, чтобы обеспечить уникальность в пределах всей БД в больших БД или даже в отдельных больших таблицах. В связи с этим, использование столбца OID в созданных пользователем таблицах в качестве первичного ключа является неправильным. Значения OID лучше всего использовать только для ссылок в системных таблицах.

Хотя для типа `oid` существуют некоторые операции сравнения, он может быть приведен к типу `integer` и значениями этого типа можно манипулировать, используя стандартные операторы для типа `integer`. (При этом следует остерегаться возможных проблем с преобразованием знаковых чисел в беззнаковые.)

Типы, которые являются псевдонимами OID, не имеют своих собственных операций, за исключением специальных подпрограмм ввода/вывода. Эти подпрограммы нужны для ввода и отображения символьных имен системных объектов, а не для использования цифровых значений типа `oid`. Типы-псевдонимы позволяют упростить поиск значений OID для объектов. Например, чтобы увидеть в таблице `pg_attribute` строки, которые относятся к таблице `mytable`, можно выполнить запрос:

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;
```

вместо:

```
SELECT * FROM pg_attribute
  WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');
```

Данный запрос выглядит не очень хорошо и слишком упрощен. Более сложный подзапрос нуждался бы в запросе правильного OID, если в других схемах существуют другие таблицы с именем `mytable`. Подпрограмма преобразования ввода значений типа `regclass` управляет поиском таблицы, соответствующей установленному пути поиска схем, и она выбирает правильный идентификатор автоматически. Похожим образом, приведение табличного OID к типу `regclass` позволяет получить символьное представление цифрового значения OID.

Все типы-псевдонимы OID допускают использование имен с указанием схем и отображают при выводе имена со схемами, если объект не найден в текущем пути поиска. Типы-псевдонимы `regproc` и `regoper` имеют ограничение по использованию, поскольку позволяют вводить только имена, которые являются уникальными (не перегруженными). В большинстве случаев более применимы типы `regprocedure` или `regoperator`. Для типа `regoperator` унарные операторы идентифицируются с помощью `NONE` для неиспользуемого операнда.

Дополнительное свойство типов псевдонимов OID состоит в том, что если какая-либо константа одного из этих типов используется в хранимых выражениях (таких как выражения, используемые для генерации значений по умолчанию в столбцах, а также в представлениях), она создает зависимость от объекта, на который указывает. Например, если какой-либо столбец имеет выражение для генерации значения по умолчанию вида `nextval('my_seq'::regclass)`, то PostgreSQL понимает, что это выражение зависит от последовательности `my_seq`; в этом случае СУБД не позволит удалить эту последовательность, если сперва не удалить данное выражение.

Типы идентификатора, используемые СУБД:

- `xid` или идентификатор транзакции (аббревиатура `xact`) — тип данных системных столбцов `xmin` и `xmax`. Идентификаторы транзакций являются 32-битными значениями;
- `cid` или идентификатор команд — тип данных системных столбцов `cmin` и `cmax`. Идентификаторы команд также являются 32-битными значениями;
- `tid` или идентификатор записи (идентификатор строки таблицы) — тип данных системного столбца `ctid`. Идентификатор записи — это пара (номер блока, индекс записи внутри блока), которая идентифицирует физическое расположения строки внутри своей таблицы.

О системных столбцах см. в 2.4.

5.19. Тип pg_lsn

Примечание. Данный тип используется только в версии СУБД 9.6.

Тип pg_lsn может быть использован для хранения LSN (Log Sequence Number), который является указателем на положение XLOG. Этот тип представляет XLogRecPtr и является внутренним системным типом PostgreSQL.

Внутреннее устройство LSN – это 64-битный integer, представляющий позицию байта в потоке WAL. Он печатается как два шестнадцатеричных числа до 8 разрядов включительно, разделенных слешом, например, 16/B374D848. Тип pg_lsn поддерживает стандартные операторы сравнения, такие как = и >. Можно получить разность двух LSN с помощью оператора -, результатом в таком случае является число байтов, разделяющих эти WAL позиции.

5.20. Псевдотипы

Система типов PostgreSQL содержит некоторое количество записей специального назначения, которые все вместе называются *псевдотипами*. Псевдотип не может быть использован как тип данных в каком-либо столбце. Каждый из доступных псевдотипов полезен в ситуациях, где поведение функции не укладывается в простое получение и возвращение некоего значения специфического типа SQL. В таблице 26 приведены существующие псевдотипы.

Таблица 26 – Псевдотипы

Имя	Описание
any	Показывает, что функция принимает входное значение любого типа
anyelement	Показывает, что функция принимает любой тип данных
anyarray	Показывает, что функция принимает массив любого типа
anynonarray	Показывает, что функция принимает любой тип данных, не являющийся массивом
anyenum	Показывает, что функция принимает любой тип данных перечисления (см. 5.7)
anyrange	Показывает, что функция принимает любой тип данных диапазонов (см. 5.17)
cstring	Показывает, что функция принимает или возвращает строку с завершающие нулем, как в языке C
internal	Показывает, что функция принимает или возвращает внутренний тип данных сервера
language_handler	Описывается обработчик вызова процедурного языка для возврата типа language_handler
fdw_handler	Описывается обработчик внешних данных для возврата типа fdw_handler

Окончание таблицы 26

Имя	Описание
`record`	Показывает, что функция возвращает неопределенный тип строки таблицы
`trigger`	Триггерная функция для возврата типа `trigger`
`void`	Показывает, что функция не возвращает значения
`opaque`	Все устаревшие имена типов, которые обычно обслуживаются для всех вышеуказанных целей

Функции, написанные на языке C (в том числе встроенные или загружаемые динамически), могут быть объявлены, чтобы принимать или возвращать любые из вышеописанных псевдотипов данных. При написании функции необходимо убедиться, что она будет вести себя правильно, когда в качестве аргумента будет использоваться какой-либо псевдотип.

Функции, написанные на процедурных языках, могут использовать псевдотипы только так, как разрешено в этом процедурном языке. В текущих процедурных языках везде запрещено использовать псевдотипы как тип аргумента и разрешено использование только псевдотипов `void` и `record`, как типов результата (плюс `trigger`, когда функция используется как триггер). Некоторые процедурные языки также поддерживают полиморфные функции, использующие типы `anyelement`, `anyarray`, `anynonarray`, `anyenum` и `anyrange`.

Псевдотип `internal` используется для описания функций, которые задуманы только для внутренних вызовов самой СУБД и не предназначены для прямых вызовов в SQL-запросах. Если функция имеет, по крайней мере, один аргумент типа `internal`, то она не может вызываться из SQL. Чтобы соответствовать этому ограничению, важно следовать следующему правилу при написании функции: не создавать функцию, которая объявляется как возвращающая тип `internal`, если она не имеет, по крайней мере, одного аргумента типа `internal`.

6. ФУНКЦИИ И ОПЕРАТОРЫ

PostgreSQL предоставляет большой набор функций и операторов для встроенных типов данных. Пользователи могут дополнительно определить собственные функции и операторы. Для просмотра списка доступных функций и операторов используются команды `\df` и `\fdo` утилиты `psql`.

С точки зрения переносимости следует обратить внимание, что подавляющее число функций и операторов, описываемых в данной главе, исключая наиболее простые операторы и операторы сравнения, а также явно отмеченные функции, отсутствуют в стандарте SQL. Некоторые из этих функций представлены в других СУБД и во многих случаях, данная функциональность является совместимой и согласованной для разных реализаций. Данная глава также не является полной; в соответствующих разделах данного руководства встречаются и дополнительные функции.

6.1. Логические операторы

Доступны следующие логические операторы: `AND`, `OR`, `NOT`.

SQL-стандарт использует трехзначную логику вычислений, в которой третьим значением является NULL-значение. Правила вычисления логических операторов `AND` и `OR` приведены в таблице 27.

Таблица 27 – Логические операторы `AND` и `OR`

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

Операторы `AND` и `OR` обладают свойством коммутативности, т. е. результат их выполнения не зависит от порядка записи операндов. Правила вычисления логического оператора `NOT` приведены в таблице 28.

Таблица 28 – Логический оператор `NOT`

a	NOT a
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Порядок вычислений выражений см. в 1.2.14.

6.2. Операторы сравнения

Доступны стандартные операторы сравнения, приведенные в таблице 29.

Таблица 29 – Операторы сравнения

Оператор	Описание
<	меньше
>	больше
<=	меньше или равно
>=	больше или равно
=	равно
<> или !=	not equal

Примечание. Оператор != преобразуется в оператор <> на этапе лексического разбора, поэтому нет возможности определить операторы != и <> так, чтобы они реализовывали различную логику.

Операторы сравнения определены для всех типов, для которых это имеет смысл. Все операторы сравнения являются бинарными операторами, которые возвращают значение типа `boolean`. Выражения типа `1 < 2 < 3` не являются допустимыми (поскольку нет оператора < для типа `boolean`).

В дополнение к операторам сравнения имеется специальная конструкция `BETWEEN`:

```
a BETWEEN x AND y
```

которая эквивалентна:

```
a >= x AND a <= y
```

Важно учитывать, что при сравнении с помощью `BETWEEN` конечные значения включаются в диапазон. `NOT BETWEEN` выполняет противоположное сравнение:

```
a NOT BETWEEN x AND y
```

эквивалентно:

```
a < x AND a > y
```

Между этими конструкциями нет никаких различий, кроме вычислительных затрат по внутреннему переводу первой конструкции во вторую. `BETWEEN SYMMENTRIC` аналогичен конструкции `BETWEEN` с тем исключением, что не требует от аргумента с левой стороны оператора `AND` быть меньше правого, корректный диапазон определяется автоматически.

Для проверки, является ли значение NULL или нет, используются конструкции:

```
expression IS NULL
expression IS NOT NULL
```

или эквивалентные им, но нестандартные:

```
expression ISNULL
expression NOTNULL
```

Нельзя для проверки на определенность использовать `expression = NULL`, поскольку NULL всегда не равно `NULL`, и это выражение будет равно `FALSE` во всех случаях. (NULL является неопределенным значением, и неизвестно, равны ли два неопределенных значения или нет.) Такое поведение соответствует стандарту SQL.

Однако существует ряд приложений, которые (неправильно) требуют, чтобы `expression = NULL` возвращало `TRUE`, если выражение `expression` вычисляется в NULL-значение. Настоятельно рекомендуется модифицировать подобные приложения для большего соответствия стандарту SQL. Если это не возможно, может быть использован конфигурационный параметр `Transform_Null_Equals`. При его установке PostgreSQL будет преобразовывать `x = NULL` в `x IS NULL`.

Примечание. Если выражение `expression` представляет собой значение типа `ROW`, `IS NULL` равно `TRUE`, когда либо само значение является неопределенным, либо все его поля содержат неопределенное значение, в то время как `IS NOT NULL` равно `TRUE`, когда само значение не является неопределенным, и все поля так же не содержат неопределенных значений. В результате такого поведения для выражений, которые являются значением-строкой таблицы `IS NULL` и `IS NOT NULL` не всегда возвращают противоположные результаты, например, выражение, которое является строкой таблицы, содержащей в своих полях как значения NULL так и не-NULL, будет возвращать `FALSE` в обоих случаях. Это соответствует стандарту SQL.

Обычный оператор сравнения возвращает NULL-значение при сравнении неопределенных значений. Например, выражения `7 = NULL` и `7 <> NULL` возвращают `NULL`. Когда такое поведение не подходит, следует использовать конструкции `IS [ NOT ] DISTINCT FROM`:

```
expression IS DISTINCT FROM expression
expression IS NOT DISTINCT FROM expression
```

Для выражений, не являющихся неопределенными значениями, `IS DISTINCT FROM` работает как оператор <>. При сравнении двух неопределенных значений возвращается `FALSE`, а в случае одного неопределенного значения — `TRUE`. Аналогично, `IS NOT DISTINCT FROM` соответствует = при сравнении значений, не являющихся неопределенными, возвращает `TRUE` при сравнении двух неопределенных значений, и `FALSE` в случае одного неопределенного значения. Таким образом, эти конструкции рассматривают NULL как нормальное значение, а не как неопределенное.

Логические выражения могут быть так же проверены с использованием следующих конструкций:

```
expression IS TRUE
```

```
expression IS NOT TRUE
expression IS FALSE
expression IS NOT FALSE
expression IS UNKNOWN
expression IS NOT UNKNOWN
```

Эти выражения подобны выражению IS NULL, всегда возвращают значения TRUE или FALSE и никогда — NULL. NULL считается логическим значением «неизвестно». IS UNKNOWN и IS NOT UNKNOWN в действительности являются тем же самым, что и соответственно IS NULL и IS NOT NULL, за исключением того, что значение операнда должно иметь тип boolean.

6.3. Математические операторы и функции

В таблице 30 представлены доступные стандартные математические операторы.

Таблица 30 – Математические операторы

Оператор	Описание	Пример	Результат
+	Сложение	2 + 3	5
–	Вычитание	2 - 3	-1
*	Умножение	2 * 3	6
/	Деление (целочисленное деление обрезает результат)	4 / 2	2
%	Модуль (остаток от деления)	5 % 4	1
^	Возведение в степень	2.0 ^ 3.0	8
|/	Квадратный корень	|/ 25.0	5
||/	Кубический корень	||/ 27.0	3
!	Факториал	5 !	120
!!	Факториал (как префиксная операция)	!! 5	120
@	Абсолютное значение	@ -5.0	5
&	Побитовое AND	91 & 15	11
|	Побитовое OR	32 | 3	35
#	Побитоое XOR	17 # 5	20
~	Побитовое NOT	~1	-2
<<	Битовый сдвиг влево	1 « 4	16
>>	Битовый сдвиг вправо	8 » 2	2

Битовые операторы работают только с целочисленным типами, тогда как остальные — со всеми числовыми типами данных. Также битовые операторы доступны для типов bit и bit varying (см. 6.6).

В таблице 31 показаны доступные в PostgreSQL математические функции. В таб-

лице dp обозначает тип double precision. Многие из этих функций имеют несколько форм с различными типами аргументов. За исключением нескольких случаев, функции возвращают тот же тип, что и заданный аргумент. Функции, работающие с аргументами double precision, как правило, реализованы с использованием системной библиотеки языка C ОС.

Таблица 31 – Математические функции

Функция	Тип результата	Описание	Пример	Результат
abs(x)	Как у аргумента	Абсолютное значение	abs(-17.4)	17.4
cbrt(dp)	dp	Кубический корень	cbrt(27.0)	3
ceil(dp или numeric)	Как у аргумента	Наименьшее целое не меньше аргумента	ceil(-42.8)	-42
ceiling(dp или numeric)	Как у аргумента	Наименьшее целое не меньше аргумента (аналог ceil)	ceiling(-95.3)	-95
degrees(dp)	dp	Перевод радиан в градусы	degrees(0.5)	28.6478897565412
div(y numeric, x numeric)	numeric	Целочисленное деление y/x	div(9,4)	2
exp(dp или numeric)	Как у аргумента	Экспонента	exp(1.0)	2.71828182845905
floor(dp или numeric)	Как у аргумента	Наибольшее целое не больше аргумента	floor(-42.8)	-43
ln(dp или numeric)	Как у аргумента	Натуральный логарифм	ln(2.0)	0.693147180559945
log(dp или numeric)	Как у 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)	Как у аргумента	Остаток от деления y/x	mod(9,4)	1
pi()	dp	Константа π	pi()	3.14159265358979
power(a dp, b dp)	dp	a в степени b	power(9.0, 3.0)	729
power(a numeric, b numeric)	numeric	a в степени b	power(9.0, 3.0)	729
radians(dp)	dp	Перевод градусов в радианы	radians(45.0)	0.785398163397448
random()	dp	Случайное число в диапазоне от 0.0 до 1.0	random()	
round(dp или numeric)	Как у аргумента	Округление до ближайшего целого	round(42.4)	42
round(v numeric, s int)	numeric	Округление до s знаков после запятой	round(42.4382, 2)	42.44

Окончание таблицы 31

Функция	Тип результата	Описание	Пример	Результат
setseed(dp)	void	Установка начального значения для последующих вызовов random() (аргумент -1.0 и 1.0)	setseed(0.54823)	
sign(dp или numeric)	Как у аргумента	Знак аргумента (-1, 0, 1)	sign(-8.4)	-1
sqrt(dp или numeric)	Как у аргумента	Квадратный корень	sqrt(2.0)	1.4142135623731
trunc(dp или numeric)	Как у аргумента	Усечение до целого	trunc(42.8)	42
trunc(v numeric, s int)	numeric	Усечение v до s знаков после запятой	trunc(42.4382, 2)	42.43
width_bucket(op numeric, b1 numeric, b2 numeric, count int)	int	Возвращает участок, в который попадает операнд для равномерной гистограммы с заданным количеством участков и диапазоном значений b1, b2	width_bucket(5.35, 0.024, 10.06, 5)	3
width_bucket(op dp, b1 dp, b2 dp, count int)				

В таблице 32 показаны имеющиеся тригонометрические функции. Все тригонометрические функции имеют тип аргумента и возвращаемого результата double precision.

Таблица 32 – Тригонометрические функции

Функция	Описание
acos(x)	Арккосинус
asin(x)	Арксинус
atan(x)	Арктангенс
atan2(y, x)	Арктангенс отношения x/y
cos(x)	Косинус
cot(x)	Котангенс
sin(x)	Синус
tan(x)	Тангенс

6.4. Строковые функции и операторы

Если не указано другое, то все функции и операторы, перечисленные ниже, работают с любым из типов character, character varying и text, однако, следует быть

внимательными при обработке типа `character` из-за возможного дополнения строки справа пробелами. Некоторые функции работают так же и с битовыми строками.

Стандарт SQL для некоторых строковых функций определяет специальный синтаксис с ключевыми словами вместо запятых, разделяющих аргументы (см. таблицу 33).

Таблица 33 – Строковые функции и операторы SQL

Функция	Тип	Описание	Пример	Результат
`string || string`	`text`	Объединение строк	`'Post' || 'greSQL'`	`PostgreSQL`
`string || не-string` или `не-string || string`	`text`	Объединение строк с одним операндом нестрокового типа	`'Value: ' || 42`	`Value: 42`
`bit_length(string)`	`int`	Число бит в строке	`bit_length('jose')`	`32`
`char_length(string)` или `character_length(string)`	`int`	Число символов в строке	`char_length('jose')`	`4`
`lower(string)`	`text`	Перевод символов строки в нижний регистр	`lower('TOM')`	`tom`
`octet_length(string)`	`int`	Число байт в строке	`octet_length('jose')`	`4`
`overlay(string placing string from int [for int])`	`text`	Замена подстроки	`overlay('Txxxxas' placing 'hom' from 2 for 4)`	`Thomas`
`position(substring in string)`	`int`	Позиция подстроки в строке	`position('om' in 'Thomas')`	`3`
`substring(string [from int] [for int])`	`text`	Выделение подстроки	`substring('Thomas' from 2 for 3)`	`hom`
`substring(string from pattern)`	`text`	Выделение подстроки, соответствующей шаблону (регулярному выражению POSIX). Информация о шаблонах приведена в 6.7	`substring('Thomas' from '...$')`	`mas`
`substring(string from pattern for escape)`	`text`	Выделение подстроки, соответствующей шаблону (регулярному выражению SQL). Информация о шаблонах приведена в 6.7	`substring('Thomas' from '%#"o_a#"_' for '#')`	`oma`
`trim([leading | trailing | both] [characters] from string)`	`text`	Удаление наибольшей подстроки, содержащей только `characters` (по умолчанию — пробелы) с начала, конца, обеих сторон (по умолчанию — с обеих сторон) из строки	`trim(both 'x' from 'xTomxx')`	`Tom`
`upper(string)`	`text`	Перевод символов строки в верхний регистр	`upper('tom')`	`TOM`

Дополнительные строковые функции приведены в таблице 34. Некоторые из них используются для внутренней реализации стандартных строковых SQL-функций, приведенных в таблице 33.

Таблица 34 – Другие строковые функции

Функция	Описание	Тип результата, пример, результат	
`ascii(string)`	ASCII-код первого символа аргумента. Для UTF-8 возвращается код Unicode. Для остальных многобайтных кодировок аргумент должен быть строго ASCII-символом.	Тип	`int`
		Пример	`ascii('x')`
		Результат	`120`
`btrim(string text [, characters text])`	Удаление наибольшей подстроки, содержащей только `characters` (по умолчанию — пробелы) с начала, конца, обеих сторон (по умолчанию — с обеих сторон) из строки.	Тип	`text`
		Пример	`btrim('xyxtrimyyx', 'xy')`
		Результат	`trim`
`chr(int)`	Символ с заданным ASCII-кодом. Для UTF-8 аргумент трактуется как код Unicode. Для остальных многобайтных кодировок аргумент должен строго соответствовать ASCII-символу. Символ NULL (0) не воспринимается, т. к. не может содержаться в строках.	Тип	`text`
		Пример	`chr(65)`
		Результат	`A`
`concat(str "any"[, str "any"[, ...] ])`	Конкатенация текстового представления всех аргументов, NULL-аргументы игнорируются.	Тип	`text`
		Пример	`concat('abcde',2, NULL, 22)`
		Результат	`abcde222`
`concat_ws(sep text, str "any"[, str "any"[, ...] ])`	Конкатенация текстового представления аргументов с разделителем, первый аргумент задает разделитель, NULL-аргументы игнорируются.	Тип	`text`
		Пример	`concat_ws(',', 'abcde', 2,NULL, 22)`
		Результат	`abcde,2,22`
`convert(string bytea, src_encoding name, dest_encoding name)`	Перекодировка строки в кодировку `dest_encoding`. Исходная кодировка указывается в `src_encoding`. Строка должна быть допустима для указанной кодировки. Преобразование может быть определено с помощью `CREATE CONVERSION`. Так же доступны некоторые предопределенные перекодировки (см. таблицу 35).	Тип	`bytea`
		Пример	`convert('text_in_utf8', 'UTF8', 'LATIN1')`
		Результат	`text_in_utf8` в кодировке ISO 8859-1
`convert_from(string bytea, src_encoding name)`	Перекодировка строки в кодировку БД. Исходная кодировка указывается в `src_encoding`. Строка должна быть допустима для указанной кодировки.	Тип	`text`
		Пример	`convert_from('text_in_utf8', 'UTF8')`
		Результат	`text_in_utf8` в кодировке БД

Продолжение таблицы 34

Функция	Описание	Тип результата, пример, результат	
`convert_to(string text, dest_encoding name)`	Перекодировка строки в кодировку `dest_encoding`.	Тип	`bytea`
		Пример	`convert_to('some text', 'UTF8')`
		Результат	`some text` в кодировке UTF-8
`decode(string text, type text)`	Декодирование бинарных данных из строки, закодированной с помощью функции `encode`. Параметр `type` тот же, что и у `encode`.	Тип	`bytea`
		Пример	`decode('MTIzAAE=', 'base64')`
		Результат	`123\000\001`
`encode(data bytea, type text)`	Кодирование бинарных данных в текстовое представление. Поддерживаются типы: `base64`, `hex`, `escape`. `escape` заменяет нулевые символы `\000` и удваивает символы обратной косой черты.	Тип	`text`
		Пример	`encode(E'123\\000\\001', 'base64')`
		Результат	`MTIzAAE=`
`format(formatstr text [, formatarg "any"[, ...] ])`	Форматирование аргументов согласно строке форматирования. Аналог C функции `sprintf` (см. 6.4.1).	Тип	`text`
		Пример	`format('Hello%s, %1$s','World')`
		Результат	`Hello World, World`
`initcap(string)`	Преобразование первого символа каждого слова (слова цифробуквенные, разделены не цифробуквенными символами.)	Тип	`text`
		Пример	`initcap('hi THOMAS')`
		Результат	`Hi Thomas`
`left(str text, n int)`	Возвращает первые `n` символов. При отрицательном значении `n` возвращаются вся строка без последних `|n|` символов.	Тип	`text`
		Пример	`left('abcde',2)`
		Результат	`ab`
`length(string)`	Длина строки в символах.	Тип	`int`
		Пример	`length('jose')`
		Результат	`4`
`length(stringbytea, encoding name)`	Длина строки в символах указанной кодировки. Строка должна быть допустима для указанной кодировки.	Тип	`int`
		Пример	`length('jose', 'UTF8')`
		Результат	`4`

Продолжение таблицы 34

Функция	Описание	Тип результата, пример, результат	
`lpad(string text, length int [, fill text])`	Дополнение строки слева до указанной длины вставкой символов (по умолчанию — пробелов). Если строка уже длиннее заданной длины, то она обрезается справа.	Тип	`text`
		Пример	`lpad('hi', 5, 'xy')`
		Результат	`xyxhi`
`ltrim(string text [, characters text])`	Удаление наибольшей подстроки, состоящей из символов (по умолчанию — пробелов) из строки слева.	Тип	`text`
		Пример	`ltrim('zzzytrim', 'xyz')`
		Результат	`trim`
`md5(string)`	Возвращает в шестнадцатеричном виде вычисленную для строки сумму MD5 (`MD5 hash`).	Тип	`text`
		Пример	`md5('abc')`
		Результат	`900150983cd24fb0 d6963f7d28e17f72`
`pg_client_encoding()`	Текущая кодировка клиента.	Тип	`name`
		Пример	`pg_client_encoding()`
		Результат	`SQL_ASCII`
`quote_ident(string text)`	Заключает в двойные кавычки строку типа `text`, чтобы она могла использоваться как идентификатор в SQL-запросах. Кавычки добавляются только при необходимости. Кавычки внутри строки удваиваются.	Тип	`text`
		Пример	`quote_ident('Foo bar')`
		Результат	`"Foo bar"`
`quote_literal(string text)`	Заключает в двойные кавычки строку типа `text`, чтобы она могла использоваться как строковый литерал в SQL-запросах. Кавычки и символы \ внутри строки удваиваются. `quote_literal` для неопределенного входного значения возвращает тоже неопределенное значение, так что в этом случае предпочтительнее функция `quote_nullable`.	Тип	`text`
		Пример	`quote_literal( 'O\'Reilly')`
		Результат	`'O''Reilly'`
`quote_literal(value anyelement)`	Преобразует входное значение в тип `text`, затем заключает в двойные кавычки, чтобы оно могло использоваться как строковый литерал в SQL-запросах. Кавычки и символы \ удваиваются.	Тип	`text`
		Пример	`quote_literal(42.5)`
		Результат	`'42.5'`
`quote_nullable(string text)`	Заключает в двойные кавычки строку типа `text`, чтобы она могла использоваться как строковый литерал в SQL-запросах. Для неопределенного значения возвращает слово `NULL`. Кавычки и символы \ внутри строки удваиваются.	Тип	`text`
		Пример	`quote_nullable(NULL)`
		Результат	`NULL`

Продолжение таблицы 34

Функция	Описание	Тип результата, пример, результат	
`quote_nullable(value anyelement)`	Преобразует входное значение в тип `text`, затем заключает в двойные кавычки, чтобы оно могло использоваться как строковый литерал в SQL-запросах. Для неопределенного значения возвращает слово `NULL`. Кавычки и символы \ удваиваются.	Тип	`text`
		Пример	`quote_nullable(42.5)`
		Результат	`'42.5'`
`regexp_matches(string text, pattern text [, flags text])`	Возвращает все найденные подстроки в результате поиска по шаблону, заданному регулярным выражением POSIX (см. 6.7.3).	Тип	`setof text[]`
		Пример	`regexp_matches( 'foobarbequebaz', '(bar) (beque)')`
		Результат	`bar,beque`
`regexp_replace(string text, pattern text, replacement text [, flags text])`	Замена подстроки (подстрок) по шаблону, заданному регулярным выражением POSIX (см. 6.7.3).	Тип	`text`
		Пример	`regexp_replace('Thomas', '.[mN]a.', 'M')`
		Результат	`ThM`
`regexp_split_to_array (string text, pattern text [, flags text ])`	Разбивка строки с использованием регулярного выражения POSIX в качестве разделителя (см. 6.7.3).	Тип	`text[]`
		Пример	`regexp_split_to_array( 'hello world', E'\\s+')`
		Результат	`hello,world`
`regexp_split_to_table (string text, pattern text [, flags text])`	Разбивка строки с использованием регулярного выражения POSIX в качестве разделителя (см. 6.7.3).	Тип	`setof text`
		Пример	`regexp_split_to_array( 'hello world', E'\\s+')`

		Результат	`hello`
			`world`
			`(2 rows)`

`repeat(string text, number int)`	Повторение строки заданное число раз.	Тип	`text`
		Пример	`repeat('Pg', 4)`
		Результат	`PgPgPgPg`
`replace(string text, from text, to text)`	Замена всех вхождений одной подстроки другой.	Тип	`setof text`
		Пример	`replace('abcdefabcdef', 'cd', 'XX')`
		Результат	`abXXefabXXef`

Продолжение таблицы 34

Функция	Описание	Тип результата, пример, результат	
`reverse(str)`	Возвращает обратную строку.	Тип	`setof text`
		Пример	`reverse('abcde')`
		Результат	`edcba`
`right(str text, n int)`	Возвращает последние `n` символов. При отрицательной значении `n` возвращается вся строка без первых `|n|` символов.	Тип	`setof text`
		Пример	`right('abcde', 2)`
		Результат	`de`
`rpad(string text, length int [, fill text])`	Дополнение строки справа до длины указанными символами (по умолчанию — пробелами). Если строка уже длиннее заданной длины, то она обрезается справа.	Тип	`text`
		Пример	`rpad('hi', 5, 'xy')`
		Результат	`hixyx`
`rtrim(string text [, characters text])`	Удаление наибольшей подстроки, состоящей из символов (по умолчанию — пробелов) из строки справа	Тип	`text`
		Пример	`rtrim('trimxxxx', 'x')`
		Результат	`trim`
`split_part(string text, delimiter text, field int)`	Разбивает строку по указанному разделителю и возвращает подстроку в указанном по номеру столбце (номера столбцов считаются, начиная с единицы)	Тип	`text`
		Пример	`split_part('abc~@~def~@~ghi', '~@~', 2)`
		Результат	`def`
`strpos(string, substring)`	Позиция подстроки в строке (аналогично `position(substring in string)`, но с другим порядком аргументов)	Тип	`int`
		Пример	`strpos('high', 'ig')`
		Результат	`2`
`substr(string, from [, count])`	Выделение подстроки из строки (аналогично `substring(string from from for count)`)	Тип	`text`
		Пример	`substr('alphabet', 3, 2)`
		Результат	`ph`
`to_ascii(string text [, encoding text])`	Преобразование указанной кодировки в семибитную кодировку ASCII (поддерживаются только кодировки `LATIN1`, `LATIN2` и `WIN1250`)	Тип	`text`
		Пример	`to_ascii('Karel')`
		Результат	`Karel`
`to_hex(number int or bigint)`	Возвращает представление числа в шестнадцатеричной форме	Тип	`text`
		Пример	`to_hex(2147483647)`
		Результат	`7fffffff`

Окончание таблицы 34

Функция	Описание	Тип результата, пример, результат	
`translate(string text, from text, to text)`	Любой символ строки, который имеется в наборе `from`, заменяется соответствующим ему символом из набора `to`	Тип	`text`
		Пример	`translate('12345', '14', 'ax')`
		Результат	`a23x5`

Функции concat, concat_ws и format являются функциями с переменным числом аргументов и могут принимать в качестве аргумента массив, помеченный ключевым словом VARIADIC. Элементы такого массива рассматриваются как набор обычных аргументов для функции. Если в качестве массива передается NULL-значение, функции concat и concat_ws возвращают NULL-значение. Функция format рассматривает NULL-массив как пустой.

Агрегатная функция string_agg описана в 6.20.

Таблица 35 – Предопределенные перекодировки

Имя преобразования *	Исходная кодировка	Целевая кодировка
ascii_to_mic	SQL_ASCII	MULE_INTERNAL
ascii_to_utf8	SQL_ASCII	UTF8
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf8	BIG5	UTF8
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf8	EUC_CN	UTF8
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf8	EUC_JP	UTF8
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf8	EUC_KR	UTF8
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf8	EUC_TW	UTF8
gb18030_to_utf8	GB18030	UTF8
gbk_to_utf8	GBK	UTF8
iso_8859_10_to_utf8	LATIN6	UTF8
iso_8859_13_to_utf8	LATIN7	UTF8
iso_8859_14_to_utf8	LATIN8	UTF8
iso_8859_15_to_utf8	LATIN9	UTF8
iso_8859_16_to_utf8	LATIN10	UTF8
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf8	LATIN1	UTF8
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf8	LATIN2	UTF8
iso_8859_2_to_windows_1250	LATIN2	WIN1250

Продолжение таблицы 35

Имя преобразования *	Исходная кодировка	Целевая кодировка
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	KOI8
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
(a) 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	KOI8
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251

Продолжение таблицы 35

Имя преобразования *	Исходная кодировка	Целевая кодировка
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_gb18030	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
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

Окончание таблицы 35

Имя преобразования *	Исходная кодировка	Целевая кодировка
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	KOI8
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	KOI8
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
ut8_to_euc_jis_2004	UTF8	EUC_JIS_2004
shift_jis_2004_to_utf8	SHIFT_JIS_2004	UTF8
ut8_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

* При именовании перекодировок придерживаются следующей схемы: официальное название исходной кодировки с замененными подчеркиванием не цифробуквенными символами, за которым следует _to_, за которым аналогичным образом следует обработанное официальное название целевой кодировки. Таким образом, полученные имена могут отличаться от обычных имен кодировок

6.4.1. format

Функция format форматирует аргументы в соответствии со строкой форматирования в стиле функции sprintf языка C.

```
format(formatstr text [, formatarg "any" [, ...] ])
```

Аргумент formatstr выступает в роли строки форматирования. Текст строки форматирования копируется непосредственно в результирующую строку, за исключением *спецификаторов форматирования*. Спецификатор форматирования указывает, как должен быть отформатирован и выведен последующий аргумент. Каждый аргумент formatarg конвертируется в текст по обычным правилам для своего типа и выводится в результирующую строку согласно спецификатору форматирования.

Спецификатор форматирования предваряется символом % и имеет следующую форму:

```
%[position][flags][width]type
```

где, полями являются:

- position (необязательное) — строка вида n$, где n является порядковым номером аргумента. 1 означает первый аргумент после строки форматирования formatstr. Если поле position не указано, по умолчанию используется следующий по порядку аргумент.
- flags (необязательное) — дополнительные опции форматирования. В настоящее время поддерживается только знак (-), указывающий левое выравнивание. При отсутствии поля width влияния не оказывает.
- width (необязательное) — задает *минимальную* ширину (количество символов) для вывода. Выводимое значение добивается пробелами слева или справа (в зависимости от флага -) до достижения указанного количества символов. Если заданное значение меньше длины выводимой строки, оно игнорируется, и строка не обрезается. Поле может быть задано следующим образом:
  - целым числом;
  - символом (*), в этом случае в качестве ширины берется значение следующего аргумента;
  - строка вида n$, где n является порядковым номером аргумента, значение которого будет использовано в качестве ширины.

  Если ширина определяется по значению аргумента, этот аргумент вычисляется раньше аргумента, заданного спецификатором форматирования. При отрицательном значении ширины, выводимое значение выравнивается по левому краю (как в присутствии флага -), с шириной |width|.
- type (обязательное) — тип преобразования формата для вывода. Поддерживаются

следующие типы:

- `s` — форматирует значение аргумента как простую строку, NULL-значение считается пустой строкой;
- `I` — рассматривает значение аргумента как SQL идентификатор, заключая его при необходимости в кавычки, генерирует ошибку для NULL-значений;
- `L` — рассматривает значение аргумента как SQL литерал, NULL-значение выводится как строка `NULL` без кавычек.

Помимо рассмотренных выше спецификаторов форматирования, специальная последовательность `%%` может быть использован для вывода символа `%`.

Использование базовых спецификаторов форматирования:

```
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')
```

Использование указания ширины вывода 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       |
```

Использование позиционных спецификаторов форматирования:

```
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 отличии от стандартной С функции `sprintf` функция PostgreSQL `format` позволяет совместно использовать спецификаторы форматирования как с указанием позиции аргумента, так и без. Без указания поля `position` всегда используется аргумент, следующий за последним аргументом, использованным для вывода. Функция `format` не требует использования всех аргументов в строке форматирования, например:

```
SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');
Результат: Testing three, two, three
```

Спецификаторы `%I` и `%L` обычно используются для безопасного создания динамических SQL-запросов.

6.5. Функции и операторы для работы с бинарными строками

Функции для работы с бинарными строками (значение типа `bytea`) приведены в таблице 36.

В стандарте SQL ряд функций работы со строками для разделения аргументов используют ключевые слова вместо запятой (см. таблицу 36). PostgreSQL предоставляет версии этих функций, которые используют обычный синтаксис (см.таблицу 37).

Примечание. Результаты примеров в данном разделе предполагают, что конфигурационный параметр сервера `bytea_output` установлен в значение `escape` (традиционный формат PostgreSQL).

Таблица 36 – Функции и операторы для работы с бинарными строками

Функция	Тип результата	Описание	Пример	Результат
`string || string`	`bytea`	Объединение строк	`E'\\\\Post'::bytea || E'\\047gres\\000'::bytea`	`\\Post'gres\000`
`octet_length(string)`	`int`	Число байт в строке	`octet_length(E'jo\\000se'::bytea)`	`5`

Окончание таблицы 36

Функция	Тип результата	Описание	Пример	Результат
overlay(string placing string from int [for int])	bytea	Замена подстроки	overlay(E'Th\\000omas'::bytea placing E'\\002\\003'::bytea from 2 for 3)	T\\002 \\003mas
position(substring in string)	int	Позиция подстроки в строке	position(E'\\000om'::in E'Th\\000omas'::bytea)	3
substring(string [from int] [for int])	bytea	Выделение подстроки из строки	substring(E'Th\\000omas'::bytea from 2 for 3)	h\000o
trim([both] bytes from string)	bytea	Удаление наибольших подстрок, состоящих из указанных символов с обеих сторон строки	trim(E'\\000'::bytea from E'\\000Tom\\000'::bytea)	Tom

Дополнительные функции приведены в таблице 37. Некоторые из них используются для внутренней реализации функций, приведенных в таблице 36.

Таблица 37 – Дополнительные функции и операторы для работы с бинарными строками

Функция	Тип результата	Описание	Пример	Результат
btrim(string bytea, bytes bytea)	bytea	Удаление наибольших подстрок, состоящих из указанных символов с обеих сторон строки	btrim(E'\\000trim\\000'::bytea, E'\\000'::bytea)	trim
decode(string text, type text)	bytea	Декодирование бинарной строки из семибитного (ASCII) представления (параметр type такой же, как и в функции encode)	decode(E'123\\000456', 'escape')	123\000456
encode(string bytea, type text)	text	Кодирование бинарной строки в семибитное (ASCII) представление (параметр type может быть: base64, hex, escape	encode(E'123\\000456'::bytea, 'escape'))	123\000456
get_bit(string, offset)	int	Получить значение бита в указанной позиции строки	get_bit(E'Th\\000omas'::bytea, 45)	1
get_byte(string, offset)	int	Получить значение байта в указанной позиции строки	get_byte(E'Th\\000omas'::bytea, 4)	109
length(string)	int	Длина бинарной строки	length(E'jo\\000se'::bytea)	5
md5(string)	text	Возвращает в шестнадцатеричном виде вычисленную для бинарной строки сумму MD5 (MD5 hash)	md5(E'Th\\000omas'::bytea)	8ab2d3c9689aaf18b4958c334c82d8b1

Окончание таблицы 37

Функция	Тип результата	Описание	Пример	Результат
set_bit(string, offset, newvalue)	bytea	Установить новое значение бита в указанной позиции строки	set_bit(E'Th\\000omas'::bytea, 45, 0)	Th\000omAs
set_byte(string, offset, newvalue)	bytea	Установить новое значение байта в указанной позиции строки	set_byte(E'Th\\000omas'::bytea, 4, 64)	Th\000o@as

См. также описание агрегирующей функции string_agg в 6.20

6.6. Функции и операторы для работы с битовыми строками

Операторы для работы с битовыми строками (значения типов bit и bit varying) приведены в таблице 38. Они могут быть использованы наряду с обычными операторами сравнения. Операнды для операций &, |, и # должны быть одинаковой длины. При выполнении сдвигов длина строки не меняется, как показано в примерах.

Таблица 38 – Операторы для работы с битовыми строками

Оператор	Описание	Пример	Результат
||	Соединение	B'10001' || B'011'	10001011
&	Побитовое И (AND)	B'10001' & B'01101'	00001
|	Побитовое ИЛИ (OR)	B'10001' | B'01101'	11101
#	Побитовое исключающее ИЛИ (XOR)	B'10001' # B'01101'	11100
~	Побитовое отрицание (NOT)	~B'10001'	01110
«	Битовый сдвиг влево	B'10001' « 3	01000
»	Битовый сдвиг вправо	B'10001' » 2	00100

Следующие стандартные SQL-функции работают с битовыми строками так же, как и с обычными: length, bit_length, octet_length, position, substring, overlay.

Следующие функции работают с битовыми строками так же, как и бинарными строками: get_bit, set_bit. При работе с битовыми строками эти функции рассматривают левый бит строк как бит 0.

Возможно преобразование целочисленных значений в битовые строки и обратно:

```
44::bit(10)                    0000101100
44::bit(3)                     100
cast(-44 as bit(12))           111111010100
'1110'::bit(4)::integer        14
```

Преобразование bit подразумевает преобразование bit(1), при этом используется наименьший значимый двоичный разряд целочисленного значения.

Примечание. При преобразовании integer к bit(n) использовались левые n

бит числа. Сейчас используются правые n бит числа. Так же преобразование целочисленного значения к битовой строке с большим количеством бит осуществляется с распространением знака влево.

6.7. Поиск строк по шаблону

PostgreSQL предоставляет три механизма для работы со строковыми шаблонами: традиционный оператор SQL LIKE, оператор SQL99 SIMILAR TO и шаблоны в стиле регулярных выражений стандарта POSIX. Кроме функций простой формы поиска по шаблону, доступны функции извлечения или замены посдстроки по шаблону и разбиение строки по шаблону.

Примечание. При необходимости иных форм шаблонов рекомендуется применять пользовательские функции, написанные на Perl или Tcl.

6.7.1. LIKE

```
<строка>     LIKE <шаблон> [ESCAPE <спецсимвол>]
<строка> NOT LIKE <шаблон> [ESCAPE <спецсимвол>]
```

Каждый шаблон определяет некоторое множество сопоставимых с ним строк. Выражение LIKE возвращает TRUE, если строка входит в это множество (соответственно, NOT LIKE возвращает для такой строки FALSE, и наоборот. Оно эквивалентно выражению NOT (<строка> LIKE <шаблон>).

Если шаблон не содержит символов процента или подчеркивания, то он представляет собой множество, состоящее из одной этой строки, и в этом случае LIKE работает так же, как оператор равенства. Символ подчеркивания (_) в шаблоне означает, что в этом месте в строке может находиться любой символ; символ процента (%) сопоставляется с подстрокой из нуля и более любых символов. Например:

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

Шаблон LIKE всегда сопоставляется со всей строкой. Чтобы шаблон сопоставлялся с какой-либо частью строки, он всегда должен начинаться и заканчиваться символом процента.

Чтобы временно отменить действие символов процента и подчеркивания, соответствующий символ в шаблоне должен быть предварен спецсимволом. По умолчанию спецсимволом является символ \, однако, можно задать любой другой с помощью конструкции ESCAPE. Чтобы временно отменить действие спецсимвола, он должен быть удвоен.

Примечание. В случае установки конфигурационного параметра standard_conforming_strings в значение off, любой символ \ должен быть задвоен (см. 1.1.2.1).

Возможно так же отказаться от использования спецсимвола вообще, записав ESCAPE ''. Это блокирует работу спецсимвола, что делает невозможным выключение действия символов процента и подчеркивания.

Ключевое слово ILIKE может быть использовано вместо LIKE, чтобы сделать работу шаблона независящей от регистра (в соответствии с правилами текущей кодировки сервера). Этого ключевого слова нет в SQL-стандарте.

Оператор ~~ эквивалентен LIKE, а ~~* — ILIKE. Соответственно, существуют обратные операторы !~~ и !~~*. Они так же являются специфичными для PostgreSQL.

6.7.2. 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 совпадения строк по шаблону, что по действию аналогично LIKE и отличается от него только правилами задания шаблона (соответствующими правилам задания регулярных выражений стандарта SQL). Синтаксис регулярных выражений SQL является смесью правил LIKE и общих правил задания регулярных выражений.

Подобно LIKE оператор SIMILAR TO требует, чтобы шаблон сопоставлялся со всей заданной строкой, в отличие от общей практики использования регулярных выражений, в которой шаблон сопоставляется с любой ее подстрокой. Как и LIKE, он использует символы процента и подчеркивания как специальные символы, соответствующие любой подстроке или любому символу, соответственно (подобно действию . и .* в регулярных выражения POSIX).

Кроме возможностей, наследованных от LIKE, оператор SIMILAR TO поддерживает следующие возможности регулярных выражений POSIX:

- | — выбор одной из двух альтернатив;
- * — строка, состоящая из нуля и более повторений предшествующего элемента;
- + — строка, состоящая из одного и более повторений предшествующего элемента;
- ? — строка, состоящая из нуля или одного повторений предшествующего элемента;
- {m} — повторение предшествующего элемента ровно m раз;
- {m,} — повторение предшествующего элемента m и более раз;
- {m,n} — повторение предшествующего элемента не менее m, но не более n раз;
- () — группировка элементов шаблона в единый логический элемент;
- [...] — символьный класс, как в регулярных выражениях POSIX.

Примечание. Точка (.) не является метасимволом в выражениях SIMILAR TO.

Так же как и для LIKE спецсимвол, заданный конструкцией ESCAPE (по умолчанию — \), временно отменяет действие спецсимволов.

Некоторые примеры:

```
'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>)
```

выполняет выделение подстроки из строки в соответствии с заданным <шаблоном>. Так же как и для SIMILAR TO, шаблон сопоставляется со всей строкой, и при неудаче сопоставления функция возвращает неопределенное значение. Для указания части шаблона, которая задает substring возвращаемую подстроку, SQL99 определяет специальный маркер, состоящий из заданного спецсимвола и следующего за ним символа двойной кавычки ("). Подстрока, соответствующая части шаблона между двумя маркерами, возвращается как результат вызова функции.

Примеры с #", разделяющим входную строку:

```
substring('foobar' from '%#"o_b#"%' for '#')   oob
substring('foobar' from '#"o_b#"%' for '#')    NULL
```

6.7.3. Регулярные выражения POSIX

В таблице 39 приведены операторы, использующие шаблоны, записанные по правилам POSIX.

Таблица 39 – Операторы сопоставления регулярный выражений

Оператор	Описание	Пример
~	Совпадение с регулярным выражением, с учетом регистра	'thomas' ~'.*thomas.*'
~*	Совпадение с регулярным выражением, без учета регистра	'thomas' ~* '.*Thomas.*'
!~	Несовпадение с регулярным выражением, с учетом регистра	'thomas' !~'.*Thomas.*'
!~*	Несовпадение с регулярным выражением, без учета регистра	'thomas' !~* '.*vadim.*'

Регулярные выражения POSIX предоставляют более богатые возможности по описанию шаблонов, чем операторы LIKE и SIMILAR TO. Многие утилиты Unix, такие как egrep, sed или awk, используют подобный синтаксис описания шаблонов.

Регулярным выражением называется последовательность символов, описывающая множество из одной и более строк. Строка считается сопоставимой с регулярным выражением, если она принадлежит множеству описываемых им строк. Так же как для оператора LIKE, любые символы шаблона представляют собой обычные символы, если только они не

имеют специального значения в языке описания регулярных выражений. Однако регулярные выражения POSIX используют свой набор специальных символов. В отличие от операторов LIKE регулярное выражение может совпадать с любым местом строки, если специальным образом не указано, что строка должна начинаться и/или заканчиваться этим регулярным выражением.

Примеры:

```
'abc' ~ 'abc'    true
'abc' ~ '^a'     true
'abc' ~ '(b|d)'  true
'abc' ~ '^(b|c)' false
```

Функция substring с двумя параметрами:

```
substring(<строка> FROM <шаблон>)
```

выполняет выделение подстроки из строки в соответствии с шаблоном POSIX. Она возвращает неопределенное значение, если строка не сопоставима с шаблоном, иначе возвращается та часть строки, которая была с ним сопоставлена. Если шаблон содержит круглые скобки, возвращается подстрока, сопоставленная с частью шаблона в первых (самых левых) круглых скобках верхнего уровня вложенности. Можно заключить весь шаблон в круглые скобки, если необходимо предотвратить подобное действие круглых скобок внутри шаблона.

Примеры:

```
substring('foobar' from 'o.b')     oob
substring('foobar' from 'o(.)b')   o
```

Функция regexp_replace выполняет замену новым тестом подстроки, найденной в соответствии с регулярным выражением POSIX, и имеет следующий синтаксис вызова:

```
regexp_replace(source, pattern, replacement [, flags ])
```

Если не было выявлено ни одного совпадения с шаблоном, возвращается исходная строка без изменений. В случае обнаружения совпадения с шаблоном возвращается исходная строка, в которой часть строки, совпадающая с шаблоном, заменена указанной подстрокой. Заменяющая подстрока может содержать \n, где n может принимать значение от 1 до 9, которая показывает, что в это место должна быть вставлена часть шаблона, выделенная скобками соответствующего уровня вложенности, и может содержать \&, показывающий, что должна быть вставлена вся часть строки, совпадающая с шаблоном. Для использования символа \ в заменяющей строке, он должен быть удвоен. (Необходимо всегда помнить, что при использовании символа \ в строковых литералах, требуется его удвоение.) Параметр flags может содержать нуль или более однобуквенных флагов, влияющих на поведение функции. Флаг i обозначает игнорирование регистра символов, флаг g — обработку всех вхождений, а не только первого. Остальные флаги описаны в таблице 47.

Примеры:

```
regexp_replace('foobarbaz', 'b..', 'X')
                                    fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                    fooXX
regexp_replace('foobarbaz', 'b(..)', E'X\\1Y', 'g')
                                    fooXarYXazY
```

Функция возвращает все найденные подстроки, совпадающие с заданным регулярным выражением POSIX-шаблона.

Функция `regexp_matches` возвращает все подстроки, найденных в исходной строке в соответствии с регулярным выражением POSIX, и имеет следующий синтаксис вызова:

```
regexp_matches(string, pattern [, flags ])
```

Функция может вернуть пустой набор строк, одну строку или несколько строк (см. флаг `g` далее). Если не было выявлено ни одного совпадения с шаблоном, функция возвращает пустой набор строк. Если шаблон не содержит выделенных скобками частей, результат представляет собой для каждой строки текстовый массив, состоящий из одного элемента, содержащем совпадающую с шаблоном подстроку. Если шаблон содержит выделенные скобками части, функция возвращает текстовый массив, в котором `n`-ый элемент является подстрокой, совпадающей с `n`-ой частью шаблона (без учета «невыбираемых частей»). Параметр `flags` может содержать нуль или более однобуквенных флагов, влияющих на поведение функции. Флаг флаг `g` означает обработку всех вхождений, а не только первого. Остальные флаги описаны в таблице 47.

Примеры:

```
SELECT regexp_matches('foobarbequebaz', '(bar)(beque)');
 regexp_matches
----------------
 {bar,beque}
(1 row)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
 regexp_matches
----------------
 {bar,beque}
 {bazil,barf}
(2 rows)

SELECT regexp_matches('foobarbequebaz', 'barbeque');
 regexp_matches
```

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

Существует возможность принудить `regexp_matches()` всегда возвращать одну строку путем использования подзапроса, что обычно удобно в списке `SELECT`, когда требуется вернуть все строки, даже не совпадающие с шаблоном:

```
SELECT col1, (SELECT regexp_matches(col2, '(bar)(beque)')) FROM tab;
```

Функция `regexp_split_to_table` разбивает строку, используя в качестве разделителя регулярное выражение POSIX, и имеет следующий синтаксис вызова:

```
regexp_split_to_table(string, pattern [, flags ])
```

Если не было выявлено ни одного совпадения с шаблоном, возвращается исходная строка без изменений. В случае обнаружения совпадения с шаблоном возвращается текст, заключенный между предыдущим найденным совпадением с шаблоном (или началом строки) и до начала текущего совпадения. Последним возвращается тест от последнего совпадения с шаблоном и концом строки. Параметр `flags` может содержать нуль или более однобуквенных флагов, влияющих на поведение функции. Доступные для использования при вызове указанной функции флаги описаны в таблице 47.

Функция `regexp_split_to_array` действует аналогично функции `regexp_split_to_table` и отличается только тем, что возвращает вместо таблицы текстовый массив. Синтаксис вызова и доступные параметры те же: `regexp_split_to_array(string, pattern [, flags ])`.

Примеры:

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumped over the
  lazy dog', E'\\s+') AS foo;
  foo
--------
 the
 quick
 brown
 fox
 jumped
 over
 the
 lazy
 dog
(9 rows)

SELECT regexp_split_to_array('the quick brown fox jumped over the lazy dog',
```

```
  E'\\s+');
              regexp_split_to_array
-------------------------------------------------
 {the,quick,brown,fox,jumped,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)
```

Последний пример показывает, что функции `regex_split` игнорируют текст нулевой длины, который возникает между началом, концом строки и совпадением с шаблоном, а так же возникающий между шаблонами. Это отличается от принятого для функции `regexp_matching`, но более соответствует ожиданиям. Другие программные средства, такие как Perl, используют похожее определение.

6.7.3.1. Использование регулярных выражений

Регулярные выражения (RE), определенные стандартом POSIX 1003.2, существуют в двух формах: расширенные ERE (от англ. «extended») и базовые RE или BRE (от англ. «basic»). PostgreSQL поддерживает использование обеих форм, а так же реализует некоторые расширения, не входящие в стандарт POSIX, но широко распространенные благодаря их доступности в языках Perl и Tcl. Подобные RE используют несовместимые с

POSIX расширения, называемые «продвинутыми» ARE (от англ. «advanced»).Форма ARE практически является надмножеством формы ERE, но форма BRE содержит некоторую несовместимость по нотации (и является более ограниченной).

Примечание. PostgreSQL по умолчанию всегда считает, что регулярные выражения следуют правилам ARE. Более ограниченные правила ERE или BRE могут быть выбраны с помощью встроенной опции шаблон регулярный выражений, как описано в 6.7.3.4, что может быть полезно для совместимости с приложениями, строго следующими правилам стандарта POSIX 1003.2.

Регулярное выражение представляет собой одно или более непустых *ветвлений*, разделенных символом вертикальной черты `|`. Оно сопоставимо с любой строкой, с которой сопоставимо хотя бы одно из ветвлений.

Ветвление представляет собой нуль или более *связанных участков* (атомов) или *ограничителей* позиции. Для его сопоставимости со строкой необходима сопоставимость с ней всей последовательности участков одного за другим. Пустое ветвление означает пустую строку.

Атом — это неделимый *элемент*, за которым следует одиночный *квантификатор*. Без квантификатора подразумевается просто совпадение с атомом. С помощью квантификатора указывается количество повторений совпадений с атомом. Возможные виды атомов приведены в таблице 40. Доступные квантификаторы и их значения приведены в таблице 41.

Таблица 40 – Атомы регулярных выражений

Атом	Описание
`(re)`	Атом, в котором `re` — любое регулярное выражение подразумевает совпадение с отметкой (параметризацией) для последующего использования
`(?:re)`	Аналогично предыдущему, но без возможности последующего использования (только для ARE)
`.`	Любой одиночный символ
`[chars]`	Символьный класс — набор символов, подразумевает совпадение с любым из символов (см. 6.7.3.2)
`\k`	Атом, в котором `k` не является алфавитно-цифровым символом, подразумевает одиночный символ, например `\\` — символ обратной косой черты
`\c`	Атом, в котором `c` является экранированным алфавитно-цифровым символом (после которого возможно следуют другие), см. 6.7.3.3. (Только для ARE, в ERE и BRE означает просто `c`)
`{`	Когда за ним следует символ, отличный от цифрового, означает просто символ `{`, в противном случае — начало квантификатора
`x`	Атом, в котором `x` — одиночный символ, означает сам символ

Регулярное выражение не может заканчиваться символом обратной косой черты `\`.

Примечание. В случае установки конфигурационного параметра `standard_conforming_strings` в значение `off`, любой символ \ должен быть задвоен (см. 1.1.2.1).

Таблица 41 – Квантификаторы регулярных выражений

Квантификатор	Значение
*	Последовательность от нуля и более вхождений атома
+	Последовательность от одного и более вхождений атома
?	Последовательность от нуля и до одного вхождения атома
{m}	Последовательность m вхождений атома
{m,}	Последовательность m и более вхождений атома
{m,n}	Последовательность от m и до n вхождений атома включительно, m не может превышать 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}

Формы, использующие `{...}` называются пределами. Числа `m` и `n`, используемые в пределах, должны быть целыми положительными от 0 до 255 включительно.

«Не жадные» квантификаторы (доступные только в ARE) обеспечивают те же возможности, что и обычные, но предпочитают меньшее количество совпадений. Подробности приведены в 6.7.3.5.

Примечание. Квантификаторы не могут следовать подряд. Так же они не могут начинаться с выражения или следовать за спецсимволами ^ и |.

Маркеры позиционирования соответствуют пустой строке, но только при определенных условиях. Маркеры позиционирования могут использоваться там же, где и допустимо использование атомов, но для них не может быть указан квантификатор. Простые маркеры позиционирования приведены в таблице 42.

Таблица 42 – Маркеры позиционирования (ограничения) регулярных выражений

Ограничитель	Описание
^	Начало строки
$	Конец строки
(?=re)	Совпадение при предварительном просмотре в любом месте, начинающемся с выражения `re` (только ARE)

Окончание таблицы 42

Ограничитель	Описание
`(?!re)`	Совпадение при предварительном просмотре в любом месте, не начинающемся с выражения `re` (только ARE)

Выражения для предварительного просмотра не могут содержать ссылку `back` (см. 6.7.3.3), и все подвыражения в них рассматриваются без возможности дальнейшего использования.

6.7.3.2. Выражения в скобках

Выражение в скобках — список символов, заключенных в `[]`. Обычно оно сопоставляется с любым единичным символом из этого списка. Если список начинается с символа `^`, то выражение сопоставляется с любым единичным символом, которого в нем нет. Если два символа в списке разделены символом `-`, то это рассматривается как сокращенная запись списка из всех символов от первого до второго включительно в порядке сортировки символов (например, выражение `[0-9]` в кодировке ASCII представляет собой любую десятичную цифру). Два таких диапазона не могут разделять одну конечную точку. Следовательно, запись `a-c-e` недопустима. Задание диапазона символов зависит от порядка их сортировки, поэтому не рекомендуется использовать их в портируемых программах.

Чтобы включить в список символ `]`, необходимо записать его первым в списке (возможно после символа `^`). Чтобы включить символ `-`, необходимо записать его либо первым, либо последним в списке, либо второй конечной точкой диапазона символов. Чтобы использовать символ `-` как первую точку диапазона, необходимо заключить его в маркеры `[.` и `.]`, чтобы представить его как сопоставимый элемент. За исключением этого и нескольких комбинаций, использующих символ `[` и управляющие последовательности (только в ARE), все другие спецсимволы теряют свое специальное значение в выражениях в квадратных скобках. В частности, символ `\`, который не является спецсимволом в ERE и BRE, но является спецсимволом экранирования в форме ARE.

Сопоставимый элемент — это последовательность из одного и более символов, которые описывают в текущей кодировке единичный символ. Для того чтобы такой символ рассматривался анализатором регулярных выражений как единичный, он должен быть заключен в маркеры `[.` и `.]`. После этого последовательность в выражении в квадратных скобках рассматривается как единичный символ. Например, если в текущей кодировке имеется символ, обозначаемый последовательностью `ch`, то регулярное выражение `[[.ch.]]*c` будет сопоставлено с первыми пятью символами строки `chchcc`.

Примечание. PostgreSQL в настоящее время не поддерживает многосимвольных сопоставимых элементов. Приведенная информация описывает возможное будущее поведение.

Сопоставимый элемент, заключенный в маркеры [= и =], представляет собой список из всех символов, входящих в *класс эквивалентности* в текущей кодировке, включая сам сопоставимый элемент. Если для этого сопоставимого элемента нет эквивалентных сопоставимых элементов, то такая конструкция аналогична использованию маркеров [. и .]. Например, если o и ^ являются элементами одного класса эквивалентности, то [[=o=]], [[=^=]] и [o^] являются синонимами. Класс эквивалентности не может быть использован как конечная точка диапазона символов.

Внутри выражения в скобах для указания всех символов одного класса может быть использовано имя класса, обрамленное маркерами [: и :]. Существуют следующие имена стандартных классов: alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit. Эти имена соответствуют именам классов, объявленным в ctype. Локаль может предоставлять иные. Класс символов не может быть использован как конечная точка диапазона символов.

Для регулярных выражений POSIX есть два специальных случая выражений в квадратных скобках: [[:<:]] и [[:>:]], сопоставляемые с пустой строкой в начале и в конце слова, соответственно. При этом слово определяется как последовательность из символов слов, перед которой и после которой таких символов нет, другими словами — максимально возможная их последовательность. Множество символов слов включает в себя символы класса alnum и символ подчеркивания _. Выражения в квадратных скобках являются расширением стандарта POSIX, и их следует использовать с осторожностью в программах, предназначенных для портирования на другие системы.

6.7.3.3. Управляющие последовательности в регулярных выражениях

Управляющие последовательности представляют собой специальные последовательности символов, начинающиеся с символа обратной косой черты \, за которым следует одиночный алфавитно-цифровой символ. Управляющие последовательности представлены в нескольких вариантах: экранированный символ, сокращенное именование класса символов, маркеры позиционирования и обратные ссылки. Символ обратной косой черты \, за которым следует алфавитно-цифровой символ, не соответствующий ни одной допустимой управляющей последовательности в форме ARE является недопустимым выражением. В форме ERE нет управляющих последовательностей: вне выражений в квадратных скобках, символ обратной косой черты \c последующим алфавитно-цифровым символом представляет просто сам символ, внутри же таких выражений, \ представляет сам символ обратной косой черты. (Это является основным отличием между формами ERE и ARE.)

Вариант управляющей последовательности *экранированный символ* введен для возможности представление непечатных или иных недопустимых символов в регулярных выражениях (смотри таблицу 43).

Вариант управляющей последовательности *сокращенное именование класса символов* предоставляет возможность краткой записи классов символов. Список сокращений классов символов приведен в таблице 44.

Вариант управляющей последовательности *маркер позиционирования* является вариантом маркера позиционирования, записанного в виде управляющей последовательности. Список доступных маркеров позиционирования приведен в таблице ‘45.

Вариант управляющей последовательности *обратная ссылка* (\n) сопоставляет строку предыдущего подвыражения ветвления, определяемую по номеру n, как показано в таблице 46). Например, ([bc])\1 сопоставляется с bb или cc, но не с bc или cb. В регулярном выражении подвыражение ветвления должно целиком предшествовать back ссылке. Подвыражения нумеруются в порядке их группировки скобками. Незахватываемые выражения в скобках не определяются как подвыражения.

Примечание. Необходимо помнить, что при указания шаблона в виде SQL констант символ обратной косой черты
должен быть удвоен.

Пример

```
’123’ ~ E’^\\d{3}’true
```

Таблица 43 – Экранированные символы в регулярных выражениях

Последовательность	Описание
\a	Звуковой сигнал (bell), как в C
\b	Символ забоя, как в C
\B	Синоним для символа обратной косой черты, для использования вместо его удвоения
\cX	X любой символ, в котором 5 младших бит являются битами символа в позиции X, а остальные обнулены
\e	Символ соответствующий наименованию ESC, восьмеричному значению 033
\f	Подача бумаги, как в C
\n	Новая строка, как в C
\r	Возврат каретки, как в C
\t	Горизонтальная табуляция, как в C
\uwxyz	Последовательность, в которой wxyz четыре шестнадцатеричных цифры, представляющая собой символ UTF16 (Unicode, 16-bit) U+wxyz в локальной кодировке
\Ustuvwxyz	Последовательность, в которой stuvwxyz восемь шестнадцатеричных цифр, зарезервированная для гипотетического расширения Unicode до 32 бит
\v	Вертикальная табуляция, как в C

Окончание таблицы 43

Последовательность	Описание
\xhhh	Последовательность, в которой hhh любые шестнадцатеричные цифры, представляющая собой символ с шестнадцатеричным значением 0xhhh (один символ, не зависимо от того, сколько цифр указано)
\0	Символ с кодом 0
\xy	Последовательность, в которой xy две восьмеричных цифры не являющиеся обратной ссылкой, представляющая собой символ с восьмеричным значением 0xy
\xyz	Последовательность, в которой xyz три восьмеричных цифры, не являющиеся обратной ссылкой, представляющая собой символ с восьмеричным значением 0xyz

Шестнадцатеричные цифры — 0–9, a–f, и A–F. Восьмеричные цифры 0–7.

Управляющая последовательность экранированный символ всегда воспринимаются как одиночный символ. Например, \135 это] в кодировке ASCII, но \135 не завершает выражение в квадратных скобках.

Таблица 44 – Экранированные символы в регулярных выражениях

Последовательность	Описание
\d	[[:digit:]]
\s	[[:space:]]
\w	[[:alnum:]_] (включая подчеркивание)
\D	[^[:digit:]]
\S	[^[:space:]]
\W	[^[:alnum:]_] (включая подчеркивание)

Внутри выражений в квадратных скобках, \d, \s, и \w теряют свои внешние скобки, а \D, \S, и \W являются недопустимыми. Таким образом, [a-c\d] эквивалентно [a-c[:digit:]], а [a-c\D] эквивалентное [a-c^[:digit:]], считается недопустимым.

Таблица 45 – Экранированные символы в регулярных выражениях

1 Последовательность	Описание
\A	Сопоставляется с началом строки. Дополнительная информация об отличии от ^ приведена в 6.7.3.5
\m	Сопоставляется с началом слова
\M	Сопоставляется с концом слова
\y	Сопоставляется с началом или концом слова
\Y	Сопоставляется с символом который не является началом или концом слова

Окончание таблицы 45

Последовательность	Описание
\Z	Сопоставляется с концом сроки. Дополнительная информация об отличии от $ приведена в 6.7.3.5

Слово определяется как было описано выше в спецификации [[:<:]] и [[:>:]]. Маркеры позиционирования не могут встречаться внутри выражений в квадратных скобках.

Таблица 46 – Экранированные символы в регулярных выражениях

Последовательность	Описание
\m	Обратная ссылка на m-ое подвыражение, где m ненулевая цифра
\mnn	Обратная ссылка на mnn-ое подвыражение, где m ненулевая цифра, а nn следующие цифры, и при этом десятичное значение mnn не более количества захваченных подвыражений

Примечание: Существует унаследованная историческая неопределенность между восьмеричными управляющими последовательностями и обратными ссылками, разрешаемая эвристическим методом, как было показано выше. Идущий первым нуль всегда обозначает восьмеричную последовательность. Мультицифровая последовательность, не начинающаяся с нуля, воспринимается как обратная ссылка, если она идет за подходящим подвыражением, Таким образом указанный номер возможен для ссылки, в противном случае он воспринимается как восьмеричная последовательность.

6.7.3.4. Метасинтаксис регулярных выражений

В дополнение к описанному выше основному синтаксису регулярных выражений существует ряд специальных форм и разнообразных синтаксических возможностей.

Регулярное выражение может начинаться с одного из двух специальных *управляющих* префиксов. Если регулярное выражение начинается с ***:, остаток регулярного выражения воспринимается в форме ARE. (Обычно в PostgreSQL этот префикс не оказывает действия, поскольку регулярные выражения по умолчанию считаются ARE, но может оказывать действие если с помощью параметров функции регулярных выражений были выбраны режимы ERE или BRE.) Если регулярное выражение начинается с ***=, остаток регулярного выражения воспринимается как литеральная строка, с которой все символы рассматриваются как отдельные.

Регулярное выражение в форме ARE может начинаться с вложенных опций: последовательность (?xyz), в которой xyz один или более алфавитно-цифровых символов, задает опции, влияющие на разбор оставшейся части регулярного выражения. Эти опции переопределяют любые предыдущие опции, включая определение формы регулярных выражения и чувствительность к регистру символов. Доступные символы опций приведены в

таблице 47.

Таблица 47 – Экранированные символы в регулярных выражениях

Опция	Описание
b	Остаток регулярного выражения рассматривается в форме BRE.
c	Сопоставление, чувствительное к регистру символов (overrides operator type).
e	Остаток регулярного выражения рассматривается в форме ERE.
i	Сопоставление, нечувствительное к регистру символов (смотри 6.7.3.5) (перегружает тип оператора).
m	Исторический синоним для n.
n	Сопоставление, чувствительное к переводу строки(смотри 6.7.3.5).
p	Сопоставление, частично чувствительное к переводу строки(смотри 6.7.3.5).
q	Остаток регулярного выражения рассматривается как литеральная строка обычных символов, заключенная в двойные кавычки.
s	Сопоставление, нечувствительное к переводу строки.
t	Компактный синтаксис (используется по умолчанию; дополнительная информация приведена далее в настоящем параграфе).
w	Сопоставление, обратно частично чувствительное к переводу строки(смотри 6.7.3.5).
x	Расширенный синтаксис (дополнительная информация приведена далее в настоящем параграфе).

Вложенные опции действуют до закрывающийся скобки), завершающей выражение. Они могут указываться только в начале ARE выражений после ***:.

В дополнение к обычному (компактному) синтаксису регулярных выражений, в котором все символы являются значимыми, существует расширенный синтаксис, доступный при указании вложенной опции x. В расширенном синтаксисе пробельные символы в регулярном выражении игнорируются, так же как и символы, расположенные между символом # и последующим символом перевода строки (или до конца регулярного выражения). Это обеспечивает возможность использования наглядного форматирования и комментирования сложных регулярных выражений. Существует три исключения из этого простого правила:

- пробельный символ или символ #, следующий за символом обратной косой черты \ сохраняется;
- пробельный символ или символ #, внутри выражения в квадратных скобках сохраняется;
- Пробельные символы и комментарии не могут присутствовать в мультисимвольных выражениях таких как (?:.

В качестве пробельных символов рассматриваются пробел, табуляция, перевод

строки и другие символы, принадлежащие классу символов `space`.

В заключение, в форме ARE, вне выражений в квадратных скобках, последовательность `(?#ttt)`, в которой `ttt` любой текст, не содержащий символа закрывающейся скобки `)`, является комментарием и полностью игнорируется. С другой стороны, это недопустимо в мультисимвольных конструкциях типа `(?:`. Подобные комментарии являются устаревшими, и вместо них рекомендуется использовать расширенный синтаксис.

Никакие из указанных метасинтаксических расширений недоступны при указании префикса `***=`, который говорит о том, что строка символов должна восприниматься как литеральная строка, а не как регулярное выражение.

6.7.3.5. Правила сопоставления регулярных выражений

В случае, когда регулярное выражение может быть сопоставлено более чем с одной подстрокой в заданной строке, оно считается сопоставленным с самой первой подстрокой. Если регулярное выражение может быть сопоставлено с несколькими подстроками, начинающимися в одной и той же позиции, то оно считается сопоставленным с самой длинной или с самой короткой подстрокой, в зависимости от того является ли регулярное выражение жадным или нет.

Является регулярное выражение жадным или нет, определяется следующими правилами:

- большинство атомов, и все маркеры позиционирования не обладают атрибутом жадности (поскольку не могут сопоставляться переменное количество раз в тексте);
- заключение в скобки регулярного выражения не меняет его жадности;
- квантифицированный атом с указанным фиксированным количеством повторений (`{m}` или `{m}?`) обладает такой же жадностью, что и сам атом;
- квантифицированный атом с иными нормальными кванитификаторами (включая `{m, n}`, где `m = n`) является жадным (предпочитает длинное сопоставление);
- квантифицированный атом с не жадными кванитификаторами (включая `{m, n}?`, где `m = n`) не является жадным (предпочитает короткое сопоставление);
- ветвление, такое что регулярное выражение не имеет оператора `|` верхнего уровня, имеет такую же жадность, что и первый атом, который обладает атрибутом жадности;
- регулярное выражение, состоящее из двух или более ветвлений, соединенных оператором |, всегда является жадным.

Перечисленные выше правила ассоциируют атрибуты жадности не только отдельным атомам, но и ветвлениям и всему регулярному выражению, содержащему квантифицированные атомы. Таким образом сопоставление существует в том случае, когда ветвление или все регулярное выражение целиком сопоставляется с наибольшей или наименьшей возможной подстрокой в целом. Как только определена длина всего сопоставления, часть

сопоставления, которая относится к каждому конкретному подвыражению, определяется, исходя из атрибутов жадности этого подвыражения таким образом, что начинающиеся ранее подвыражения регулярного выражения имеют приоритет перед подвыражениями, начинающимися позднее.

Пример

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
Результат: 123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
Результат: 1
```

В первом случае регулярное выражение целиком жадное, поскольку атом `Y*` жадный. Следовательно, с позиции `Y` выбирается наиболее длинная возможная подстрока, которой является `Y123`. Предназначенная для вывода часть определяется круглыми скобками, или `123`. Во втором случае регулярное выражение не является жадным, так как и `Y*?` не является жадным. Следовательно, начиная с Y выбирается наиболее короткая возможная подстрока `Y1`. Подвыражение `[0-9]{1,3}` жадное, но не может изменить решения по выбору общей длины сопоставления, так что принудительно ограничивается сопоставлением с `1`.

Если регулярное выражение содержит одновременно жадные и нежадные подвыражения, общая длина сопоставления выбирается максимально возможной или минимально возможной, на основании атрибута жадности, ассоциированного со всем регулярным выражением. Атрибуты ассоциированные непосредственно с подвыражениями влияют только на количество их сопоставлений по отношению друг к другу.

Квантификаторы `{1,1}` и `{1,1}?` могут быть использованы для принудительного задания подвыражению или всему регулярному выражению атрибута жадности или не жадности соответственно.

Длина сопоставления измеряется в символах, а не сопоставимых элементах. Пустая строка считается длиннее, чем полное отсутствие сопоставлений. Например, регулярное выражение `bb*` сопоставится с тремя средними символами строки `abbbc`, а `(wee|week)(knights|nights)` со всеми десятью символами строки `weeknights`. Если попытаться сопоставить выражение `(.*).*` со строкой `abc`, то подвыражение в скобках будет сопоставлено со всей строкой. Если попытаться сопоставить со строкой `bc` выражение `(a*)*` то с ней не будет сопоставлено ни подвыражение, ни целое выражение. Следовательно, подвыражению в скобках будет сопоставлена пустая строка.

При выполнении сопоставления в режиме игнорирования регистра символов каждый символ, имеющий соответствующие ему символы в других регистрах, вне выражения в квадратных скобках трансформируется в выражение в квадратных скобках, содержащее

все возможные его формы. Например, символ `x` трансформируется в `[xX]`. Когда такой символ появляется в выражении в квадратных скобках, то в это выражение будут добавлены все соответствующие ему формы в других регистрах, таким образом, что, например, `[x]` трансформируется в `[xX]`, а `[^x]` трансформируется в `[^xX]`.

При выполнении сопоставления в режиме чувствительности к переводу строки `.` и выражения в скобках, использование `^` не сопоставляется с символом новой строки. Таким образом, сопоставление не может пересекать границы строк, если это не задано явно в регулярном выражении. Следовательно маркеры позиционирования `^` и `$`, в дополнении к сопоставлению с началом и концом строки, сопоставляются с пустой строкой после и до перевода строки соответственно. При этом в форме ARE управляющие последовательности `\A` и `\Z` продолжают сопоставляться только с началом и концом строки.

При выполнении сопоставления в режиме частичной чувствительности к переводу строки `.` выражения в скобках действуют как и в случае сопоставления в режиме чувствительности к переводу строки, тогда как маркеры позиционирования `^` и `$` нет.

При выполнении сопоставления в режиме обратной частичной чувствительности к переводу строки, маркеры позиционирования `^` и `$` действуют как и в случае сопоставления в режиме чувствительности к переводу строки, тогда как `.` и выражения в скобках нет. Данный вариант реализован для симметричности.

6.7.3.6. Ограничения и совместимость

На длину регулярный выражений в данной реализации не накладывается никаких ограничений. В тоже время, программы, предназначенные для портирования на другие системы, не должны использовать регулярные выражений длиннее 256 символов, так как POSIX-совместимые реализации могут отклонить подобные выражения.

Единственное свойство формы ARE, действительно не соответствующее форме ERE POSIX, заключается в том, что символ обратной косой черты `\` не теряет своего специального значения внутри выражений в квадратных скобках. Остальные свойства формы ARE используют синтаксис, который является недопустимым или имеет неопределенные и непредсказуемые проявления в форме ERE POSIX. Синтаксис префиксов `***` одинаково не соответствует синтаксису POSIX ARE и ERE.

Множество расширений формы ARE были позаимствованы из Perl, но некоторые изменены для большего соответствия, а часть расширений Perl не представлена вообще. Несовместимость нотации включает `\b`, `\B`, отсутствие специальной обработки завершающий переводов строки, дополнение выражений в квадратных скобках для возможности чувствительного к переводу строки сопоставления, ограничения по использованию подвыражений и back ссылок при предварительном просмотре выражения, и семантика сопоставления с выбором наиболее длинного или наиболее короткого варианта.

Существует два важных различия между синтаксисом форм ARE и ERE в версиях PostgreSQL до 7.4:

- В форме ARE, обратная косая черта \ с последующим алфавитно-цифровым символом является или управляющей последовательностью или ошибкой, тогда как ранее рассматривалась как еще одна форма записи алфавитно-цифрового символа. Это не должно составлять проблемы, поскольку не существует причин использовать подобные последовательности в ранних версиях;
- В форме ARE, обратная косая черта \ является спецсимволом внутри выражений в квадратных скобках [], так что литерал \ в подобных выражениях должен указываться задвоенным \\.

6.7.3.7. Базовая форма регулярных выражений

Форма BRE отличается от формы ERE следующим. |, +, ? являются обыкновенными символами и не имеют никаких иных функциональных замен. Разделителями пределов являются \{ и \}, а { и } считаются обыкновенными символами. Группировка вложенных подвыражений осуществляется с помощью \(и \), а (и) считаются обыкновенными символами. Маркер позиционирования ^ считается простым символом в любом месте кроме начала всего выражения или вложенного подвыражения. Аналогично маркер позиционирования $ считается простым символом в любом месте кроме конца всего выражения или вложенного подвыражения. А символ * считается обычным в начале выражения или вложенного подвыражения (после возможного ^). Так же возможны обратные ссылки с однозначным номером, а \< и >\ являются синонимами для [[:<:]] и [[:>:]] соответственно. Других управляющих последовательностей не предусмотрено.

6.8. Функции форматирования

Функции форматирования PostgreSQL предоставляют богатые возможности для представления различных типов данных (даты и времени, целых чисел, чисел с плавающей точкой и прочие) в нужном виде и для преобразования форматированных строк в соответствующие типы данных (смотри таблицу 48). Все эти функции следуют общему правилу задания аргументов: первым аргументом является значение, которое необходимо отформатировать, а вторым - шаблон, согласно которому будет выполняться вывод или ввод.

Функция to_timestamp также может принимать один аргумент типа double precision для преобразования из типа Unix epoch в тип временной метки с часовым поясом timestamp with time zone. Целочисленное представление Unix epoch неявно приводится к типу double precision.

Таблица 48 – Функции форматирования

Функция	Тип результата	Описание	Пример
to_char(timestamp, text)	text	Переводит временную метку timestamp в строку	to_char(current_timestamp, 'HH12:MI:SS')
to_char(interval, text)	text	Переводит временной интервал в строку	to_char(interval '15h 2m 12s', 'HH24:MI:SS')
to_char(int, text)	text	Переводит целое число в строку	to_char(125, '999')
to_char(double precision, text)	text	Переводит число типа double precision или real в строку	to_char(125.8::real, '999D9')
to_char(numeric, text)	text	Переводит число типа numeric в строку	to_char(-125.8, '999D99S')
to_date(text, text)	date	Переводит календарную дату в строку	to_date('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	Переводит строку в число numeric	to_number('12,454.8-', '99G999D9S')
to_timestamp(text, text)	timestamp with time zone	Переводит строку во временную метку timestamp	to_timestamp('05 Dec 2000', 'DD Mon YYYY')
to_timestamp(double precision)	timestamp with time zone	Переводит переменную типа UNIX epoch в значение временной метки timestamp	to_timestamp(200120400)

В строке, определяющей формат выходной строки, можно задать ряд специальных шаблонов, которые при выводе будут замещены соответствующим образом отформатированными данными из заданного значения. Любой другой текст строки формата будет скопирован в исходном виде. Аналогично в строке формата для ввода данных эти шаблоны определяют местонахождение отдельных частей входных данных.

В таблице 49 показаны шаблоны, доступные для форматирования типов календарных дат и времени.

Таблица 49 – Шаблоны для форматирования дат и времени

Шаблон	Описание
HH	Час (01 – 12)
HH12	Час (01 – 12)
HH24	Час (00 – 23)
MI	Минуты (00 – 59)
SS	Секунды (00 – 59)
MS	Миллисекунды (000 – 999)
US	Микросекунды (000000 – 999999)
SSSS	Секунды после полуночи (0 – 86399)

Продолжение таблицы 49

Шаблон	Описание
AM, am, PM или pm	Индикатор полудня (без дробной части)
A.M., a.m., P.M. или p.m.	Индикатор полудня (с дробной частью)
Y,YYY	Год (четыре и более цифр) с запятой
YYYY	Год (четыре и более цифр)
YYY	Последние три цифры года
YY	Последние две цифры года
Y	Последняя цифра года
IYYY	Год (четыре и более цифр) ISO
IYY	Последние три цифры года ISO
IY	Последние две цифры года ISO
I	Последняя цифра года ISO
BC, bc, AD или ad	Индикатор эры (без дробной части)
B.C., b.c., A.D., или a.d.	Индикатор эры (с дробной частью)
MONTH	Полное имя месяца в верхнем регистре (дополненное пробелами до 9 символов)
Month	Полное имя месяца в смешанном регистре (дополненное пробелами до 9 символов)
month	Полное имя месяца в нижнем регистре (дополненное пробелами до 9 символов)
MON	Трехбуквенная аббревиатура месяца в верхнем регистре
Mon	Трехбуквенная аббревиатура месяца в смешанном регистре
mon	Трехбуквенная аббревиатура месяца в нижнем регистре
MM	Номер месяца (01 – 12)
DAY	Полное имя дня в верхнем регистре (дополненное пробелами до 9 символов)
Day	Полное имя дня в смешанном регистре (дополненное пробелами до 9 символов)
day	Полное имя дня в нижнем регистре (дополненное пробелами до 9 символов)
DY	Трехбуквенная аббревиатура имени дня в верхнем регистре
Dy	Трехбуквенная аббревиатура имени дня в смешанном регистре
dy	Трехбуквенная аббревиатура имени дня в нижнем регистре
DDD	День в году (001 – 366)
IDDD	ISO день в году (001 – 371; 1 день года — понедельник первой ISO недели)
DD	День в месяце (01 – 31)
D	День недели (1 – 7; воскресенье = 1)
ID	ISO день недели (1 – 7; понедельник = 1)
W	Неделя месяца (1 – 5), где первая неделя начинается с первого дня месяца
WW	Неделя года (1 – 53), где первая неделя начинается с первого дня года
IW	ISO номер недели года (первый четверг нового года принадлежит первой неделе)
CC	Столетие (2 цифры)

Окончание таблицы 49

Шаблон	Описание
J	Юлианский день (дней с 1-го января 4712 года до н.э.)
Q	Квартал (игнорируется функциями to_date и to_timestamp)
RM	Месяц римскими цифрами (I – XII; I = январь) в верхнем регистре
rm	Месяц римскими цифрами (i – xii; i = январь) в нижнем регистре
TZ	Имя временной зоны в верхнем регистре
tz	Имя временной зоны в нижнем регистре

К шаблону могут применяться уточняющие модификаторы. Например, FMMonth - это шаблон Month с модификатором FM. В таблице 50 показаны модификаторы для форматирования дат и времени.

Таблица 50 – Модификаторы шаблонов дат и времени

Модификатор	Описание	Пример
Префикс FM	Режим заполнения (подавляет вывод незначащих пробелов или нулей)	FMMonth
Суффикс TH	Добавление порядкового числового суффикса в верхнем регистре	DDTH
Суффикс th	Добавление порядкового числового суффикса в нижнем регистре	DDth
Префикс FX	Указатель фиксированного формата	FX Month DD Day
Префикс TM	Режим трансляции в имена (выводятся локализованные имена дней и месяцев в соответствии с lc_time)	TMMonth
Суффикс SP	spell mode (еще не реализовано)	DDSP

При форматировании дат и времени необходимо учитывать следующее:

1) Префикс FM подавляет вывод начальных нулей и хвостовых пробелов, которые иначе будут добавлены для выравнивания вывода до фиксированной длины. В PostgreSQL модификатор FM действует только на следующую спецификацию, тогда как в Oracle — на все последующие, а повторяющиеся указания FM переключают это поведение.

2) Префикс TM не включает замыкающих пробелов.

3) Функции to_timestamp и to_date при разборе входной строки пропускают пробелы, если не задан модификатор FX. При этом FX должен быть указан в начале шаблона. Например, вызов to_timestamp('2000 JUN','YYYY MON') будет выполнен, а to_timestamp('2000 JUN','FXYYYY MON') будет прерван с ошибкой из-за лишних пробелов во входной строке.

4) Обычный текст может использоваться в форматных строках to_char, для оформления вывода. Можно поместить подстроку в двойные кавычки для интерпрета-

ции в качестве обычного текста даже шаблонов. Например, в форматной строке '"Hello Year" YYYY' подстрока YYYY будет интерпретироваться как шаблон и заменяться четырьмя цифрами года, а буква Y в подстроке Year нет. В функциях to_date, to_number, и to_timestamp заключенная в кавычки строка пропускает соответствующее количество символов, например "XX" пропустит два входных символа.

5) Если необходимо вывести в форматной строке двойную кавычку, ее необходимо предварить обратной косой чертой. Например: '\"YYYY Month\"'.

6) Преобразование с помощью шаблона YYYY из строки в timestamp или date имеет ограничения при использовании года с более чем четырьмя цифрами. Необходимо использовать какой-либо разделитель после года во входной строке, иначе год всегда будет интерпретироваться как имеющий четыре цифры. Например, 20000 год в вызове: to_date('200001131', 'YYYYMMDD') будет интерпретирован неправильно (как 2000-01-13), а в вызовах to_date('20000-1131', 'YYYY-MMDD') и to_date('20000Nov31', 'YYYYMonDD') — правильно (как 20000-11-31).

7) При преобразовании из строки в timestamp или date, поле CC игнорируется в присутствии полей YYY, YYYY или Y, YYY. Если CC используется совместно с YY и Y, год вычисляется по формуле (CC-1)*100+YY. Если задан только век, за год принимается первый год указанного века.

8) День недели в формате ISO (отличающийся от Григорианского) может быть задан в функциях to_timesatmp и to_date двумя способами:

   а) Год, неделя и день недели, например to_date('2006-42-4','IYYY-IW-ID') возвратит дату 2006-10-19. При отсутствии указания дня недели по умолчанию принимается 1 (Понедельник).

   б) Год и день в году, например to_date('2006-291', 'IYYY-IDDD') так же возвращает 2006-10-19.

Попытка создать дату используя смешанное представления из полей дат ISO и Григорианских является бессмысленной и может привести к непредсказуемому результату. В контексте ISO года, понятия месяца или дня месяца не существуют. В контексте же Григорианского года не существует понятия ISO недели. Необходимо использовать тот или иной подход раздельно.

9) Миллисекунды (MS) и микросекунды (US) при преобразовании из строк во временные отметки (timestamp) используются как доля секунды после десятичной точки. Например, в вызове to_timestamp('12:3', 'SS:MS') миллисекунды будут интерпретированы не как 3, а как 300, поскольку оно рассматривается как 12 + 0.3, а не как 12 + 3/1000. Таким образом при использова-

нии формата SS:MS, значения 12:3, 12:30 и 12:300 задают одно и тоже количество миллисекунд. Для задания трех миллисекунд необходимо использовать 12:003, которое рассчитается как 12+0.003 = 12.003 секунды. Вот более сложный пример: to_timestamp('15:12:02.020.001230', 'HH:MI:SS.MS.US') интерпретируется как 15 часов 12 минут и 2 секунды + 20 миллисекунд + 1230 микросекунд, что равно 2.021230 секунды.

10) Полученный с помощью to_char(..., 'ID') день недели соответствует полученному с помощью функции extract(isodow from ...), но полученный с помощью to_char(..., 'D') номер дня не будет соответствовать extract(dow from ...).

11) Вызов to_char(interval) применяет HH и HH12 как часы в пределах одного дня, тогда как HH24 может привести к выходу за границу дня, установке значения более 24.

В таблице 51 показаны шаблоны, используемые для форматирования чисел.

Таблица 51 – Шаблоны для форматирования чисел

Шаблон	Описание
9	Значение с заданным числом цифр
0	Значение с начальными нулями
. (точка)	Десятичная точка
, (запятая)	Разделитель тысяч
PR	Отрицательное значение в угловых скобках
S	Отрицательное значение со знаком минус (с локализацией)
L	Символ валюты (с локализацией)
D	Десятичная точка (с локализацией)
G	Разделитель тысяч (с локализацией)
MI	Знак минуса в заданной позиции (если число меньше нуля)
PL	Знак плюса в заданной позиции (если число больше нуля)
SG	Знак плюса или минуса в заданной позиции
RN	Римское число (в диапазоне 1 - 3999)
TH или th	Преобразование в порядковый номер
V	Смещение на n цифр
EEEE	Научная нотация

При форматировании чисел необходимо учитывать:

- При форматировании чисел со знаком с использованием SG, PL или MI знак необязательно выводится рядом с числом. Например, to_char(-12, 'S9999) выведет ' -12', а to_char(-12, 'MI9999') выведет '- 12'.

- 9 задает значение с количеством цифр, равным количеству девяток. Если цифр в числе меньше, чем количество девяток, то на этих местах будут выведены пробелы.
- TH не работает с числами меньше ноля или дробными.
- PL, SG и TH являются расширениями PostgreSQL.
- V по сути является умножением исходного значения на 10^n, где n число цифр, указанных за V. Функция to_char не поддерживает использование V в сочетании с десятичной точкой. Таким образом, 99.9V99 не обрабатывается.
- Научная нотация EEEE не может быть использована в комбинации с любыми шаблонами форматирования, отличными от цифровых шаблонов и шаблонов положения десятичной точки, и должна быть указана в конце строки форматирования (например 9.99EEEE.

К шаблону могут применяться уточняющие модификаторы. Например, FM9999 является шаблоном 9999 с модификатором FM. В таблице 52 показаны модификаторы для форматирования чисел.

Таблица 52 – Модификаторы шаблонов чисел

Модификатор	Описание	Пример
Префикс FM	Режим заполнения (подавляет вывод незначащих пробелов или нулей)	FM9999
Суффикс TH	Добавление порядкового числового суффикса в верхнем регистре	999TH
Суффикс th	Добавление порядкового числового суффикса в нижнем регистре	999th

В таблице 53 показаны примеры использования функции to_char.

Таблица 53 – Примеры использования функции 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, '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'

Окончание таблицы 53

Выражение	Результат
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'
26 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'

6.9. Функции и операторы типов дат и времени

В таблице 55 перечислены доступные функции для обработки дат и времени. Таблица 54 показывает смысл базовых арифметических операторов (+, – и прочих). Функции форматирования описаны в 6.8.

Типы данных календарных дат и времени описаны в 5.5.

Все функции и операторы, описанные ниже, которые работают с типами `time` или `timestamp` имеют два варианта: для работы с этими типами с указанием временных зон и без их указания. Для краткости эти две формы не разделяются. Операторы + и * обладают свойством коммутативности (например, `date+integer` и `integer+date`). Далее

приводится только одна из подобных пар.

Таблица 54 – Операторы типов дат и времени

Оператор	Пример	Результат
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00:00'
+	time '01:00' + interval '3 hours'	time '04:00:00'
-	- interval '23 hours'	interval '-23:00:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3'
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00:00'
-	time '05:00' - time '03:00'	interval '02:00:00'
-	time '05:00' - interval '2 hours'	time '03:00:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00:00'
-	interval '1 day' - interval '1 hour'	interval '1 day -01:00:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00:00'
*	900 * interval '1 second'	interval '00:15:00'
*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

Таблица 55 – Функции для типов дат и времени

Функция	Описание	Тип результата, Пример, Результат	
age(timestamp, timestamp)	Интервал между двумя временными отметками	Тип	interval
		Пример	age(timestamp '2001-04-10', timestamp '1957-06-13')

Продолжение таблицы 55

Функция	Описание	Тип результата, Пример, Результат	
		Результат	`43 years 9 mons 27 days`
`age(timestamp)`	Интервал между текущим днем и временной отметкой	Тип	`interval`
		Пример	`age(timestamp '1957-06-13')`
		Результат	`43 years 8 mons 3 days`
`clock_timestamp()`	Текущая дата и время (изменяется в процессе выполнения); см. 6.9.4	Тип	`timestamp with time zone`
		Пример	
		Результат	
`current_date`	Текущая дата; см. 6.9.4	Тип	`date`
		Пример	
		Результат	
`current_time`	Текущее время; см. 6.9.4	Тип	`time with time zone`
		Пример	
		Результат	
`current_timestamp`	Текущая дата и время (начала транзакции); см. 6.9.4	Тип	`time with time zone`
		Пример	
		Результат	
`date_part(text, timestamp)`	Заданная часть временной отметки (эквивалентно `extract`); см. 6.9.1	Тип	`double precision`
		Пример	`date_part('hour', timestamp '2001-02-16 20:38:40')`
		Результат	`20`

Продолжение таблицы 55

Функция	Описание	Тип результата, Пример, Результат	
`date_part(text, interval)`	Заданная часть временного интервала (эквивалентно `extract`); см. 6.9.1	Тип	`double precision`
		Пример	`date_part('month', interval '2 years 3 months')`
		Результат	`3`
`date_trunc(text, timestamp)`	Усечение временной отметки до указанной точности; см. 6.9.2	Тип	`timestamp`
		Пример	`date_trunc('hour', timestamp '2001-02-16 20:38:40')`
		Результат	`2001-02-16 20:00:00`
`extract(field from timestamp)`	Заданная часть временной отметки; см. 6.9.1	Тип	`double precision`
		Пример	`extract(hour from timestamp '2001-02-16 20:38:40')`
		Результат	`20`
`extract(field from interval)`	Заданная часть временного интервала; см. 6.9.1	Тип	`double precision`
		Пример	`extract(month from interval '2 years 3 months')`
		Результат	`3`
`isfinite(date)`	Проверка на «конечность» даты	Тип	`boolean`
		Пример	`isfinite(date '2001-02-16')`
		Результат	`true`
`isfinite(timestamp)`	Проверка на «конечность» временной отметки	Тип	`boolean`
		Пример	`isfinite(timestamp '2001-02-16 21:28:30')`
		Результат	`true`
`isfinite(interval)`	Проверка на «конечность» временного интервала	Тип	`boolean`
		Пример	`isfinite(interval '4 hours')`
		Результат	`true`
`justify_days( interval)`	Применить интервал таким образом, чтобы тридцатидневные периоды представлялись как месяцы	Тип	`interval`
		Пример	`justify_days(interval '30 days')`
		Результат	`1 month`
`justify_hours( interval)`	Применить интервал таким образом, чтобы двадцатичетырехчасовые периоды представлялись как дни	Тип	`interval`
		Пример	`justify_hours(interval '24 hours')`
		Результат	`1 day`
`justify_interval( interval)`	Применить интервал, используя `justify_days` и `justify_hours` с дополнительными преобразованиями	Тип	`interval`
		Пример	`justify_interval(interval '1 mon -1 hour')`
		Результат	`29 days 23:00:00`

Окончание таблицы 55

Функция	Описание	Тип результата, Пример, Результат	
localtime	Текущее время дня; см. 6.9.4	Тип	time
		Пример	
		Результат	
localtimestamp	Текущая дата и время (начала транзакции); см. 6.9.4	Тип	timestamp
		Пример	
		Результат	
now()	Текущая дата и время (начала транзакции); см. 6.9.4	Тип	timestamp with time zone
		Пример	
		Результат	
statement_timestamp()	Текущая дата и время (начала транзакции); см. 6.9.4	Тип	timestamp with time zone
		Пример	
		Результат	
timeofday()	Текущая дата и время (аналогично clock_timestamp, но в текстовом виде); смотри 6.9.4	Тип	text
		Пример	
		Результат	
transaction_timestamp()	Текущая дата и время (начала транзакции); см. 6.9.4	Тип	timestamp with time zone
		Пример	
		Результат	

В дополнение к перечисленным функциям поддерживается SQL оператор OVERLAPS.

Пример

```
(start1, end1) OVERLAPS (start2, end2)
(start1, length1) OVERLAPS (start2, length2)
```

Выражение возвращает истину в случае, когда перекрываются два временных периода, заданных своими концевыми точками, и ложь в противном случае. Концевыми точками могут выступать пары дат, времени или временных отметок, или даты, времени, временной отметки с последующим временным интервалом.

Пример

```
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 (или вычитании) значения интервала к временной отметке с часовым поясом, компонент дней увеличивает (или уменьшает) дату временной отметки на указанное количество дней. Изменения пересекающие границу перехода на летнее время (при установленном часовом поясе сессии в соответствии с DST) `interval '1 день'` не всегда соответствует `interval '24 часа'`. Например, если часовой пояс сессии установлен в `CST7CDT`, то временная отметка с учетом часового пояса `'2005-04-02 12:00-07' + interval '1 day'` даст в результате временную отметку с учетом часового пояса `'2005-04-03 12:00-06'`, тогда как добавление `interval '24 hours'` к той же временной отметке даст `'2005-04-03 13:00-06'`, так как переход осуществляется в момент времени `2005-04-03 02:00` для часового пояса `CST7CDT`.

Необходимо отметить, что возможна неоднозначность при получении количества месяцев при вызове `age`, поскольку разные месяцы содержат разное количество дней. Подход используемый в PostgreSQL при вычислении частей месяцев рассматривает месяцы, начиная с раннего из двух дат. Например, `age('2004-06-01', '2004-04-30')` использует `Апрель` что бы вернуть `1 месяц 1 день`, тогда как использование `Мая` возвратило бы `1 месяц 2 дня` поскольку в мае 31 день, а в апреле только 30.

6.9.1. EXTRACT, date_part

```
EXTRACT(<поле> from <источник>)
```

Функция `extract` выделяет указанную часть временного значения, такую как год или час. `<источник>` — выражение, которое возвращает типы `timestamp`, `time` или `interval`. Выражения, возвращающее тип `date`, преобразуются к типу `timestamp` перед вызовом функции. `<поле>` — идентификатор или строка, которая показывает какую именно часть надо извлечь. Функция `extract` возвращает значение типа `double precision`. Допустимыми значения `<поля>` рассмотрены далее.

1) `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 начинается 0001-01-01 00:00:00 AD. Это определение применяется во всех странах, использующих Григорианский календарь. Не существует века с номером 0, после -1 идет сразу 1.

PostgreSQL версий ранее 8.0 не следовал этому соглашению и просто возвращал год деленный на 100.

2) day — для значений timestamp возвращается день месяца (от 1 до 31); для interval возвращается количество 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
```

3) decade — значение поля год, разделенное на 10:

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Результат: 200
```

4) dow — день недели, значение поля изменяется от 0 до 6, считая с воскресенья:

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Результат: 5
```

Следует отметить, что полученный таким образом номер дня недели отличается от полученного вызовом функции to_char(..., 'D').

5) doy — день года, значение поля изменяется от 0 до 365/366:

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Результат: 47
```

6) epoch — для значений типа timestamp with time zone является числом секунд, прошедших с 1970-01-01 00:00:00-00 UTC (может быть отрицательным); для значений типов date и timestamp является числом секунд, прошедших с 1970-01-01 00:00:00-00 локального времени; для значений типа interval является числом секунд в этом интервале:

```
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 значение типа epoch обратно в timestamp возможно следующим образом:

```
SELECT TIMESTAMP WITH TIME ZONE 'epoch'
        + 982384720.12 * INTERVAL '1 second';
```

7) hour — час, значение поля изменяется от 0 до 23:

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
```

```
Результат: 20
```

8) isodow — день недели, значение поля изменяется от 1 до 7, считая с понедельника:

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
Результат: 7
```

Это идентично dow за исключением воскресенья. Подобная нумерация соответствует стандарту ISO 8601.

9) isoyear — год в соответствии с ISO 8601 (неприменимо к интервалам).

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
Результат: 2005
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
Результат: 2006
```

Каждый год ISO начинается с понедельника недели, содержащей 4-е января, так что ранние январские или поздние декабрьские даты могут давать различное значение года для ISO и Григорианского календарей.

Поле доступно начиная с версии PostgreSQL 8.3.

10) microseconds — значение поля секунд, включая дробную часть, умноженное на 1 000 000:

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Результат: 28500000
```

11) millennium — тысячелетие:

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Результат: 3
```

1900-е годы относятся ко второму тысячелетию. Третье тысячелетие начинается 1 января 2001 года. PostgreSQL версии ранее 8.0 не следовали этому соглашению, а просто делили год на 1000.

12) milliseconds — значение поля представляет собой секунды, включая дробную часть, умноженные на 1000:

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Результат: 28500
```

13) minute — минуты, значение поля изменяется от 0 до 59:

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Результат: 38
```

14) month — для значений типа timestamp является числом месяцев с начала года от 1 до 12. Для значений типа interval является остатком от деления числа месяцев на 12 и изменяется от 0 до 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
```

15) quarter — квартал года, значение поля изменяется в интервале от 1 до 4, в котором находится заданная временная отметка (только для типа timestamp):

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Результат: 1
```

16) second — значение поля секунды, включая дробную часть в интервале от 0 до 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
```

17) timezone — смещение в секундах временной зоны от UTC. Положительные значения соответствуют временным зонам, расположенных к востоку от UTC, а отрицательные к западу от UTC.

18) timezone_hour — часовая часть значения временной зоны;

19) timezone_minute — минутная часть значения временной зоны;

20) week — количество недель с начала года, в которое входит указанное значение. По определению стандарта ISO 8601 первой неделей года считается неделя, которая включает в себя 4 января этого года (неделя ISO начинается с понедельника). Ранние январские даты могут относится либо к 52-й либо к 53-й неделе предыдущего года. Например, 2005-01-01 попадает в 53-ю неделю 2004 года, 2006-01-01 в 52-ю неделю 2005, а 2012-12-31 часть первой недели 2013. Для получения точного значения рекомендуется совместно с week использовать isoyear.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Результат: 7
```

21) year — значение поля года:

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Результат: 2001
```

Функция extract в основном предназначена для выполнения вычислений. Для форматирования значений дат и времени для отображения смотри 6.8.

Функция date_part эмулирует традиционную для Ingres функцию, эквивалентную стандартной функции extract:

```
date_part('<поле>', <источник>)
```

Следует обратить внимание, что в этом случае <поле> является строковым значением, а не именем. Допустимые значения для <поля> те же самые, что и для функции

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
```

6.9.2. date_trunc

Функция date_trunc по смыслу эквивалентна функции trunc для чисел:

```
date_trunc('<поле>', <источник>)
```

<источник> — выражение типа timestamp или interval (значения типа date и time приводятся к типам timestamp или interval автоматически). <поле> позволяет выбрать точность, до которой будет усечено исходное значение. Возвращается значение типа timestamp или interval, в котором все поля меньшие, чем указанное, сброшены в ноль (или единицу для полей дня и месяца).

Допустимыми значениями <поля> являются:

- microseconds;
- milliseconds;
- seconds;
- minute;
- hour;
- day;
- week;
- month;
- quarter;
- year;
- decade;
- century;
- millennium.

Пример

```
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
```

6.9.3. AT TIME_ZONE

Конструкция AT TIME ZONE позволяет преобразовывать значение временных отметок в различные временные зоны как показано в таблице 56.

Таблица 56 – Варианты использования AT TIME ZONE

Выражение	Тип результата	Описание
timestamp without time zone AT TIME ZONE zone	timestamp with time zone	Принимает метку времени как заданную в указанной временной <зоне>
timestamp with time zone AT TIME ZONE zone	timestamp without time zone	Конвертирует указанное время в локальное время для новой временной <зоны>
time with time zone AT TIME ZONE zone	time with time zone	Конвертирует локальное время из одной временной зоны в другую

В этих выражениях временная <зона> может быть задана как текстовая строка (например: 'PST') или как интервал (например: interval '-08:00'). Временная зона может быть задана любым из описанных в 5.5.3 способом.

Пример (считая локальной зоной PST8PDT):

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'MST';
Результат: 2001-02-16 19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'MST';
Результат: 2001-02-16 18:38:40
```

В первом примере временная отметка без временной зоны интерпретируется как временная отметка зоны MST (UTC-7) , которая затем преобразуется в PST (UTC-8) для отображения. Во втором примере временная отметка зоны EST (UTC-5) преобразуется в локальное время зоны MST (UTC-7). Функция timezone(<зона>, <временная отметка>) эквивалентна стандартной SQL-конструкции AT TIME ZONE <зона>.

6.9.4. Текущие дата и время

PostgreSQL предоставляет ряд функций для получения значений текущих даты и времени. Доступны следующие определенные стандартом SQL функции, использующие 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 значения с временной зоной, LOCALTIME и LOCALTIMESTAMP — без временных зон.

Для CURRENT_TIME, CURRENT_TIMESTAMP, LOCATIME и LOCALTIMESTAMP может быть задана точность, которая задает округление результата до заданного числа дробной части. Без указания точности результат будет выдан с максимально доступной точностью.

Примеры:

```
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
```

Поскольку указанные функции возвращают время начала транзакции, во время ее существования возвращаемые значение не меняются. Можно считать это особенностью PostgreSQL, и сделано для однозначного представления о текущем времени в течении одной транзакции, так что бы все осуществленные в ее пределах модификации имели одинаковую метку времени.

Примечание. Другие системы могут этого не придерживаться и обновлять эти значения чаще.

PostgreSQL также предоставляет набор функций для получения времени начала отдельной команды, а так же действительного времени в момент вызова функций:

- transaction_timestamp();
- statement_timestamp();
- clock_timestamp();
- timeofday();
- now().

Функция transaction_timestamp() эквивалентна CURRENT_TIMESTAMP, но имеет название, лучше отражающее смысл возвращаемого значения. statement_timestamp() возвращает время начала текущей SQL команды (если более

точно, то время получения последней команды от клиента). statement_timestamp() и transaction_timestamp() возвращают одно значение для первой команды в транзакции. clock_timestamp() возвращает действительно текущее время, следовательно, не остается неизменным во время выполнения команды. timeofday() как и clock_timestamp() возвращает текущее время, но в виде текстовой строки. now() является традиционным для PostgreSQL эквивалентом функции CURRENT_TIMESTAMP.

Все типы дат и времени преобразовывают специальную константу now, которая задает текущие дату и время. Таким образом все следующие выражения эквивалентны:

- SELECT CURRENT_TIMESTAMP;
- SELECT now();
- SELECT TIMESTAMP 'now'; – (использование для DEFAULT некорректно).

Примечание. При использовании последней формы для задания значения по умолчанию для столбца таблицы система преобразует эту константу в значение в момент создания таблицы и в дальнейшем всегда будет использовать именно это значение, а не текущее на момент вставки данных. Первые же две формы обеспечат вставку времени, текущего на момент выполнения конкретной операции, поскольку они являются вызовами функций.

6.9.5. Задержка исполнения

Существует функция для задержки исполнения серверного процесса:

```
pg_sleep(seconds)
```

pg_sleep обеспечивает задержку процесса, связанного с текущей сессией, на указанное количество секунд. Задержка указывается значением типа double precision, что позволяет указывать дробные части секунд.

Пример

```
SELECT pg_sleep(1.5);
```

Примечания:

1. Эффективная точность задания интервала зависит от операционной системы, в общем случае это значение составляет 0.01 секунды. Задержка длится как минимум указанное количество времени, но может и превышать это значение, что зависит различных факторов подобных степени загрузки сервера.
2. Необходимо следить за тем, что бы сессия перед вызовом pg_sleep не осуществляла большого количества блокировок, так как в этом случае могут быть задержаны и другие сессии, что может сильно снизить производительность всей системы.

6.10. Функции для работы с перечислениями

Для более удобной и прозрачной работы со значениями типа перечислений, описанных в 5.7, доступен ряд функций, приведенных в таблице 57. В следующем ниже примере показано создание перечисления.

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

Таблица 57 – Функции для работы с перечислениями

Функция	Описание	Пример	Результат
enum_first(anyenum)	Первое значение перечисления	enum_first(null::rainbow)	red
enum_last(anyenum))	Последнее значение перечисления	enum_last(null::rainbow))	purple
enum_range(anyenum)	Возвращает все значения перечисления в виде упорядоченного массива	enum_range(null::rainbow)	red,orange,yellow, green,blue,purple
enum_range(anyenum, anyenum)	Возвращает все значения перечисления из указанного диапазона в виде упорядоченного массива. Аргументы должны быть элементами того же перечисления. При этом указание неопределенного значения для начала или конца диапазона соответствует заданию диапазона начиная с первого или до последнего значения перечисление соответственно	enum_range('orange'::rainbow, 'green'::rainbow)	orange,yellow,green
		enum_range(NULL, 'green'::rainbow)	red,orange, yellow,green
		enum_range('orange'::rainbow, NULL)	orange,yellow,green

Необходимо отметить, что за исключением формы вызова enum_range с двумя аргументами, эти функции безразличны к значению аргумента, важно только соответствие его типа. С тем же результатом в качестве аргумента могут использоваться значения указанного типа и неопределенные значения. Более общим случаем является применение функций к столбцам таблицы или аргументам функции, нежели чем к заданным именам перечислений как предлагалось в примерах.

6.11. Геометрические функции и операторы

Геометрические типы point, box, lseg, line, path, polygon и circle имеют большой набор встроенных функций и операторов, показанных в таблицах 58, 59 и 60.

Примечание. Необходимо отметить, что оператор эквивалентности ~= обладает обычным смыслом для типов point, box, polygon и circle. Некоторые из этих типов так же имеют оператор равенства =, но он применим для сравнения одинаковых площадей. Другие скалярные операторы сравнения <= и подобные для указанных типов так же сравнивают площади.

Таблица 58 – Геометрические операторы

Оператор	Описание	Пример
+	Смещение	box '((0,0),(1,1))' + point '(2.0,0)'
-	Смещение	box '((0,0),(1,1))' - point '(2.0,0)'
*	Масштабирование/вращение	box '((0,0),(1,1))' * point '(2.0,0)'
/	Масштабирование/вращение	box '((0,0),(1,1))' / point '(2.0,0)'
#	Пересечение	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	Число точек в ломаной линии или полигоне	# '((1,0),(0,1),(-1,0))'
@-@	Длина окружности	@-@ path '((0,0),(1,0))'
@@	Геометрический центр	@@ circle '((0,0),10)'
##	Ближайшая точка на отрезке	point '(0,0)' ## lseg '((2,0),(0,2))'
<->	Расстояние между	circle '((0,0),1)' <-> circle '((5,0),1)'
&&	Объекты перекрываются?	box '((0,0),(1,1))' && box '((0,0),(2,2))'
<<	Целиком слева от?	circle '((0,0),1)' << circle '((5,0),1)'
>>	Целиком справа от?	circle '((5,0),1)' >> circle '((0,0),1)'
&<	Не расширяет вправо?	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	Не расширяет влево?	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<<|	Объект целиком ниже?	box '((0,0),(3,3))' <<| box '((3,4),(5,5))'
>>|	Объект целиком выше?	box '((3,4),(5,5))' |>> box '((0,0),(3,3))'
&<|	Не расширяет выше?	box '((0,0),(1,1))' &<| box '((0,0),(2,2))'
|&>	Не расширяет ниже?	box '((0,0),(3,3))' |&> box '((0,0),(2,2))'
<^	Ниже?(возможно касание)	circle '((0,0),1)' <^ circle '((0,5),1)'
>^	Выше?(возможно касание)	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	Объекты пересекаются?	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?-	Горизонтально?	?- lseg '((-1,0),(1,0))'
?-	Выровнены горизонтально?	point '(1,0)' ?- point '(0,0)'
?|	Вертикально?	?| lseg '((-1,0),(1,0))'
?|	Выровнены вертикально?	point '(1,0)' ?| point '(0,0)'
?-|	Перпендикулярно?	lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'
?||	Параллельно?	lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'
@>	Содержит?	circle '((0,0),2)' @> point '(1,1)'
<@	Содержится в?	point '(1,1)' <@ circle '((0,0),2)'
~=	Такой же?	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

Примечание. Ранее версии PostgreSQL 8.2, операторы содержания @> и <@ назывались соответственно ~ и @. Старые имена доступны, но считаются устаревшими.

Таблица 59 – Геометрические функции

Функция	Тип	Описание	Пример
area(object)	double precision	Площадь	area(box '((0,0),(1,1))')

Окончание таблицы 59

Функция	Тип	Описание	Пример
center(object)	point	Центр	center(box '((0,0),(1,2))')
diameter(circle)	double precision	Диаметр окружности	diameter(circle '((0,0),2.0)')
height(box)	double precision	Высота	height(box '((0,0),(1,1))')
isclosed(path)	boolean	Замкнутая ломаная?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	Не замкнутая ломаная?	isopen(path '[(0,0),(1,1),(2,0)]')
length(object)	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 ломанную в замкнутую	pclose(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	Преобразует ломанную в не замкнутую	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	Радиус окружности	radius(circle '((0,0),2.0)')
width(box)	double precision	Ширина	width(box '((0,0),(1,1))')

Таблица 60 – Функции преобразования геометрических типов

Функция	Тип	Описание	Пример
box(circle)	box	Окружность в прямоугольник	box(circle '((0,0),2.0)')
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))')
circle(box)	circle	Прямоугольник в окружность	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	Центр и радиус в окружность	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 к линии	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)	point	Многоугольник в ломанную	path(polygon '((0,0),(1,1),(2,0))')
point(double precision, double precision)	point	Создание точки	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)')

Окончание таблицы 60

Функция	Тип	Описание	Пример
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 4-х угольник	polygon(box '((0,0),(1,1))')
polygon(circle	polygon	Окружность в 12-ти угольник	polygon(circle '((0,0),(2,0))')
polygon(npts, circle)	polygon	Окружность в npts угольник	polygon(12,circle '((0,0),(2,0))')
polygon(path)	polygon	Ломаная в многоугольник	polygon(path '((0,0),(1,1),(2,0))')

К координатам точки можно обращаться так же как к элементам массива с индексами [0, 1]. Например, если t.p является столбцом таблицы t типа point, то SELECT p[0] FROM t выведет X координату, UPDATE t SET p[1] = ... изменит Y координату. Подобным же образом типы box и lseg можно рассматривать как массивы из двух точек.

Функция area применяется для получения площади значений типов box, circle и path. Для ломанных эта функция работает, только когда они не являются пересекающимися. Например для ломанной '((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))'::PATH функция не применима, хотя, визуально похожая ломанная '((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))'::PATH будет обработана корректно.

6.12. Функции и операторы типов сетевых адресов

В таблице 61 показаны операторы, доступные для типов cidr и inet. Операторы <<, <<=, >> и >>= проверяют вхождение подсетей: они учитывают только сетевые части двух сравниваемых адресов, игнорируя часть, определяющую полный адрес.

Таблица 61 – Операторы типов cidr и inet

Оператор	Описание	Пример
<	Меньше чем	inet '192.168.1.5' < inet '192.168.1.6'
<=	Меньше или равно	inet '192.168.1.5' <= inet '192.168.1.5'
=	Равно	inet '192.168.1.5' = inet '192.168.1.5'
>	Больше или равно	inet '192.168.1.5' >= inet '192.168.1.5'
>=	Больше	inet '192.168.1.5' > inet '192.168.1.4'
<>	Не равно	inet '192.168.1.5' <> inet '192.168.1.4'
<<	Содержится в	inet '192.168.1.5' << inet '192.168.1/24'
<<=	Содержится в или равно	inet '192.168.1/24' <<= inet '192.168.1/24'

Окончание таблицы 61

Оператор	Описание	Пример
>>	Содержит	inet ’192.168.1/24’ >> inet ’192.168.1.5’
>>=	Содержит или равно	inet ’192.168.1/24’ >>= inet ’192.168.1/24’
~	Побитовое отрицание NOT	~inet ’192.168.1.6’
&	Побитовое И AND	inet ’192.168.1.6’ & inet ’0.0.0.255’
|	Побитовое ИЛИ OR	inet ’192.168.1.6’ | inet ’0.0.0.255’
+	Сложение	inet ’192.168.1.6’ + 25
-	Вычитание	inet ’192.168.1.43’ - 36
-	Вычитание	inet ’192.168.1.43’ - inet ’192.168.1.19’

В таблице 62 перечислены функции, доступные для использования с типами inet и cidr. Функции host(), text() и abbrev() в основном используются как варианты для отображения значений этих типов.

Таблица 62 – Функции типов cidr и inet

Функция	Тип	Описание	Пример	Результат
abbrev(inet)	text	Сокращенная запись	abbrev(inet ’10.1.0.0/16’)	10.1.0.0/16
abbrev(cidr)	text	Сокращенная запись	abbrev(cidr ’10.1.0.0/16’)	10.1/16
broadcast(inet)	inet	Широковещательный адрес	broadcast(’192.168.1.5/24’)	192.168.1.255/24
family(inet)	int	Выделяет семейство по IP-адресу; 4 для IPv4, 6 для IPv6	family(’::1’)	6
host(inet)	text	IP-адреса как текст	host(’192.168.1.5/24’)	192.168.1.5
hostmask(inet)	inet	Маска узла по IP-адресу	hostmask(’192.168.23.20/30’)	0.0.0.3
masklen(inet)	int	Длина маски сети	masklen(’192.168.1.5/24’)	24
netmask(inet)	inet	Маска сети	netmask(’192.168.1.5/24’)	255.255.255.0
network(inet)	cidr	Сетевая часть IP-адреса	network(’192.168.1.5/24’)	192.168.1.0/24
set_masklen(inet, int)	inet	Задает длину маски сети	set_masklen(’192.168.1.5/24’, 16)	192.168.1.5/16
set_masklen(cidr, int)	cidr	Задает длину маски сети	set_masklen(’192.168.1.0/24’::cidr, 16)	192.168.0.0/16
text(inet)	text	IP-адрес и сетевая маска как текста	text(inet ’192.168.1.5’)	192.168.1.5/32

Любое cidr значение может быть приведено к типу inet явным или неявным образом, что дает возможность использовать для типа cidr все принимающие тип inet функции. (В том случае, если указаны различные функции для этих типов, это подразумевает

их различное поведение). Так же возможно преобразование и значений типа `inet` в тип `cidr`. При этом по умолчанию обновляются разряды правее сетевой маски. Так же возможно преобразовать текстовое значение в типы `inet` или `cidr` использую стандартный синтаксис приведения типов: `inet(expression)` или `colname::cidr`.

В таблице 63 перечислены функции, доступные для использования с типом `macaddr`. Функция `trunc(macaddr)` возвращает MAC-адрес с последними тремя байтами, сброшенными в ноль. Она может быть использована для задания связи префикса MAC-адреса с производителем оборудования.

Таблица 63 – Функции типа macaddr

Функция	Тип	Описание	Пример	Результат
`trunc(macaddr)`	`macaddr`	Устанавливает последние три байта адреса в ноль	`trunc(macaddr '12:34:56:78:90:ab')`	`12:34:56:00:00:00`

Для типа `macaddr` также определены стандартные операторы сравнения (>, < и прочие), предназначенные для лексикографического упорядочивания данных и побитовые арифметические операторы (~, &, |) для NOT, AND и OR.

6.13. Функции и операторы текстового поиска

В таблицах 64, 65 и 66 приведены функции и операторы доступные для текстового поиска. О возможностях текстового поиска в PostgreSQL описано в главе 12.

Таблица 64 – Операторы текстового поиска

Оператор	Описание	Пример	Результат
`@@`	`tsvector соответствует tsquery ?`	`to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat')`	`t`
`@@@`	`Устаревший синоним для @@`	`to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat')`	`t`
`||`	`Объединение tsvector`	`'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector`	`'a':1 'b':2,5 'c':3 'd':4`
`&&`	`Логическое умножение tsquery`	`'fat | rat'::tsquery && 'cat'::tsquery)`	`( 'fat' | 'rat' ) & 'cat`
`||`	`Логическое сложение tsquery`	`'fat | rat'::tsquery || 'cat'::tsquery`	`( 'fat' | 'rat' ) | 'cat'`
`!!`	`Логическое отрицание tsquery`	`!! 'cat'::tsquery`	`!'cat'`
`@>`	`Содержит ли один tsquery другой?`	`'cat'::tsquery @> 'cat & rat'::tsquery`	`f`
`<@`	`Содержится ли один tsquery в другом ?`	`'cat'::tsquery <@ 'cat & rat'::tsquery`	`t`

Примечание. Операторы для определения вложенности запросов рассматривают только лексемы, представленные в обоих запросах, игнорируя операторы комбинирования.

В дополнение к рассмотренным операторам для типов `tsvector` и `tsquery` существуют и обыкновенные `B-tree` операторы сравнения, что не очень применимо для текстового поиска, но позволяет строить индексы по столбцам этих типов.

Таблица 65 – Функции текстового поиска

Функция	Тип, описание, пример, результат	
`get_current_ts_config()`	Тип	`regconfig`
	Описание	Получение конфигурации поиска по умолчанию
	Пример	`get_current_ts_config()`
	Результат	`english`
`length(tsvector)`	Тип	`integer`
	Описание	Количество лексем в `tsvector`
	Пример	`length('fat:2,4 cat:3 rat:5A'::tsvector)`
	Результат	`3`
`numnode(tsquery)`	Тип	`integer`
	Описание	Количество лексем и операторов в `tsquery`
	Пример	`numnode('(fat & rat) | cat'::tsquery)`
	Результат	`5`
`plainto_tsquery([ config regconfig , ] query text)`	Тип	`tsquery`
	Описание	Преобразование в `tsquery` с игнорированием пунктуации
	Пример	`plainto_tsquery('english', 'The Fat Rats')`
	Результат	`'fat' & 'rat'`
`querytree(query tsquery)`	Тип	`text`
	Описание	Получение индексной части `tsquery`
	Пример	`querytree('foo & ! bar'::tsquery)`
	Результат	`'foo'`
`setweight(tsvector, "char")`	Тип	`tsvector`
	Описание	Задание весов элементам `tsvector`
	Пример	`setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A')`
	Результат	`'cat':3A 'fat':2A,4A 'rat':5A`
`strip(tsvector)`	Тип	`tsvector`
	Описание	Удаление позиций и весов из `tsvector`
	Пример	`strip('fat:2,4 cat:3 rat:5A'::tsvector)`
	Результат	`'cat' 'fat' 'rat'`
`to_tsquery([ config regconfig , ] query text)`	Тип	`tsquery`
	Описание	Нормализация слов и преобразование в `tsquery`
	Пример	`host(to_tsquery('english', 'The & Fat & Rats'))`
	Результат	`'fat' & 'rat'`

Окончание таблицы 65

Функция	Тип, описание, пример, результат	
to_tsvector([config regconfig ,] document text)	Тип	tsvector
	Описание	Свертка документа в tsvector
	Пример	to_tsvector('english', 'The Fat Rats')
	Результат	'fat':2 'rat':3
ts_headline([config regconfig,] document text, query tsquery [, options text])	Тип	text
	Описание	Отображение совпадения с запросом
	Пример	ts_headline('x y z', 'z'::tsquery)
	Результат	x y z
ts_rank([weights float4[],] vector tsvector, query tsquery [, normalization integer])	Тип	float4
	Описание	Степень соответствия документа запросу
	Пример	ts_rank(textsearch, query)
	Результат	0.818
ts_rank_cd([weights float4[],] vector tsvector, query tsquery [, normalization integer])	Тип	float4
	Описание	Степень соответствия документа запросу по плотности распределения
	Пример	ts_rank_cd('0.1, 0.2, 0.4, 1.0', textsearch, query)
	Результат	2.01317
ts_rewrite(query tsquery, target tsquery, substitute tsquery)	Тип	tsquery
	Описание	Замена элемента запроса
	Пример	ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery)
	Результат	'b' & ('foo' | 'bar')
ts_rewrite(query tsquery, select text)	Тип	tsquery
	Описание	Замена элемента запроса результатом команды SELECT
	Пример	SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases')
	Результат	'b' & ('foo' | 'bar')
tsvector_update_trigger()	Тип	trigger
	Описание	Генерация триггера для автоматического обновления столбца типа tsvector
	Пример	CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swedish', title, body)
	Результат	
tsvector_update_trigger_column()	Тип	trigger
	Описание	Генерация триггера для автоматического обновления столбца типа tsvector
	Пример	CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, configcol, title, body)
	Результат	

Примечание. Все функции, принимающие необязательный параметр конфигура-

ции regconfig используют значение default_text_search_config, если он не указан.

Функции, представленные в таблице 66, приведены отдельно, поскольку используются в основном не для ежедневной работы с полнотестовым поиском, а для разработки отладки новых конфигурация поиска.

Таблица 66 – Функции отладки текстового поиска

Функция	Тип, описание, пример, результат	
ts_debug([config regconfig,] document text, OUT alias text, OUT description text, OUT token text, OUT dictionaries regdictionary[], OUT dictionary regdictionary, OUT lexemes text[])	Тип	setof record
	Описание	Тестирование конфигурации
	Пример	ts_debug('english', 'The Brightest supernovaes')
	Результат	(asciiword,"Word, all ASCII¨,The, {english_stem},english_stem,{}) ...
ts_lexize(dict regdictionary, token text)	Тип	text[]
	Описание	Тестирование словаря
	Пример	ts_lexize('english_stem', 'stars')
	Результат	{star}
ts_parse(parser_name text, document text, OUT tokid integer, OUT token text)	Тип	setof record
	Описание	Тестирование анализатора
	Пример	ts_parse('default', 'foo - bar')
	Результат	(1,foo) ...
ts_parse(parser_oid oid, document text, OUT tokid integer, OUT token text)	Тип	setof record
	Описание	Тестирование анализатора
	Пример	ts_parse(3722, 'foo - bar')
	Результат	(1,foo) ...
ts_token_type(parser_name text, OUT tokid integer, OUT alias text, OUT description text)	Тип	setof record
	Описание	Получение типов лексем, определенных анализатором
	Пример	ts_token_type('default')
	Результат	(1,asciiword,"Word, all ASCII") ...
ts_token_type(parser_oid oid, OUT tokid integer, OUT alias text, OUT description text)	Тип	setof record
	Описание	Получение типов лексем, определенных анализатором
	Пример	ts_token_type(3722)
	Результат	(1,asciiword,"Word, all ASCII") ...
ts_stat(sqlquery text, [weights text,] OUT word text, OUT ndoc integer, OUT nentry integer)	Тип	setof record
	Описание	Получение статистики столбца типа tsvector
	Пример	ts_stat('SELECT vector from apod')
	Результат	(foo,10,15) ...

6.14. Функции для работы с XML

В даном пункте описываются функции и выражения для работы со значениями типа xml. Информация о типе xml приведена в 5.13. Выражения xmlparse и xmlserialize для преобразования типа xml повторно не рассматриваются.

6.14.1. Создание XML-документов

Для создания XML-документов из данных SQL доступен ряд функций и выражений, особенно удобных для форматирования результатов запросов в виде XML-документов для их дальнейшей обработки клиентским приложением.

6.14.1.1. xmlcomment

`xmlcomment(text)`

Функция `xmlcomment` создает XML-значение типа комментария, содержащее в качестве комментария указанный текст. Для создания корректного XML-комментария передаваемый текст не должен содержать `"--"` или завершаться символом `-`. Результатом вызова функции для неопределенного значения является неопределенное значение.

Пример

```
SELECT xmlcomment('hello');

  xmlcomment
--------------
 <!--hello-->
```

6.14.1.2. xmlconcat

`xmlconcat(xml[, ...])`

Функция `xmlconcat` объединяет список отдельно заданных XML-значений в один фрагмент XML, при этом опускаются неопределенные значения опускаются. Неопределенное значение результата получается только если все аргументы являются неопределенными значениями.

Пример

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

      xmlconcat
----------------------
 <abc/><bar>foo</bar>
```

Аналогичным образом объединяются и XML-определения. При этом, если все определения имеют одинаково заданную версию XML, эта версия указывается и в результате, иначе версия в результате не указывается. Так же если у всех аргументов указано значение `'yes'` XML-определения `standalone`, или у большинства указано, а у хотя бы одного указано `'no'`, в результате указывается это определение. В противном случает XML-определение `standalone` в результате не указывается. Если результат требует указания XML-определения `standalone`, но при этом не задана версия XML, в результат принуди-

тельно вставляется XML версия 1.0, так как XML требует задания версии для определения. Определения кодировки XML игнорируется и не входит в результат в любом случае.

Пример

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1"
standalone="no"?><bar/>');

             xmlconcat
-----------------------------------
 <?xml version="1.1"?><foo/><bar/>
```

6.14.1.3. xmlelement

```
xmlelement(name name [, xmlattributes(value [AS attname]
 [, ... ])] [, content, ...])
```

Выражение `xmlelement` создает XML элемент с указанным именем, атрибутами и содержанием.

Пример

```
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>
```

В имени элемента или атрибута, не являющимся допустимым для XML, недопустимые символы заменяются последовательностью символов `_xHHHH_`, где `HHHH` является их шестнадцатеричным представлением в Unicode кодировке.

Пример

```
SELECT xmlelement(name "foo\$bar", xmlattributes('xyz' as "a&b"));

            xmlelement
--------------------------------
 <foo_x0024_bar a_x0026_b="xyz"/>
```

Явное задание имени атрибута необязательно в случае указания в качестве значения атрибута ссылки на столбец, в этом случае по умолчанию в качестве имени атрибута используется имя столбца. Во всех остальных случаях обязательно явное задание имени атрибута. Следующий пример является допустимым.

Пример

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

А этот нет:

Пример

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

Указанное содержимое элемента форматируется в соответствии с его типом. В том случае, когда содержимое само по себе переставляет XML, возможно создание сложного XML документа.

Пример

```
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 форматируется в виде допустимых символьных данных XML. Это означает замену символов <, >, и & в соответствующие общепринятые аббревиатуры. Двоичные данные (тип данных bytea) представляются либо в шестнадцатеричном, либо в base64 виде, что зависит от значения конфигурационного параметра xmlbinary. Ожидается, что индивидуальное поведение конкретных типов данных изменится, чтобы обеспечить соответствие типов данных SQL и PostgreSQL спецификации XML Schema,

после чего появится более точное описание.

6.14.1.4. xmlforest

xmlforest(content [AS name] [, ...])

Выражение xmlforest создает XML-последовательность элементов с указанными именами и содержанием.

Пример

```
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 видно во втором примере, имя элемента может быть опущено, если значение задается ссылкой на столбец, в этом случае по умолчанию в качестве имени элемента используется имя столбца. В других случаях имя должно быть явно указано.

Недопустимые для XML имена элементов обрабатываются как было описано выше при вызове xmlelement. Так же и содержимое приводится в соответствие с требованиями XML, кроме случая, когда содержимое уже является типом xml.

Необходимо отметить, что полученный таким образом XML не является XML документом, так как содержит более одного элемента. Так что возможно более правильным является окружение выражения xmlforest вызовом xmlelement.

6.14.1.5. xmlpi

xmlpi(name target [, content])

Выражение xmlpi создает XML processing instruction. При этом при указании инструкции не должна содержать ?>.

Пример

```
SELECT xmlpi(name php, 'echo "hello world";');
```

```
            xmlpi
-----------------------------
 <?php echo "hello world";?>
```

6.14.1.6. xmlroot

`xmlroot(xml, version text | no value [, standalone yes|no|no value])`

Выражение `xmlroot` позволяет изменять свойства корневого узла XML-значения. Если указана версия или определение `standalone`, новые значения заменяют существующие.

Пример

```
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>
```

6.14.1.7. xmlagg

`xmlagg(xml)`

Функция `xmlagg` является, в отличие от описанных ранее, агрегирующей. Она объединяет указанные значения так же как и `xmlconcat`. Дополнительная информация по агрегирующим функциям приведена в 6.20.

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/>
```

Для определения порядка объединяемых значений возможно использование следующего подхода:

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

6.14.2. XML-утверждения

Описанные в этом разделе Выражения проверяют свойства XML значений.

6.14.2.1. IS DOCUMENT

```
xml IS DOCUMENT
```

Выражение `IS DOCUMENT` возвращает `TRUE`, если XML-аргумент является корректным XML-документом, `FALSE` если таковым не является (является фрагментом или т. п.), и неопределенное значение, если аргумент является неопределенным значением. Разница между XML-документом и XML-фрагментом рассматривается в разделе 5.13.

6.14.2.2. XMLEXISTS

```
XMLEXISTS(text PASSING [BY REF] xml [BY REF])
```

Функция `xmlexists` возвращает `TRUE`, если XPath выражение, заданное первым аргументом, возвращает XML-узлы, иначе возвращает `FALSE`. (Если какой-либо из аргументов не определен, возвращается неопределенное значение.)

Пример

```
SELECT xmlexists('//town[text() = "Toronto"]' PASSING BY REF
       '<towns><town>Toronto</town><town>Ottawa</town></towns>');

xmlexists
------------
t
(1 row)
```

Выражение `BY REF` в PostgreSQL не имеет эффекта, но допускается в целях совместимости с SQL и другими реализациями. Согласно стандарту SQL первое выражение `BY REF` обязательно, второе — нет. Следует отметить, что SQL стандарт подразумевает принятие функцией `xmlexists` в качестве первого аргумента выражение XQuery, тогда как PostgreSQL в настоящий момент поддерживает только XPath, являющийся подмножеством XQuery.

6.14.2.3. xml_is_well_formed

```
xml_is_well_formed(text)
xml_is_well_formed_document(text)
xml_is_well_formed_content(text)
```

Указанные функции проверяют, что переданный текст является «правильным» (well-formed) XML, и возвращают результат типа Boolean. `xml_is_well_formed_document` проверяет на «правильность» XML документ. `xml_is_well_formed_content` проверяет на «правильность» XML содержание. Поведение `xml_is_well_formed` зависит от конфигурационного параметра `xmloption`, принимающего значения `DOCUMENT` или `CONTENT`.

Это означает, что функция xml_is_well_formed удобна в случае проверки возможности успешного приведения к типу xml, тогда как две остальные функции удобны для проверки возможности успешного выполнения соответствующих вариантов XMLPARSE.

Пример

```
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 проверка включает и соответствие пространств имен.

6.14.3. Обработка XML

Для обработки значений типа `xml`, PostgreSQL поддерживает функциональность языка запросов `xpath` и `xpath_exists`, вычисляющие выражения XPath 1.0:

```
xpath(xpath, xml[, nsarray])
```

Функция `xpath` выполняет XPath-выражение xpath над XML-значением `xml`. Результатом вызова является массив XML-значений соответствующий набору XML-узлов, отобранных XPath-выражением.

Вторым аргументом должен быть «правильный» XML документ. Обычно он должен иметь один корневой элемент.

Третьим необязательным аргументом функции передается массив отображений пространства имен. Массив должен быть двухмерным, с размеров второго измерения равным 2. Первым элементом указывается имя пространства имен, вторым URI. Не требуется совпадения псевдонимов, заданных этим аргументом, с используемыми в самом XML-документе.

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 работы с пространством имен по умолчанию, возможен следующий вариант:

```
SELECT xpath('//mydefns:b/text()',
             '<a xmlns="http://example.com"><b>test</b></a>',
             ARRAY[ARRAY['mydefns', 'http://example.com']]);

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

```
xpath_exists(xpath, xml [, nsarray])
```

Функция `xpath_exists` является специализированной формой фунции `xpath`. Вместо возвращения отдельных XML значений, удовлетворяющих XPath, эта функция возвращает Boolean значение, показывающее, удовлетворяется ли запрос или нет. Функция

эквивалентна предикату XMLEXISTS, за исключением того, что поддерживает аргумент отображений пространства имен.

Пример

```
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)
```

6.14.4. Отображение таблиц в XML

Следующие функции предназначены для отображения содержимого таблиц реляционной базы данных в XML значения и могут рассматриваться как экспортирующие:

```
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)
```

Все функции в качестве результата возвращают значения типа xml.

table_to_xml отображает содержимое указанной таблицы. Тип аргумента regclass позволяет указывать таблицы в строковом виде, используя обычную нотацию, включая необязательное указание схемы или двойные кавычки. query_to_xml выполняет переданный запрос и отображает в XML результат его выполнения. cursor_to_xml выбирает заданное количество записей по указанному курсору. Поскольку результат выполнения перечисленных функций формируется в памяти, для больший таблиц предпочтительно применение последнего варианта.

Если значение tableforest FALSE, результирующий XML-документ выглядит следующим образом:

```
<tablename>
  <row>
    <columnname1>data</columnname1>
    <columnname2>data</columnname2>
  </row>

  <row>
    ...
  </row>
```

```
  ...
</tablename>
```

Если значение `tableforest TURE`, результатом является следующий XML-фрагмент:

```
<tablename>
  <columnname1>data</columnname1>
  <columnname2>data</columnname2>
</tablename>

<tablename>
  ...
</tablename>

...
```

При отсутствии заданного имени таблицы, при выполнении функций с запросом или курсором, в первом формате используется строка `table`, во втором `row`.

Выбор формата возлагается на пользователя. Первый формат соответствуют корректному XML-документу, что может быть необходимым для многих приложений. Второй формат более удобен при работе с курсорами (`cursor_to_xml`), когда полученные значения потом собираются в единый документ. Функции создания XML-содержимого были рассмотрены выше, для пробы варианта может быть использован отдельный вызов `xmlelement`.

Данные отображаются в XML аналогично описанной выше функции xmlelement.

Параметр `nulls` определяет необходимость включения в результат неопределенных значений. В случае задания этого параметра, неопределенные значения в столбцах представляются следующим образом:

```
<columnname xsi:nil="true"/>
```

где `xsi` префикс пространства имен XML для XML Schema Instance. В результирующее значение добавляются соответствующие объявления пространства имен. Если параметр nulls содержит ложь, столбцы с неопределенными значениями не выводятся.

Параметр `targetns` задает желаемое пространство имен результата. Если нет необходимости задавать пространство имен, может быть указана пустая строка. Следующие функции возвращают документ XML Schema, описывающий отображение сделанное соответствующими функциями, рассмотренными выше:

```
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)
```

Передача одинаковых параметров в соответствующие функции отображения в XML и создания документов XML Schema является существенным.

Следующие функции осуществляют отображение в XML и создание документов XML Schema виде одного объединенного документа (или фрагмента), что может быть удобно при создании документа, содержащего собственное описание:

```
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 существуют функции для аналогичного отображения в XML целиком схемы или даже 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)
```

П р и м е ч а н и е. Необходимо отметить, что вызовы указанных функций создают большое количество данных, которые должны быть размещены и обработаны в памяти. При создании отображения большой схемы или базы данных, считается более правильным использование потабличного отображения, или даже использование курсоров.

Результат отображения схемы выглядит следующим образом:

```
<schemaname>

table1-mapping

table2-mapping
```

```
...

</schemaname>
```

где формат отображения таблиц зависит от рассмотренного ранее параметра `tableforest`. Результат отображения базы данных выглядит так:

```
<dbname>

<schema1name>
  ...
</schema1name>

<schema2name>
  ...
</schema2name>

...

</dbname>
```

Далее в качестве примера использования указанных функций приведен XSLT-шаблон, преобразующий результат вызова `table_to_xml_and_xmlschema` в HTML документ, содержащий выравненное представления данных таблицы. Похожим образом результат вызова рассмотренных функций может быть преобразован в любой другой XML подобный формат.

Пример

XSLT шаблон для преобразования вывода SQL/XML в HTML

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.w3.org/1999/xhtml"
>

  <xsl:output method="xml"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
      doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
      indent="yes"/>

  <xsl:template match="/*">
```

```
    <xsl:variable name="schema" select="//xsd:schema"/>
    <xsl:variable name="tabletypename"
                  select="$schema/xsd:element[@name=name(current())]/@type"/>
    <xsl:variable name="rowtypename"
                  select="$schema/xsd:complexType[@name=
  $tabletypename]/xsd:sequence/xsd:element[@name='row']/@type"/>

    <html>
      <head>
        <title><xsl:value-of select="name(current())"/></title>
      </head>
      <body>
        <table>
          <tr>
            <xsl:for-each select="$schema/xsd:complexType[@name=
  $rowtypename]/xsd:sequence/xsd:element/@name">
              <th><xsl:value-of select="."/></th>
            </xsl:for-each>
          </tr>

          <xsl:for-each select="row">
            <tr>
              <xsl:for-each select="*">
                <td><xsl:value-of select="."/></td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

6.15. Функции и операторы для работы с JSON

В таблице 67 приведены доступные операторы для JSON типа (см. 5.14).

Таблица 67 – Операторы типов json и jsonb

Оператор	Тип правого операнда	Описание	Пример	Результат
->	int	Возвращает элемент массива JSON (нумерация начинается с 0)	'[1,2,3]'::json->2	{"c":"baz"}
->	text	Возвращает поле объекта JSON	'"a":1,"b":2'::json->'b'	{"b":"foo"}
-»	int	Возвращает элемент массива JSON как текст	'[1,2,3]'::json->2	3
-»	text	Возвращает поле объекта JSON как текст	'"a":1,"b":2'::json-»'b'	2
#>	text[]	Возвращает поле объекта JSON по указанному пути	'"a":[1,2,3],"b":[4,5,6]' ::json#>'a,2'	{"c": "foo"}
#»	text[]	Возвращает поле объекта JSON по указанному пути как текст	'"a":[1,2,3],"b":[4,5,6]' ::json#»'a,2'	3

Примечание. Существуют разновидности этих операторов для типов json и jsonb. Операторы получения поля/элемента/пути возвращают соответствующий тип по входящему типу, за исключением тех, для которых явно указано возвратить text. Операторы получения поля/элемента/пути возвращают NULL, а не ошибку, даже если входной JSON не имеет корректной структуры для совпадения с запросом; например, если не существует элемента.

Стандартные операторы сравнения (см. таблицу 6.2) могут быть использованы для jsonb, но не для json. Они руководствуются правилами сортировки для B-деревьев, описанных в 5.14.4.

Некоторые следующие операторы также могут быть использованы для jsonb (см. таблицу 68). Некоторые из этих операторов могут быть индексированы с помощью операторных классов jsonb. Полное описание операций содержания и существования jsonb см. 5.14.3. 5.14.4 описывает способы эффективной индексации данных jsonb.

Таблица 68 – Дополнительные операторы типа jsonb

Оператор	Тип правого операнда	Описание	Пример
@>	jsonb	Содержит ли левое значение правое?	'{"a":1, "b":2}'::jsonb @>'{"b":2}'::jsonb
<@	jsonb	Содержится ли левое значение в правом?	'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb
?	text	Ключ/элемент типа *string* содержится в JSON?	'{"a":1, "b":2}'::jsonb ? 'b'
?|	text[]	Какой-нибудь из ключей/элементов типа *string* содержится в JSON?	'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'c']
?&	text[]	Все ключи/элементы типа *string* содержится в JSON?	'["a", "b"]'::jsonb ?\& array['a', 'b']

В таблице 69 приведены функции создания и обработки данных JSON (см. 5.14).

Примечание. Многие из этих функций и операторов будет преобразовывать Unicode последовательности в JSON строках в соответствующий символ. Это не будет проблемой, если на вход передан jsonb, потому что преобразования уже были сделаны; но для типа json, это может привести к ошибкам, как это отмечалось в 5.14.

Примечание. В функциях json_populate_record, json_populate_recordset, json_to_record и json_to_recordset, тип привидения из JSON является "best effort"и не может привести к желаемым значениям для некоторых типов. Ключи JSON согласованы с одинаковыми именами столбцов в целевом тип строки. Поля JSON, которые не появляются в целевом тип строки, будут исключены из вывода, и значениям целевым столбцам, которые не им соответствуют, будет присвоено значение NULL.

Примечание. Поведение функций array_to_json и row_to_json такое же, как для to_json кроме опции pretty_print. Поведение, описанное в to_json также относится к каждому индивидуальному значению с помощью функций создания JSON.

Примечание. Расширение hstore имеет преобразование типа из hstore в json, таким образом, значения hstore представляются как объекты JSON, а не строковые значения.

Примечание. Значение null функции json_typeof нельзя путать с NULL SQL. Если json_typeof('null'::json) возвратит null, то json_typeof(NULL::json) возвратит SQL NULL.

В разделе описания агрегирующих функций 6.20 описана функция json_agg, эффективно агрегирующая значения JSON.

Таблица 69 – Функции создания и обработки данных JSON

Функция	Описание	Тип результата, пример, результат	
`array_to_json( anyarray [, pretty_bool]))`	Возвращает массив как JSON. Многомерный массив PostgreSQL становится массивом массивов JSON. Если `pretty_bool` равняется `TRUE`, между элементами первого уровня добавляется перевод строки	Тип	`json`
		Пример	`array_to_json( '1,5,99,100'::int[])`
		Результат	`[[1,5],[99,100]]`
`row_to_json( record [, pretty_bool])`	Возвращает `row` как JSON. Если `pretty_bool` равняется `TRUE`, между элементами первого уровня добавляется перевод строки	Тип	`json`
		Пример	`row_to_json( row(1,'foo'))`
		Результат	`{"f1":1,"f2":"foo"}`
`json_build_array( VARIADIC "any")`	Возвращает неоднородно-типизированный массив JSON из `VARIADIC` из списка аргументов.	Тип	`array`
		Пример	`json_build_array(1,2,'3',4,5)`
		Результат	`[1, 2, "3", 4, 5]`
`json_build_object( VARIADIC "any")`	Возвращает JSON объект из списка аргуементов `VARIADIC`. По соглашению, список аргументов состоит из чередующихся ключей и значений	Тип	`json`
		Пример	`json_build_object('foo',1,'bar',2)`
		Результат	`{"foo": 1, "bar": 2}`
`json_object( text[])`	Возвращает объект JSON из массива строк (типа `text`). Массив должен иметь либо ровно одно измерение с четным числом членов, в этом случае они принимаются в виде чередующихся пар ключ-значение, или двух измерениях, таких, что каждый внутренний массив имеет ровно два элемента, которые принимаются в виде ключ-значение.	Тип	`json`

		Пример	`json_object('{a, 1, b, "def c, 3.5}')`
			
			`json_object('{{a, 1},{b, "def"},{c, 3.5}}')}`

		Результат	`{"a": "1", "b": "def", "c": "3.5"}`
`json_object( keys text[], values text[])`	Этот вариант функции принимает ключи и значения попарно из двух отдельных массивов. В остальном она идентична функции с одним аргументом.	Тип	`json`
		Пример	`json_object('a, b', '1, 2')`
		Результат	`{"a": "1", "b": "2"}`

Продолжение таблицы 69

Функция	Описание	Тип результата, пример, результат	
`to_json( anyelement)`	Возвращает любое значение как JSON. Если тип данных не является встроенным, и существует функция приведения к `json`, будет вызвана соответствующая функция преобразования. В противном случае для любых типов, отличных от числовых, Boolean, или неопределенного значения, будет использовано текстовое представление, экранированное в соответствии с требованиями JSON.	Тип	`json`
		Пример	`to_json('Fred said "Hi."'::text)`
		Результат	`"Fred said \"Hi.\""`

`json_array_length(json)`
 	
`jsonb_array_length(jsonb)`
	Возвращает количество элементов в массиве JSON.	Тип	`int`

		Пример	`json_array_length( '[1,2,3,{"f1":1,"f2":[5,6]},4]')`
		Результат	`5`

`json_each(json)`
 	
`jsonb_each(jsonb)`
	Преобразует объект JSON в набор пар ключ/значение.	Тип	`SETOF key text, value json`
			
			`SETOF key text, value jsonb`

		Пример	`select * from json_each( '{"a":"foo", "b":"bar"}')`

		Результат	`key | value`
			`-----+-------`
			`a  | "foo"`
			`b  | "bar"`

`json_each_text(json)`
 	
`jsonb_each_text(jsonb)`
	Преобразует объект JSON в набор пар ключ/значение (типа `text`).	Тип	`SETOF key text, value json`

		Пример	`select * from json_each_text( '{"a":"foo", "b":"bar"}')`

		Результат	`key | value`
			`-----+-------`
			`a  | foo`
			`b  | bar`

Продолжение таблицы 69

Функция	Описание	Тип результата, пример, результат	

`json_extract_path (from_json json, VARIADIC path_elems text[])`
 	
`jsonb_extract_path (from_json jsonb, VARIADIC path_elems text[])`
	Возвращает объект JSON указанный в `path_elems`	Тип	`json`
			
			`jsonb`

		Пример	`json_extract_path( '{"f2":{"f3":1}, "f4":{"f5":99,"f6":"foo"}}','f4')`
		Результат	`{"f5":99,"f6":"foo"}`

`json_extract_path_text( from_json json, VARIADIC path_elems text[])`
 	
`jsonb_extract_path_text( from_json jsonb, VARIADIC path_elems text[])`
	Возвращает объект JSON указанный в `path_elems`	Тип	`text`

		Пример	`json_extract_path_text( '{"f2":{"f3":1}, "f4":{"f5":99,"f6":"foo"}}', 'f4', 'f6')`
		Результат	`foo`

`json_object_keys(json)`
 	
`jsonb_object_keys(jsonb)`
	Возвращает набор ключей объекта JSON. Отображается только внешний объект.	Тип	`SETOF text`

		Пример	`json_object_keys( {"f1":"abc", "f2":{"f3":"a", "f4":"b"}}')`

		Результат	`json_object_keys`
			`------------------`
			`f1`
			`f2`

Продолжение таблицы 69

Функция	Описание	Тип результата, пример, результат	

`json_populate_record(base anyelement, from_json json, [use_json_as_text bool=false])`
 	
`jsonb_populate_record(base anyelement, from_json jsonb, [use_json_as_text bool=false])`
	Преобразует объект `from_json` в `row`, чьи столбцы соответствуют типу записи, заданному в `base`. Столбцы в `base`, для которых отсутствуют ключи в `from_json` будут заполнены неопределенным значением. Если столбец задан более одного раза, будет использовано последнее значение.	Тип	`anyelement`

		Пример	`select * from json_populate_record(null::x, '{"a":1,"b":2}')`

		Результат	`a | b`
			`---+---`
			`1 | 2`

`json_populate_recordset(base anyelement, from_json json, [use_json_as_text bool=false])`
 	
`jsonb_populate_recordset(base anyelement, from_json jsonb, [use_json_as_text bool=false])`
	Преобразует объект `from_json` в набор, чьи столбцы соответствуют типу записи, заданному в `base`. Столбцы в `base`, для которых отсутствуют ключи в `from_json` будут заполнены неопределенным значением. Если столбец задан более одного раза, будет использовано последнее значение.	Тип	`SETOF anyelement`

		Пример	`select * from json_populate_recordset(null::x, '[{"a":1,"b":2},{"a":3,"b":4}]')`

		Результат	`a | b`
			`---+---`
			`1 | 2`
			`3 | 4`

Продолжение таблицы 69

Функция	Описание	Тип результата, пример, результат	

`json_array_-elements(json)`
 	
`json_array_-elements(jsonb)`
	Преобразует массив JSON в набор элементов JSON.	Тип	`SETOF json`
			
			`SETOF jsonb`

		Пример	`select * from ('[1,true, [2,false]]')`

		Результат	`value`
			`-----------`
			`1`
			`true`
			`[2,false]`

`json_array_-elements_-text(json)`
 	
`json_array_-elements_-text(jsonb)`
	Преобразует массив JSON в набор элементов типа `text`.	Тип	`SETOF text`

		Пример	`select * from json_array_elements_text('["foo", "bar"]')`

		Результат	`value`
			`-----------`
			`foo`
			`bar`

`json_-typeof(json)`
 	
`json_-typeof(jsonb)`
	Возвращает тип внешнего значения JSON ввиде строки типа `text`. Возможные типы: `object`, `array`, `string`, `number`, `boolean`, и `null`.	Тип	`text`

		Пример	`select json_typeof('-123.4')`

		Результат	`value`
			`-----------`
			`number`

`json_to_-record(json)`
 	
`json_to_-record(jsonb)`
	Возвращает произвольную запись из объекта JSON (см примечание ниже). Как и все функции, возвращающие `record`, в запросе должно быть явное указание на структуру записи с помощью конструкции `AS`.	Тип	`text`

		Пример	`select * from json_to_record ('{"a":1,"b":[1,2,3],"c":"bar"}') as x(a int, b text, d text)`

		Результат	`a |  b  | d`
			`---+---------+---`
			`1 | [1,2,3] |`

Окончание таблицы 69

Функция	Описание	Тип результата, пример, результат	

`json_to_recordset(json)`
 	
`json_to_recordset(jsonb)`
	Возвращает произвольный набор записей из массива JSON объектов (см примечание ниже). Как и все функции, возвращающие `record`, в запросе должно быть явное указание на структуру записи с помощью конструкции `AS`.	Тип	`text`

		Пример	`select * from json_to_recordset '[{"a":1,"b":"foo"}, {"a":"2","c":"bar"}]' as x(a int, b text)`

		Результат	`a | b`
			`---+-----`
			`1 | foo`
			`2 |`

6.16. Функции для управления последовательностями

Этот подраздел описывает функции PostgreSQL для работы со последовательностями (счетчиками). Последовательности (иначе называемые генераторами) представляют собой специальные таблицы, содержащие всего одну строку и создаваемые с помощью команды CREATE SEQUENCE. Они используются, как правило, для генерации уникальных идентификаторов строк в таблицах. Функции последовательностей, описанные в таблице 70, предоставляют простой, безопасный для использования в многопользовательском окружении, метод для получения последовательности числовых значений.

Таблица 70 – Функции последовательностей

Функция	Тип	Описание
currval(regclass)	bigint	Возвращает значение в последний раз полученное с помощью функции nextval для указанной последовательности
lastval()	bigint	Возвращает значение в последний раз полученное с помощью функции nextval для любой последовательности
nextval(regclass)	bigint	Продвигает счетчик последовательности и возвращает новое значение
setval(regclass, bigint)	bigint	Устанавливает текущее значение последовательности в новое
setval(regclass, bigint, boolean)	bigint	Устанавливает новое текущее значение и флаг is_called последовательности

Во всех функциях в таблице 70 имя последовательности задается как аргумент типа regclass, что соответствует просто OID последовательности из таблицы pg_class системного каталога. Возможно использование стандартного преобразования, путем указания имени последовательности в одинарных кавычках. Для совместимости с правилами обработки имен все функции перед использованием имен последовательностей конвертируют их к нижнему регистру (если они не заданы в двойных кавычках). Таким образом:

- nextval('foo') работает со счетчиком с именем — foo;
- nextval('FOO') работает со счетчиком с тем же именем — foo;
- nextval('"Foo"') работает со счетчиком с другим именем — Foo.

Имена последовательностей можно задавать с помощью полных имен:

- nextval('myschema.foo') работает с myschema.foo;
- nextval('"myschema".foo') работает с myschema.foo;
- nextval('foo') ищет путь для foo.

Информация о типе regclass приведена в разделе 5.18.

Примечание. В версиях PostgreSQL младше 8.1 указанные функции принимали

аргументы типа text, а не regclass, и преобразование в соответствующий OID осуществлялся при каждом вызове. В целях обратной совместимости сохранена возможность использования тестовых аргументов, но при этом производится неявное преобразование из типа text в тип regclass заранее.

При записи аргумента указанных функций в виде простой строки, он становится константой типа regclass, т. е. непосредственно OID, соответствующий первоначально заданному для последовательности, без учета последующего возможного переименования, изменения схемы и прочих дейтсвий. Подобное поведение называется «раннее связывание» и предпочтительно для задания значений по умолчанию для столбцов или представлений. Иногда требуется «позднее связывание», когда идентификатор последовательности определяется во время исполнения. Для достижения подобного поведения, достаточно явно задать тип аргумента text, вместо regclass:

```
nextval('foo'::text)
```

В этом случае foo будет определено во время исполнения.

Следует заметить, что позднее связывание является единственным способом в версиях PostgreSQL ранее 8.1, и следует использовать этот способ для сохранения работоспособности старых приложений. Имя счетчика может быть результатом какого-либо выражения, а не только текстовой константой.

Доступны следующие функции для работы с последовательностями:

1) nextval — продвигает значение счетчика к следующему значению и возвращает новое значение. Эта операция выполняется автоматически, при этом даже если несколько сессий запускают эту функцию параллельно, каждая из них получит уникальное значение.

Если последовательность была создана с параметрами по-умолчанию, последующие вызовы nextval будут возвращать последовательные значения, начиная с 1. Другое поведение может быть задано специальными параметрами команды CREATE SEQUENCE.

Примечание. Для того чтобы избежать блокировок транзакций, которые работают с одним и тем же счетчиком, функция nextval никогда не «откатывается». Таким образом, после получения значения оно будет считаться «использованным» в любом случае.

2) currval — возвращает значение счетчика, в последний раз полученное с помощью функции nextval для указанной последовательности в текущей сессии. Если при этом nextval ни разу не была вызвана, то эта функция будет завершена с ошибкой. Поскольку currval возвращает локальное для сессии значение, то результат ее вызова всегда предсказуем, даже если параллельно с этим счетчиком

работают другие сессии.

3) lastval — возвращает значение счетчика, в последний раз полученное с помощью функции nextval в текущей сессии. Функция идентична currval за исключением того, что учитывает вызов nextval любой последовательности. Если при этом nextval ни разу не была вызвана, то эта функция будет завершена с ошибкой.

4) setval — устанавливает счетчик в новое значение. Форма функции с двумя параметрами устанавливает is_called переменную в TRUE. Это означает, что следующий вызов nextval сначала увеличит значение счетчика, а затем вернет новое значение. В форме с тремя параметрами переменная is_called может быть установлена в TRUE или FALSE. При установке ее в FALSE последующий вызов nextval вернет установленное значение без его предварительного увеличения:

Пример

SELECT setval('foo', 42); следующий вызов nextval вернет 43

SELECT setval('foo', 42, true); тоже, что и выше

SELECT setval('foo', 42, false); следующий вызов nextval вернет 42

Примечание. Поскольку последовательности не поддерживают транзакций, «откат» для вызова функции setval невозможен.

6.17. Условные выражения

В данном пункте описываются условные выражения PostgreSQL, совместимые со стандартом SQL.

Примечание. Если ваши потребности превышают возможности рассматриваемых условных выражений, возможно следует рассмотреть вариант создания хранимых процедур на более выразительном языке программирования.

6.17.1. CASE

Выражение CASE является наиболее общим видом условного выражения, подобного выражениям IF и ELSE в других языках.

```
CASE WHEN <условие> THEN <результат>
     [WHEN ...]
     [ELSE <результат>]
END
```

CASE может использоваться только там, где допустимо использование скалярного выражения. <условие> — выражение, возвращающее результат типа boolean. Если его результат — TRUE, то тогда результатом выражения CASE является <результат> следующий за условием. Если — FALSE, то последующие выражения WHEN просматриваются таким же образом. Если <условие> всех выражений WHEN оказались FALSE, то результатом выра-

жения CASE станет <результат> выражения ELSE. Если же выражение ELSE отсутствует, то результатом выражения CASE станет неопределенное значение 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 всех выражений <результат> должны приводиться к единому типу. Простейшая форма выражения CASE является вариантом общего выражения:

```
CASE <выражение>
    WHEN <значение> THEN <результат>
    [WHEN ...]
    [ELSE <результат>]
END
```

<выражение> вычисляется и затем сравнивается со всеми <значениями> выражений WHEN пока оно не совпадет с одним из них. Если ни одно из <значений> не совпало, то <результат> выражения ELSE (или значение NULL) будет возвращено как результат всего выражения CASE. Эта форма выражения работает аналогично выражению switch в языках C/C++.

Таким образом, предыдущий пример может быть записан следующим образом:

```
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
```

Выражение CASE не вычисляет подвыражения, не являющиеся необходимыми для определения результата. Далее приведен пример, позволяющий избежать ошибки деления на ноль:

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

Примечание. Как описано в 1.2.14, существуют различные ситуации, в которых подвыражения выражения оцениваются в разное время, так что принцип «CASE вычисляет только необходимые подвыражения» ненадежно. Например, вычисление постоянной 1/0, приводит к делению на ноль во время планирования запроса, даже если оно находится в пределах CASE, поэтому не будет ошибки во время выполнения.

6.17.2. COALESCE

```
COALESCE(<значение1> [, ...])
```

Выражение COALESCE возвращает первое непустое NOTNULL из своих <значений>. Неопределенное значение возвращается только в случае, когда все аргументы являются неопределенными значениями. Эта функция часто используется для подстановки значений по умолчанию при извлечении данных для отображения.

Пример

```
SELECT COALESCE(description, short_description, '(none)') ...
```

Так же как и выражение CASE, COALESCE не вычисляет не нужные для определения результата подвыражения, так что аргументы правее первого NOTNULL аргумента не вычисляются. Данная стандартная для SQL функция обеспечивает возможности похожие на NVL и IFNULL, представленные в других СУБД.

6.17.3. NULLIF

```
NULLIF(<значение1>, <значение2>)
```

Выражение NULLIF возвращает неопределенное выражение NULL тогда и только то-

гда, когда <значение1> и <значение2> эквивалентны. Иначе возвращается <значение1>. Это выражение можно использовать для выполнения обратной выражению COALESCE операции.

Пример

```
SELECT NULLIF(value, '(none)') ...
```

Если value = '(none)' будет возвращено неопределенное значение, иначе значение первого аргумента.

6.17.4. GREATEST и LEAST

```
GREATEST(<значение> [, ...])
LEAST(<значение> [, ...])
```

Функции GREATEST и LEAST возвращают соответсвенно наибольшее и наименьшее значения из списка любого числа выражений. Выражения должны быть конвертируемы к общему типу данных, который будет типом результата. Значение NULL в списке игнорируется. Результат равен NULL тогда и только тогда, когда все выражения эквивалентны NULL.

Функции GREATEST и LEAST не являются стандартными для SQL. Некоторые другие СУБД заставляют данные функции возвращать NULL если любой из аргументов равен NULL, а не все аргументы.

6.18. Функции и операторы для работы с массивами

В таблице 71 приведены доступные для работы с массивами операторы.

Таблица 71 – Операторы для работы с массивами

Оператор	Описание	Пример	Результат
=	Равно	ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]	t
<>	Не равно	ARRAY[1,2,3] <> ARRAY[1,2,4]	t
<	Меньше	ARRAY[1,2,3] < ARRAY[1,2,4]	t
>	Больше	ARRAY[1,4,3] > ARRAY[1,2,4]	t
<=	Меньше либо равно	ARRAY[1,2,3] <= ARRAY[1,2,3]	t
>=	Больше либо равно	ARRAY[1,4,3] <= ARRAY[1,4,3]	t
@>	Содержит	ARRAY[1,4,3] @> ARRAY[3,1]	t
<@	Содержится в	ARRAY[2,7] <@ ARRAY[1,7,4,2,6]	t

Окончание таблицы 71

Оператор	Описание	Пример	Результат
&&	Перекрываются (содержат общие элементы)	ARRAY[1,4,3] && ARRAY[2,1]	t
||	Объединения массивов	ARRAY[1,2,3] || ARRAY[4,5,6]	{1,2,3,4,5,6}
||	Объединения массивов	ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]]	{{1,2,3},{4,5,6},{7,8,9}}
||	Объединение элемента с массивом	3 || ARRAY[4,5,6]	{3,4,5,6}
||	Объединение массива с элементом	ARRAY[4,5,6] || 7	{4,5,6,7}

Сравнение массивов производится путем поэлементного сравнения их содержимого, используя стандартные B-tree функции сравнения, соответствующие типу элементов. В многомерных массивах просмотр осуществляется в порядке строк (последние индексы более часто). В случае одинакового содержания массивов разной размерности, первое различие в размерностях определяется в порядке сортировки. Подобное поведение отличается от поведения в версиях PostgreSQL младше 8.2., где два массива считались равными при одинаковом содержимом, даже если они отличались размерностями или диапазонами измерений.

Подробнее поведение операторов для массивов рассмотрено в 5.15.

В таблице 72 приведены доступные для работы с массивами функции. Примеры и обсуждение их использования рассматриваются в 5.15.

Таблица 72 – Функции для работы с массивами

Функция	Тип, описание, пример, результат	
array_append(anyarray, anyelement)	Тип	anyarray
	Описание	Добавление элемента в конец массива
	Пример	array_append(ARRAY[1,2], 3)
	Результат	{1,2,3}
array_cat(anyarray, anyarray)	Тип	anyarray
	Описание	Объединение двух массивов
	Пример	array_cat(ARRAY[1,2,3], ARRAY[4,5])
	Результат	{1,2,3,4,5}
array_ndims(anyarray)	Тип	int
	Описание	Количество измерений массива
	Пример	array_ndims(ARRAY[[1,2,3], [4,5,6]])
	Результат	2

Продолжение таблицы 72

Функция	Тип, описание, пример, результат	
array_dims(anyarray)	Тип	text
	Описание	Текстовое представление размерности массива
	Пример	array_dims(ARRAY[[1,2,3], [4,5,6]])
	Результат	[1:2][1:3]
array_fill(anyelement, int[], [, int[]])	Тип	anyarray
	Описание	Возвращает массив заданной размерности, инициализированный указанным значением, при этом нижняя граница измерений может отличаться от 1
	Пример	array_fill(7, ARRAY[3], ARRAY[2])
	Результат	[2:4]={7,7,7}
array_length(anyarray, int)	Тип	int
	Описание	Возвращает длину указанного измерения
	Пример	array_length(array[1,2,3], 1)
	Результат	3
array_lower(anyarray, int)	Тип	int
	Описание	Возвращает нижнюю границу указанного измерения
	Пример	array_lower('[0:2]={1,2,3}'::int[], 1)
	Результат	0
array_prepend(anyelement, anyarray)	Тип	anyarray
	Описание	Добавление элемента в начало массива
	Пример	array_prepend(1, ARRAY[2,3])
	Результат	{1,2,3}
array_remove(anyarray, anyelement)	Тип	anyarray
	Описание	Удаление из массива всех элементов, равных заданному (массив должен быть одномерным) Примечание. (Доступно только в версии PostgreSQL 9.6.)
	Пример	array_remove(ARRAY[1,2,3,2], 2)
	Результат	{1,3}
array_replace(anyarray, anyelement, anyelement)	Тип	anyarray
	Описание	Замена всех элементов массива, равных заданному, на указанный Примечание. (Доступно только в версии PostgreSQL 9.6.)
	Пример	array_replace(ARRAY[1,2,5,4], 5, 3)
	Результат	{1,2,3,4}
array_to_string(anyarray, text [, text])	Тип	text
	Описание	Объединяет элементы массива в одну строку через указанный разделитель и необязательную NULL-строку
	Пример	array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*')
	Результат	1,2,3,*,5

Окончание таблицы 72

Функция	Тип, описание, пример, результат	
array_upper(anyarray, int)	Тип	int
	Описание	Возвращает верхнюю границу указанного измерения
	Пример	array_upper(ARRAY[1,2,3,4], 1)
	Результат	4
cardinality(anyarray)	Тип	int
	Описание	Возвращает общее число элементов массива или 0, если массив пуст. Примечание. Данная функция может быть использована только в СУБД версии 9.6.
	Пример	cardinality(ARRAY[[1,2],[3,4]])
	Результат	4
string_to_array(text, text [, text])	Тип	text[]
	Описание	Создание массива из текстовой строки с заданным разделителем и необязательной NULL-строкой
	Пример	string_to_array('xx~^~yy^~zz', '~^~', 'yy')
	Результат	{xx,NULL,zz}
unnest(anyarray)	Тип	setof anyelement
	Описание	Преобразование массива в набор строк
	Пример	unnest(ARRAY[1,2])
	Результат	1 2 (2 rows)
unnest(anyarray, anyarray, [, ...])	Тип	setof anyelement, anyelement [, ...]
	Описание	Преобразование нескольких массивов (возможно разных типов) в наборы строк. Это разрешено только с приминением условия FROM (см. 4.2.1). Примечание. Данная функция может быть использована только в СУБД версии 9.6.
	Пример	unnest(ARRAY[1,2],ARRAY['foo','bar','baz'])
	Результат	1 foo 2 bar NULL baz (3 rows)

При вызове string_to_array, если параметр delimiter равен NULL, каждый символ входной строки становится отдельным элементом в результирующем массиве. Если delimiter задан пустой строкой, вся входная строка возвращается как один элемент. В других случаях входная строка разбивается по каждому вхождению строки delimiter.

При вызове string_to_array, если параметр NULL-строки опущен или задан как NULL, ни одна из подстрок входной строки не будет заменена на NULL. При вызове array_to_string, если параметр NULL-строки опущен или задан как NULL, любой NULL-элемент массива просто пропускается и не отображается в выходной строке.

Примечание. Существует два отличия в поведении string_to_array в версиях PostgreSQL ранее 9.1. Во-первых возвращается пустой массив, а не NULL-значение, если входная строка является пустой. Во-вторых, если delimiter равен NULL, функция разделяет входную строку посимвольно, а не возвращает неопределенное значение.

Агрегирующая функция array_agg для работы с массивами рассматривается в 6.20.

6.19. Функции и операторы для работы с диапазонами

В таблице 73 приведены доступные для работы с диапазонами операторы.

Таблица 73 – Операторы для работы с диапазонами

Оператор	Описание	Пример	Результат
=	Равно	int4range(1,5) = '[1,4]'::int4range	t
<>	Не равно	numrange(1.1,2.2) <> numrange(1.1,2.3)	t
<	Меньше	int4range(1,10) < int4range(2,3)	t
>	Больше	int4range(1,10) > int4range(1,5)	t
<=	Меньше либо равно	numrange(1.1,2.2) <= numrange(1.1,2.2)	t
>=	Больше либо равно	numrange(1.1,2.2) >= numrange(1.1,2.0)	t
@>	Содержит диапазон	int4range(2,4) @> int4range(2,3)	t
@>	Содержит элемент	'[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp	t
<@	Диапазон содержится в	int4range(2,4) <@ int4range(1,7)	t
<@	Элемент содержится в	42 <@ int4range(1,7)	t
&&	Перекрываются (содержат общие элементы)	int8range(3,7) && int8range(4,12)	t
«	Полностью слева от	int8range(1,10) « int8range(100,110)	t
»	Полностью справа от	int8range(50,60) » int8range(20,30)	t
&<	Не расширяется вправо от	int8range(1,20) &< int8range(18,20)	t
&>	Не расширяется влево от	int8range(7,20) &> int8range(5,10)	t
-|-	Смежный с	numrange(1.1,2.2) -|- numrange(2.2,3.3)	t
+	Объединение	numrange(5,15) + numrange(10,20)	[5,20)
*	Пересечение	int8range(5,15) * int8range(10,20)	[10,15)
-	Разность	int8range(5,15) - int8range(10,20)	[5,10)

Простые операторы сравнения <, >, <=, и >= сначала проверяют нижние границы, и только если они равны — верхние. Эти операторы не слишком полезны, но используются для поддержки B-tree индексов на полях такого типа.

Операторы слева-от/справа-от/смежный-с всегда возвращают FALSE для пустых диапазонов, так что, пустой диапазон не считается идущи до или после любого другого.

Операторы объединения и пересечения вызывают ошибку, если результирующий диапазон не является единым (состоит из двух несвязных диапазонов), поскольку подобный диапазон не может быть представлен.

В таблице 74 приведены доступные для работы с диапазонами функции.

Таблица 74 – Функции для работы с диапазонами

Функция	Тип, описание, пример, результат	
lower(anyrange)	Тип	тип элемента диапазона
	Описание	Нижняя граница диапазона
	Пример	lower(numrange(1.1,2.2))
	Результат	1.1
upper(anyrange)	Тип	тип элемента диапазона
	Описание	Верхняя граница диапазона
	Пример	upper(numrange(1.1,2.2))
	Результат	2.2
isempty(anyrange)	Тип	boolean
	Описание	Проверка на пустоту
	Пример	isempty(numrange(1.1,2.2))
	Результат	false
lower_inc(anyrange)	Тип	boolean
	Описание	Проверка на конечность нижней границы
	Пример	lower_inc(numrange(1.1,2.2))
	Результат	true
upper_inc(anyrange)	Тип	boolean
	Описание	Проверка на конечность верхней границы
	Пример	upper_inc(numrange(1.1,2.2))
	Результат	false
lower_inf(anyrange)	Тип	boolean
	Описание	Проверка на бесконечность нижней границы
	Пример	lower_inf('(,)'::daterange)
	Результат	true
upper_inf(anyrange)	Тип	boolean
	Описание	Проверка на бесконечность верхней границы
	Пример	upper_inf('(,)'::daterange)
	Результат	true

Функции lower и upper возвращают неопределенное значение для пустого диапа-

зона, или если указанная граница является бесконечной. Функции lower_inc, upper_inc, lower_inf и upper_inf возвращают FALSE для пустых диапазонов.

6.20. Агрегирующие функции

Агрегирующие функции вычисляют результирующее значение из набора входных данных. В таблицах 75 и 76 перечислены встроенные агрегирующие функции. Общий синтаксис агрегатных функций рассмотрен в 1.2.7.

Таблица 75 – Основные агрегирующие функции

Функция	Тип аргумента, тип результата, описание	
array_agg(expression)	Тип аргумента	Любой
	Тип результата	Массив элементов типа аргумента
	Описание	Объединение входных значений в массив
avg(expression)	Тип аргумента	smallint, int, bigint, real, double precision, numeric, interval
	Тип результата	numeric для целочисленных аргументов, double precision для аргументов с плавающей точкой, для остальных совпадает с типом аргумента
	Описание	Среднее арифметическое всех входных значений
bit_and(expression)	Тип аргумента	smallint, int, bigint, bit
	Тип результата	Тот же что и у аргумента
	Описание	Побитовое И для всех NOT NULL аргументов
bit_or(expression)	Тип аргумента	smallint, int, bigint, bit
	Тип результата	Тот же что и у аргумента
	Описание	Побитовое ИЛИ для всех NOT NULL аргументов
bool_and(expression)	Тип аргумента	bool
	Тип результата	bool
	Описание	Истина, если все входные значения не являются неопределенными
bool_or(expression)	Тип аргумента	bool
	Тип результата	bool
	Описание	Истина, если хотя бы одно из значений не является неопределенным
count(*)	Тип аргумента	
	Тип результата	bigint
	Описание	Число входных значений
count(expression)	Тип аргумента	any
	Тип результата	bigint
	Описание	Число входных значений, для который выражение имеет определенное (NOT NULL) значение
every(expression)	Тип аргумента	bool
	Тип результата	bool
	Описание	Эквивалентно bool_and

Окончание таблицы 75

Функция	Тип аргумента, тип результата, описание	
`json_agg(any)`	Тип аргумента	`any`
	Тип результата	`json`
	Описание	Объединение входных записей в массив объектов JSON Примечание. Только в версии PostgreSQL 9.6
`json_object_agg(name, value)`	Тип аргумента	`(any, any)`
	Тип результата	`json`
	Описание	Объединение входных ключ-значение в массив объектов JSON Примечание. Только в версии PostgreSQL 9.6
`max(expression)`	Тип аргумента	Массив любого типа, `numeric, string, date/time`
	Тип результата	Тот же что и у аргумента
	Описание	Максимальное значение <выражения> для всех входных значений
`min(expression)`	Тип аргумента	Массив любого типа, `numeric, string, date/time`
	Тип результата	Тот же что и у аргумента
	Описание	Минимальное значение <выражения> для всех входных значений
`string_agg(expression, delimiter)`	Тип аргумента	`(text, text), (bytea, bytea)`
	Тип результата	Тот же что и у аргумента
	Описание	Объединение входных значений в строку с разделителями
`sum(expression)`	Тип аргумента	`smallint, int, bigint, real, double precision, numeric, interval`
	Тип результата	`bigint` для аргументов `smallint` или `int`, `numeric` для `bigint`, `double precision` для аргументов с плавающей точкой, для остальных совпадает с типом аргумента
	Описание	Сумма <выражений> всех входных значений
`xmlagg(expression)`	Тип аргумента	`xml`
	Тип результата	`xml`
	Описание	Объединение XML-значений (смотри также 6.14.1.7)

Следует заметить, что за исключением функции `count` все функции возвращают неопределенное (`NULL`) значение на пустом множестве входных значений (например, при отсутствии выбранных строк). В частности, функция `sum` при отсутствии суммируемых значений возвращает `NULL`, а не ноль, как этого можно было бы ожидать, а `array_agg` возвращает неопределенное значение вместо пустого массива при отсутствии строк на входе. Чтобы в подобном случае вернуть ноль или пустой массив, можно использовать функцию `COALESCE`.

Примечание. Агрегирующие функции дли типа `Boolean bool_and` и `bool_or` соответствуют стандартным агрегирующим функциям SQL `every` и `any` или `some`. Что касается `any` и `some`, следующее выражение может быть неоднозначным при использовании

стандартного синтаксиса:

```
SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
```

В этом случае, если выражение возвращает одну строку, конструкция ANY может быть рассмотрена как относящаяся к выражению SELECT или как агрегирующая функция. Следовательно, стандартные имена не могут быть применены для подобного агрегирования.

Примечание. Пользователи, привыкшие работать с другими СУБД, могут быть удивлены производительностью агрегирующей функции count, примененной к целой таблице. Запрос приведенного ниже вида будет выполнен PostgreSQL как полное сканирование указанной таблицы.

```
SELECT count(*) FROM sometable;
```

Результат агрегирующих функции array_agg, json_agg, string_agg и xmlagg, так же как и похожих определенные пользователем функций, сильно зависит от порядка входных значений. В текущей реализации порядок входных значений принципиально не задается. В тоже время могут быть использованы стандартные возможности сортировки вложенных запросов.

Пример

```
SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;
```

Правда подобный синтаксис не поддерживается стандартом SQL и может быть непереносим на другие системы.

В таблице 76 приведены агрегирующие функции, обычно используемые в статистическом анализе. (Указанные функции рассматриваются отдельно, чтобы не смешивать их с обычными часто используемыми.) Под символом N подразумевается количество входных значений, не являющихся неопределенными. В любом случае неопределенное значение возвращается в случае отсутствия смысла в вычислениях, например, когда N = 0.

Таблица 76 – Статистические агрегирующие функции

Функция	Тип аргумента, тип результата, описание	
corr(Y, X)	Тип аргумента	double precision
	Тип результата	double precision
	Описание	Коэффициент корреляции
covar_pop(Y, X)	Тип аргумента	double precision
	Тип результата	double precision
	Описание	Ковариация совокупности
covar_samp(Y, X)	Тип аргумента	double precision
	Тип результата	double precision
	Описание	Выборочная ковариация

Продолжение таблицы 76

Функция	Тип аргумента, тип результата, описание	
regr_avgx(Y, X)	Тип аргумента	double precision
	Тип результата	double precision
	Описание	Среднее независимых переменных (sum(X)/N)
regr_avgy(Y, X)	Тип аргумента	double precision
	Тип результата	double precision
	Описание	Среднее зависимых переменных (sum(X)/N)
regr_count(Y, X)	Тип аргумента	double precision
	Тип результата	bigint
	Описание	Коэффициент корреляции
regr_intercept(Y, X)	Тип аргумента	double precision
	Тип результата	double precision
	Описание	Отрезок на оси y, получаемый решением уравнения линейной парной регрессии, заданной парой (X,Y) мат. методом подбора наименьших квадратов
regr_r2(Y, X)	Тип аргумента	double precision
	Тип результата	double precision
	Описание	Квадрат коэффициента корелляции
regr_slope(Y, X)	Тип аргумента	double precision
	Тип результата	double precision
	Описание	Наклон при пересечении оси y, получаемый решением уравнения линейной парной регрессии, заданной парой (X,Y) методом наименьших квадратов
regr_sxx(Y, X)	Тип аргумента	double precision
	Тип результата	double precision
	Описание	sum(X^2) - sum(X)^2/N («сумма квадратов» независимых переменных)
regr_sxy(Y, X)	Тип аргумента	double precision
	Тип результата	double precision
	Описание	sum(X*Y) - sum(X) * sum(Y)/N («сумма произведений» независимой переменной)
regr_syy(Y, X)	Тип аргумента	double precision
	Тип результата	double precision
	Описание	sum(Y^2) - sum(Y)^2/N («сумма квадратов» зависимых переменных)
stddev(expression)	Тип аргумента	smallint, int, bigint, real, double precision, или numeric
	Тип результата	double precision для аргументов с плавающей точкой, для остальных numeric
	Описание	Устаревший аналог stddev_samp
stddev_pop(expression)	Тип аргумента	smallint, int, bigint, real, double precision, или numeric
	Тип результата	double precision для аргументов с плавающей точкой, для остальных numeric
	Описание	Стандартное отклонение для совокупности

Окончание таблицы 76

Функция	Тип аргумента, тип результата, описание	
stddev_samp(expression)	Тип аргумента	smallint, int, bigint, real, double precision, ил230pти numeric
	Тип результата	double precision для аргументов с плавающей точкой, для остальных numeric
	Описание	Выборочное стандартное отклонение
variance(expression)	Тип аргумента	smallint, int, bigint, real, double precision, или numeric
	Тип результата	double precision для аргументов с плавающей точкой, для остальных numeric
	Описание	Устаревший аналог var_samp
var_pop(expression)	Тип аргумента	smallint, int, bigint, real, double precision, или numeric
	Тип результата	double precision для аргументов с плавающей точкой, для остальных numeric
	Описание	Стандартное расхождение совокупности (квадрат стандартного отклонения)
var_samp(expression)	Тип аргумента	smallint, int, bigint, real, double precision, или numeric
	Тип результата	double precision для аргументов с плавающей точкой, для остальных numeric
	Описание	Выборочное расхождение совокупности (квадрат стандартного отклонения)

Примечание. Сортирующие и гипотетические агрегирующие функции существуют только в СУБД версии 9.6.

В таблице 77 представлены некоторые агрегирующие функции, которые используют синтакс *order-set aggregate*. Эти функции иногда называют «обратными функциями распределения».

Таблица 77 – Сортирующие агрегирующие функции

Функция	Типы аргументов, агрегируемые типы, тип результата, описание	
mode() WITHIN GROUP (ORDER BY sort_- expression)	Типы аргументов	(отсутсвуют)
	Агрегируемые типы	Любые сортируемые типы
	Тип результата	Такой же, как и тип сортируемого выражения
	Описание	Возвращает наиболее часто встречающееся значения (произвольно выбирает первый, если есть несколько одинаково частых значений)

Окончание таблицы 77

Функция	Типы аргументов, агрегируемые типы, тип результата, описание	
`percentile_cont(fraction) WITHIN GROUP (ORDER BY sort_expression)`	Типы аргументов	`double precision`
	Агрегируемые типы	`double precision` или `interval`
	Тип результата	Такой же, как и тип сортируемого выражения
	Описание	Непрерывной процентиль: возвращает значение, соответствующее указанной фракции в запросе, интерполируя между соседними входными элементов, если это необходимо
`percentile_cont(fractions) WITHIN GROUP (ORDER BY sort_expression)`	Типы аргументов	`double precision[]`
	Агрегируемые типы	`double precision` или `interval`
	Тип результата	Массив типа сортируемого выражения
	Описание	Кратный непрерывный процентиль: возвращает массив результатов, соответствующих параметру `fractions`, при этом каждый ненулевой элемент заменяется значением соответствующего этому процентиля
`percentile_disc(fraction) WITHIN GROUP (ORDER BY sort_expression)`	Типы аргументов	`double precision`
	Агрегируемые типы	Любые сортируемые типы
	Тип результата	Такой же, как и тип сортируемого выражения
	Описание	Дискретный процентиль: возвращает первое значение, чье положение при упорядочении больше, либо равно указанному параметру `fraction`
`percentile_disc(fractions) WITHIN GROUP (ORDER BY sort_expression)`	Типы аргументов	`double precision[]`
	Агрегируемые типы	Любые сортируемые типы
	Тип результата	Массив типа сортируемого выражения
	Описание	Кратный дискретный процентиль: возвращает массив результатов, соответствующих параметру `fractions`, при этом каждый ненулевой элемент заменяется входным значением, соответствующим этому процентелю

Все агрегирующие функции, приведенные в таблице 77 игноририруют нулевые значения при сортировки входных значений. В тех функциях, которые принимают параметр

fraction, он должен иметь значение 0 и 1; иначе будет выводиться ошибка. Тем не менее, при нулевом значении fraction просто возвращается нулевой результат.

Таблица 78 – Гипотетические агрегирующие функции

Функция	Типы аргументов, агрегируемые типы, тип результата, описание	
rank(args) WITHIN GROUP (ORDER BY sorted_args)	Типы аргументов	VARIADIC "any"
	Агрегируемые типы	VARIADIC "any"
	Тип результата	bigint
	Описание	Оценка гипотетической строки (с разрывами для повторяющихся строк)
dense_rank(args) WITHIN GROUP (ORDER BY sorted_args)	Типы аргументов	VARIADIC "any"
	Агрегируемые типы	VARIADIC "any"
	Тип результата	bigint
	Описание	Оценка гипотетической строки (без разрыва)
percent_rank(args) WITHIN GROUP (ORDER BY sorted_args)	Типы аргументов	VARIADIC "any"
	Агрегируемые типы	VARIADIC "any"
	Тип результата	double precision
	Описание	Относительная оценка гипотетического подряд, начиная от 0 до 1
cume_dist(args) WITHIN GROUP (ORDER BY sorted_args)	Типы аргументов	VARIADIC "any"
	Агрегируемые типы	VARIADIC "any"
	Тип результата	double precision
	Описание	Относительная оценка гипотетического подряд, начиная от *1/N* до 1

Каждая из агрегирующих функций, перечисленных в таблице 78 связана с оконной функцией с таким же названием, определенной в 6.21. В каждом случае совокупным результатом понимается значение такое, что соответствующая функция окна вернулся бы к «гипотетической», построенной по строке args, если такие строки были добавлены в отсортированном виде, вычисленных из sorted_args.

Для каждого из этих агрегирующих функций количество аргументов, представленных

в args, должны соответствовать количеству и типам аргументов агрегированных данных в sorted_args. В отличие от большинства встроенных агрегирущих функций, эти функции не строгие, то есть они не выдают ошибки при содержании нулей в строках. Нулевые значения сортируются в соответствии с правилом, указанным оператором ORDER BY.

6.21. Оконные функции

Оконные функции обеспечивают возможность выполнения вычислений на наборах строк, относящихся к строке текущего запроса.

Встроенные оконные функции приведены в таблице 79. Следует отметить, что указанные функции должны вызываться с использованием синтаксиса оконных функций, требующего наличия выражения OVER.

В дополнение к рассматриваемым функциям любые встроенные или определенные пользователем агрегирующие (но не сортирующие или гипотетические) функции могут быть использованы как оконные функции. Перечень встроенных агрегирующих функций приведен в 6.20. Агрегирующие функции действуют как оконные функции только посредством выражения OVER следующего за вызовом функции. В противном случае они действуют как обычные агрегирующие функции.

Таблица 79 – Основные оконные функции

Функция	Тип результата	Описание
row_number()	bigint	Число текущих строк в выборке, начиная с 1
rank())	bigint	Ранг текущей строки с промежутками, тоже самое, что и row_number для первой равноправной строки
dense_rank()	bigint	Ранг текущей строки без промежутков. Функция подсчитывает равноправные группы
percent_rank()	double precision	Относительный ранг текущей строки: (ранг - 1)/(общее количество строк - 1)
come_dist()	double precision	Относительный ранг текущей строки: (число строк предшествующих или равноправных текущей строке)/(общее количество строк)
ntitle(num_buckets integer)	integer	Целочисленное ранжирование от 1 до значения аргумента разделяющее выборку на максимально эквивалентные части
lag(value any [, offset integer [, default any]]))	Тот же, что и у аргумента	Возвращает значение вычисленное для строки отстоящей на offset строк перед текущей строкой выборки. Если нет такой строки, то возвращается значение по умолчанию default. И offset и значение по умолчанию default вычисляются с учетом текущей строки. Если offset и default не указаны, то offset полагается равным 1, а default равным NULL
lead(value any [, offset integer [, default any]]))	Тот же, что и у аргумента	Возвращает значение вычисленное для строки отстоящей на offset строк после текущей строкой выборки. Если нет такой строки, то возвращается значение по умолчанию default. И offset и значение по умолчанию default вычисляются с учетом текущей строки. Если offset и default не указаны, то offset полагается равным 1, а default равным NULL

Окончание таблицы 79

Функция	Тип результата	Описание
first_value(value any)	Тот же, что и у аргумента	Возвращает значение вычисленное для строки, которая является первой строкой в оконной конструкции
last_value(value any)	Тот же, что и у аргумента	Возвращает значение вычисленное для строки, которая является последней строкой в оконной конструкции
nth_value(value any)	Тот же, что и у аргумента	Возвращает значение вычисленное для строки, которая является nth строкой в оконной конструкции, начиная с 1. NULL если нет такой строки

Все приведенные в таблице 79 функции зависят от порядка сортировки, заданного выражением ORDER BY, при определении ассоциированного окна. Строки, не указанные явно в выражении ORDER BY, считаются равноправными. Четыре функции ранжирования определены таким образом, что бы выдавать одинаковый результат для любых двух равноправных строк.

Следует отметить, что first_value, last_value и nth_value оперируют только со строками в пределах «оконной конструкции», которая по умолчанию содержит строки начиная с отсортированной порции и завершая последней равноправной строкой для текущей строки. Существует вероятность получения бесполезного результата от функции nth_value и особенно от функции last_value. Существует возможность переопределить «оконную конструкцию» посредством добавления соответствующей спецификации (RANGE или ROW) к выражению OVER.

В случае использования агрегирующих функций как оконных гарантируется, что агрегирования осуществляется в пределах конструкции окна для текущей строки. Для получения агрегирования в пределах всего результата запроса необходимо опустить ORDER BY или использовать ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING. Агрегирование использующее ORDER BY и определение конструкции окна по умолчанию приведет поведению типа «текущая сумма».

Примечание. Стандарт SQL определяет значения RESPECT NULLS или IGNORE NULLS опции для lead, lag, first_value, last_value, и nth_value. В СУБД PostgreSQL это не реализовано: поведение всегда соответствует поведению определенному стандартом по умолчанию для значения опции RESPECT NULLS. Аналогично не реализованы предусмотренные стандартом значения опции FROM FIRST и FROM LAST для nth_value: поддерживается только поведение определяемое стандартом для значения опции FROM FIRST. Для получения результата соответствующего значению опции FROM LAST может быть использована сортировка ORDER BY.

6.22. Выражения для подзапросов

Данный пункт описывает описывает совместимые со стандартом SQL выражения для подзапросов, используемые в PostgreSQL. Все они возвращают результат типа boolean.

6.22.1. EXISTS

```
EXISTS ( <подзапрос> )
```

Аргументом функции EXISTS может быть произвольный запрос SELECT (называемый подзапросом, поскольку он выступает как часть полного запроса). Если в результате выполнения подзапроса возвращается хотя бы одна строка, результатом функции EXISTS является истина, в противном случае возвращается ложь.

Подзапрос может ссылаться на значения из общего запроса, которые выступают в качестве констант для каждого отдельного вычисления подзапроса.

Подвыражение вычисляется только для того, чтобы определить будет ли возвращена хотя бы одна строка. Поэтому не стоит пытаться выполнять таким образом подзапросы, которые должны иметь некоторое побочное значение (такое, например, как вызов последовательности функций), поскольку нельзя предсказать результат подобных подзапросов.

Поскольку результат этой функции зависит от самого факта возврата или невозврата строк и не зависит от их содержания, то как правило подобные запросы записываются в виде EXISTS(SELECT 1 WHERE ...). Из этого правила, однако, имеются исключения (например, подзапросы, использующие INTERSECT).

Следующий простой пример: поход на простое объединение по столбцу col2, однако выводит не больше чем одну строку для каждой строки таблицы tab1, даже если она соответствует более чем одной строке таблицы tab2:

```
SELECT col1
FROM tab1
WHERE EXISTS(SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

6.22.2. IN

```
<выражение> IN (<подзапрос>)
```

Справа в круглых скобках записывается подзапрос, который должен вернуть в точности один столбец. Слева записывается выражение, которое вычисляется и сравнивается со строками, возвращенными подзапросом. Результатом выражения IN будет истина, если найдется хоть одна строка, значение которой совпадет со значением выражения. Если не совпадет ни одна строка результатом будет ложь (включая частный случай, когда подзапрос не возвращает ни одной строки).

Если при этом <выражение> слева вычисляется в неопределенное значение (NULL), или для него не находится справа ни одного эквивалентного значения и по крайней мере одно значение справа является так же неопределенным, то все выражение IN имеет

неопределенное (а не ложь) значение. Это соответствует правилам стандарта SQL для вычисления логических выражений.

Как и в случае EXISTS нельзя рассчитывать, что подзапрос будет вычислен полностью.

Пример

```
row_constructor IN (<подзапрос>)
```

Левая часть этой формы IN представляет собой конструктор строки, описанный в 1.2.13. Правая часть является подзапросом, который должен возвращать в точности столько же столбцов, сколько входит в описания строки слева. Левое выражение вычисляется и сравниваются построчно со всеми строками, возвращенными подзапросом справа. Все выражение IN возвращает истину, если при этом было найдено хотя бы одно совпадение. Если не совпадает ни одна строка, результатом будет ложь (включая частный случай, когда подзапрос не возвращает ни одной строки).

Как обычно значение NULL вычисляются по SQL-правилам логических операций. Строки считаются эквивалентными, если все соответствующие поля равны и не содержат неопределенных значений, и не эквивалентными, если есть хотя бы одно отличающееся поле не содержащее неопределенного значения, в противном случае результатом является неопределенное значение. Если все результаты сравнения строк являются неэквивалентными или неопределенными, и хотя бы одно неопределенном, результатом выражения IN является неопределенное значение.

6.22.3. NOT IN

```
<выражение> NOT IN (<подзапрос>)
```

Подзапрос справа должно возвращать в точности один столбец. выражение слева вычисляется и сравнивается с записями, возвращенными подзапросом, и в случае отсутствия совпадений результатом всего выражения является истина (включая частный случай когда подзапрос не возвращает строк). Если было найдено хоть одно совпадение, то результатом выражения является ложь.

При этом нужно иметь в виду, что если <выражение> вычисляется в NULL или если для него не нашлось справа эквивалентного значения, и хотя бы одно из этих значений является значением NULL, то все выражение имеет значение NULL, а не истина, как можно было бы ожидать. Это соответствует правилам стандарта SQL для вычисления логических выражений.

Как и в случае EXISTS нельзя рассчитывать, что подзапрос будет вычислен полностью.

Пример

```
row_constructor NOT IN (<подзапрос>)
```

Левая часть этой формы NOT IN представляет собой конструктор строки, описанный в 1.2.13. Правая часть является подзапросом, который должен возвращать в точности столько же столбцов, сколько входит в описания строки слева. Левое выражение вычисляется и сравниваются построчно со всеми строками, возвращенными подзапросом справа. Все выражение NOT IN возвращает истину, если при этом не было найдено ни одного совпадения (включая частный случай, когда подзапрос не возвращает ни одной строки). Если совпадает хотя бы одна строка, результатом будет ложь.

Как обычно значения NULL вычисляются по SQL-правилам логических операций. Строки считаются эквивалентными, если все соответствующие поля равны и не содержат неопределенных значений, и не эквивалентными, если есть хотя бы одно отличающееся поле не содержащее неопределенного значения, в противном случае результатом является неопределенное значение. Если все результаты сравнения строк являются неэквивалентными или неопределенными, и хотя бы одно неопределенном, результатом выражения NOT IN является неопределенное значение.

6.22.4. ANY/SOME

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

Подзапрос в правой части должен возвращать в качестве результата один столбец. Результат вычисления выражения слева сравнивается со строками результата выполнения правого подзапроса, с помощью оператора, который должен возвращать результат типа boolean. Результат выражения ANY будет истиной, если при выполнении оператора был получен хотя бы один результат TRUE. Если такого результата не было найдено, то результатом выражения будет ложь (включая частный случай, когда подзапрос не возвращает ни одной строки).

SOME является синонимом ANY. Выражение IN эквивалентно = ANY.

Если сравнение не успешно и хотя бы для одной строки подзапроса был получен неопределенный результат NULL, то результатом всего выражения ANY будет неопределенное значение, а не ложь. Это соответствует правилам стандарта SQL для вычисления логических выражений.

Как и в случае EXISTS нельзя рассчитывать, что подзапрос будет вычислен полностью.

Пример

```
row_constructor operator ANY (subquery)
row_constructor operator SOME (subquery)
```

Левая часть этой формы ANY представляет собой конструктор строки, описанный в 1.2.13. Правая часть является подзапросом, который должен возвращать в точности столько же столбцов, сколько входит в описания строки слева. Левое выражение вычисляется и сравниваются построчно со всеми строками, возвращенными подзапросом справа, используя указанный оператор. Все выражение ANY возвращает истину, если при этом было найдено хотя бы одно успешное (TRUE) выполнение оператора. Если не совпадает ни одна строка, результатом будет ложь (включая частный случай, когда подзапрос не возвращает ни одной строки). Результатом всего выражения ANY будет неопределенное значение, если все сравнения не успешны, и хотя бы для одной строки подзапроса был получен неопределенный результат NULL. В 6.23.5 приведены правила построчного сравнения.

6.22.5. ALL

```
expression operator ALL (subquery)
```

Подзапрос в правой части должен возвращать в качестве результата один столбец. Результат вычисления выражения слева сравнивается со строками результата выполнения правого подзапроса, с помощью оператора, который должен возвращать результат типа boolean. Результат выражения ALL будет истиной, если при выполнении оператора все сравнения дали результат TRUE (включая частный случай, когда подзапрос не возвращает ни одной строки). Если хотя бы одно сравнения вернуло ложь, то результатом всего выражения будет ложь. Результатом всего выражения ALL будет неопределенное значение, если не было ни одного неуспешного сравнения, и хотя бы для одной строки подзапроса был получен неопределенный результат NULL. Выражение NOT IN эквивалентно <> ALL. Как и в случае EXISTS нельзя рассчитывать, что подзапрос будет вычислен полностью.

Пример

```
row_constructor operator ALL (subquery)
```

Левая часть этой формы ALL представляет собой конструктор строки, описанный в 1.2.13. Правая часть является подзапросом, который должен возвращать в точности столько же столбцов, сколько входит в описания строки слева. Левое выражение вычисляется и сравниваются построчно со всеми строками, возвращенными подзапросом справа, используя указанный оператор. Результат выражения ALL будет истиной, если при выполнении оператора все сравнения дали результат TRUE (включая частный случай, когда подзапрос не возвращает ни одной строки). Если хотя бы одно сравнения вернуло ложь, то результатом всего выражения будет ложь. Результатом всего выражения ALL будет неопределенное значение, если не было ни одного неуспешного сравнения, и хотя бы для одной строки подзапроса был получен неопределенный результат NULL. В 6.23.5 приведены правила сравнения строк.

6.22.6. Построчное сравнение

```
row_constructor operator (subquery )
```

Левая часть представляет собой конструктор строки, описанный в 1.2.13. Правая часть является подзапросом, который должен возвращать в точности столько же столбцов, сколько входит в описания строки слева. Более того подзапрос не должен возвращать более одной строки. (Если не возвращается ни одной строки, результат устанавливается в неопределенное значение.) Левое выражение вычисляется и сравниваются построчно со строкой, возвращенной подзапросом справа, используя указанный оператор.

В 6.23.5 приведены правила сравнения строк.

6.23. Сравнение строк и массивов

Раздел описывает некоторые специальные конструкции для реализации множественного сравнения между группам значений. Форма этих конструкций синтаксически схожа с рассмотренными ранее, за исключением того, что в ней не используются подзапросы. Формы, содержащие выражения с массивами, являются расширениями PostgreSQL, остальные совместимы со стандартом SQL. Все рассмотренные в разделе формы выражений возвращают результат типа Boolean (TRUE/FALSE).

6.23.1. IN

```
<выражение> IN (<значение>[, ...])
```

С правой стороны в этой форме функции IN в круглых скобках задается список скалярных выражений. Результат всего выражения истина, если выражение слева окажется эквивалентным какому-либо из выражений справа. Это сокращенная форма записи для:

```
<выражение> = <значение1>
OR
<выражение> = <значение2>
OR
...
```

Если при этом выражение слева вычисляется в неопределенное значение (NULL), или для него не находится справа ни одного эквивалентного значения и по крайней мере одно значение справа является так же неопределенным, то все выражение IN имеет неопределенное (а не FALSE) значение. Это соответствует правилам стандарта SQL для вычисления логических выражений.

6.23.2. NOT IN

```
<выражение> NOT IN (<значение>[, ...])
```

С правой стороны в этой форме функции NOT IN в круглых скобках задается список скалярных выражений. Результат всего выражения истина, если выражение слева не

эквивалентно ни одному из выражений справа. Это сокращенная форма записи для:

```
<выражение> <> <значение1>
AND
<выражение> <> <значение2>
AND
...
```

Если при этом выражение слева вычисляется в неопределенное значение (NULL), или для него не находится справа ни одного эквивалентного значения и по крайней мере одно значение справа является так же неопределенным, то все выражение NOT IN имеет неопределенное (а не TRUE) значение. Это соответствует правилам стандарта SQL для вычисления логических выражений.

Примечание. x NOT IN y эквивалентно NOT (x IN y) во всех случаях. Однако при работе с неопределенными значениями может потребоваться больше внимания, чем при использовании выражения IN, рекомендуется по возможности использовать положительные условия.

6.23.3. ANY/SOME (массив)

```
expression operator ANY (array expression)
expression operator SOME (array expression)
```

Выражение в правой части должно возвращать значение типа array. Результат вычисления выражения слева сравнивается с каждым элементом массива с помощью оператора, который должен возвращать результат типа boolean. Результат выражения ANY будет истиной, если при выполнении оператора был получен хотя бы один результат TRUE. Если такого результата не было найдено, то результатом выражения будет ложь (включая частный случай, когда массив не содержит элементов).

Если выражение справа представляет собой неопределенное значение, результатом выражения ANY так же будет неопределенное значение. Если выражение слева вычисляется в неопределенное значение то результат естественным образом тоже становиться неопределенным значением (хотя операторы нестрогого сравнения могут выдавать и иной результат). Так же, если массив из правого выражения содержит какой-нибудь элемент с неопределенным значением, и не одного успешного сравнения не было осуществлено, результатом выражения ANY будет неопределенное значение, а не ложь (опять же для операторов строгого сравнения). Это соответствует правилам стандарта SQL для вычисления логических выражений.

SOME является синонимом ANY.

6.23.4. ALL (массив)

```
expression operator ALL (array expression)
```

Выражение в правой части должно возвращать значение типа array. Результат вычисления выражения слева сравнивается с каждым элементом массива с помощью оператора, который должен возвращать результат типа `boolean`. Результат выражения `ALL` будет истиной, если при выполнении оператора все сравнения дали результат `TRUE` (включая частный случай, когда подзапрос не возвращает ни одной строки). Если хотя бы одно сравнения вернуло ложь, то результатом всего выражения будет ложь.

Если выражение справа представляет собой неопределенное значение, результатом выражения `ALL` так же будет неопределенное значение. Если выражение слева вычисляется в неопределенное значение то результат естественным образом тоже становиться неопределенным значением (хотя операторы нестрогого сравнения могут выдавать и иной результат). Так же, если массив из правого выражения содержит какой-нибудь элемент с неопределенным значением, и не было выполнено неуспешного сравнения, результатом выражения `ALL` будет неопределенное значение, а не истина (опять же для операторов строгого сравнения). Это соответствует правилам стандарта SQL для вычисления логических выражений.

6.23.5. Сравнение строк

`row_constructor operator row_constructor`

Обе части представляют собой конструктор строки, описанный в 1.2.13. Оба значения типа строка должны содержать одинаковое количество полей. Обе части вычисляются и затем построчно сравниваются. Построчное сравнение поддерживается для операторов =, <>, <, <=, > или >=, или похожих по смыслу. Точнее, оператор может быть использован для построчного сравнения если он входит в класс операторов `B-Tree`, или является обратным для оператора `B-Tree` =.

Операторы = и <> работают несколько отлично от остальных. Строки считаются эквивалентными, если все соответствующие поля равны и не содержат неопределенных значений, и не эквивалентными, если есть хотя бы одно отличающееся поле не содержащее неопределенного значения, в противном случае результатом является неопределенное значение.

В случае операторов <, <=, > и >=, сравнение осуществляется слева-направо до первой пары неравных или содержащих неопределенное значение элементов. Если хотя бы один из них имеет неопределенное значение, результат сравнения строк так же будет неопределенным, иначе результат сравнения этой пары определит результат всего сравнения. Например, `ROW(1,2,NULL) < ROW(1,3,0)` истина, а не неопределенное значение, поскольку в этом случае последняя третья пара не рассматривается.

Примечание. До версии PostgreSQL 8.2, операторы <, <=, > и >= не соответствовали спецификации SQL. Сравнение типа `ROW(a,b) < ROW(c,d)` осуществлялось как `a <`

c AND b < d, тогда как должно быть a < c OR (a = c AND b < d).

row_constructor IS DISTINCT FROM row_constructor

Эта конструкция похожа на использование оператора <> для построчного сравнения, но не поддерживает неопределенных входных значений. Вместо этого, неопределенное значение считается неравным (отличным от) любому определенному, а два неопределенных значения считаются равными (не различными). Результатом этой конструкции может быть только истина или ложь, и никогда неопределенное значение.

row_constructor IS NOT DISTINCT FROM row_constructor

Эта конструкция похожа на использование оператора = для построчного сравнения, но не поддерживает неопределенных входных значений. Вместо этого, неопределенное значение считается неравным (отличным от) любому определенному, а два неопределенных значения считаются равными (не различными). Результатом этой конструкции может быть только истина или ложь, и никогда неопределенное значение.

6.23.6. Сравнение составных типов

Примечание. Сравнение составных типов доступно только в СУБД версии 9.6.

record operator record

Спецификация SQL требует, что бы результатом строкового сравнения было неопределенное значение, если результат зависит от сравнения двух неопределенных значений или одного неопределенного с любым другим определенным. PostgreSQL обеспечивает это только при сравнения строк, построенных с помощью строковых конструкторов, или при результата строкового конструктора с результатом подзапроса (как в 6.22). В других контекстах, когда сравниваются два составных значения, поля, содержащие неопределенное значение, считаются равными, и неопределенное значение считается больше любого определенного, что является необходимым для корректной сортировки и индексирования составных значений.

Сравнение составных типов реализовано с помощью сравнения строк. При сравнении разрешается оператор =, <>, <, <=, > или >=, или другие, имеющие семантику, подобную семантике одного из перечисленных операторов. (Для определенности, оператор может быть оператором сравнения строк, если принадлежит операторному классу B-Tree, или является негатором для = операторного класса B-Tree). Поведение по умолчанию из указанных операторов это же, как для IS [NOT] DISTINCT FROM для конструкторов строк (см. 6.23.5).

Для поддержки соответствия строк, которые не включают элементы операторного класса B-Tree по умолчанию, следующие операторы сравнения определены для составного типа: *=, *<> *<, *<=, *> и *>=. Эти операторы сравнения сравнивают внутреннее бинарное представление двух строк. Две строки могут иметь различное бинарное представление,

даже если результат сравнения двух строк на равенство равен TRUE. Порядок строк в этих операторах является детерминированным. Эти операторы используются для внутренних материализованных представлений и могут быть полезными для других специальных целей, таких как репликация, но не предназначены для написания запросов.

6.24. Функции, возвращающие набор строк

В настоящем пункте описываются функции, могущих возвращать более одной строки. В настоящее время в их состав входят только функции генерации серий, приведенные в таблицах 80 и 81. См. 4.2 для определения способов объединять результаты функций, возвращающих набор строк.

Таблица 80 – Функции генерации серий

Функция	Тип аргумента, тип результата, описание	
generate_series(start, stop)	Тип аргумента	int или bigint
	Тип результата	Набор int или набор bigint (по типу аргумента)
	Описание	Генерирует последовательность от указанного начального значения до конечного с шагом 1
generate_series(start, stop, step)	Тип аргумента	int или bigint
	Тип результата	Набор int или набор bigint (по типу аргумента)
	Описание	Генерирует последовательность от указанного начального значения до конечного с заданным шагом
generate_series(start, stop, step interval)	Тип аргумента	timestamp или timestamp with time zone
	Тип результата	Набор timestamp или набор timestamp with time zone (по типу аргумента)
	Описание	Генерирует последовательность от указанного начального значения до конечного с заданным шагом

В случае задания положительного шага, если начальное значение превышает конечное, не возвращается ни одной строки. Обратно, при отрицательном шаге ни возвращается ни одной строки, если начальное значение уже меньше конечного. Так же не возвращаются строки для аргументов с неопределенным значением. Задание нулевого шага считается ошибкой.

Примеры:

1.
```
select * from generate_series(2,4);
 generate_series
-----------------
               2
               3
               4
(3 rows)
```

2.
```
select * from generate_series(5,1,-2);
```

```
 generate_series
-----------------
               5
               3
               1
(3 rows)
```

3. `select * from generate_series(4,3);`

```
 generate_series
-----------------
(0 rows)
```

4. В этом примере использован целочисленный оператор сложения для даты

```
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)
```

5. `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 81 – Генерирующие индекс массива функции

Функция	Тип результата, описание	
generate_subscripts(array anyarray, dim int)	Тип результата	набор int
	Описание	Генерирует последовательность номеров элементов указанного измерения массива
generate_subscripts(array anyarray, dim int, reverse boolean)	Тип результата	набор int
	Описание	Генерирует последовательность номеров элементов указанного измерения массива. В случае задания reverse = TRUE, последовательность возвращается в обратном порядке.

Функция generate_subscripts очень удобна для получения правильных порядковых номеров элементов указанного измерения массива. Не возвращается результата в случае отсутствия указанного измерения в массиве, или для переменной типа массив, имеющей неопределенное значение (но для элементов, содержащих неопределенное значение, номер возвращается).

Примеры:

1. Основное использование

```
select generate_subscripts('{NULL,1,NULL,2}'::int[], 1) as s;
 s
---
 1
 2
 3
 4
(4 rows)
```

2. Представляет массив с измерением и значениями, требующими выборки подзапросом

```
select * from arrays;
         a
--------------------
 {-1,-2}
 {100,200,300}
(2 rows)
```

3. Выборка a как массива, s как индекса массива, a[s] как значения

```
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)
```

4. Разбор двумерного массива. Создающая или заменяющая функция `unnest2(anyarray)` возвращает набор элементов как `setof anyelement`

```
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)
```

Примечание. Когда условие `FROM` функции заканчивается `WITH ORDINALITY`, значение столбца `bigint` добавляется к результату, которое инициализируется с 1 и увеличивается на 1 для каждой строки вывода функции. Это особенно полезно в случае вывода функций, возвращающих несколько значений, таких как `unnest()`.

```
--  функция, возвращающая набор строк с WITH ORDINALITY
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_clog         | 12
 pg_snapshots    | 13
 pg_multixact    | 14
 PG_VERSION      | 15
 pg_xlog         | 16
 pg_hba.conf     | 17
 pg_stat_tmp     | 18
 pg_subtrans     | 19
(19 rows)
```

6.25. Функции получения системной информации

В таблице 82 перечислены функции, которые отображают информацию о системе и текущей сессии. В дополнение к этим функциям существует ряд статистических функций, предоставляющих системную информацию.

Таблица 82 – Функции отображения информации о сессии

Функция	Тип результата	Описание
current_catalog	name	Имя текущей базы данных (в стандарте SQL - «catalog»)
current_database	name	Имя текущей базы данных
current_query	text	Текст текущего исполняемого запроса (может содержать более одной команды)
current_schema[()]	name	Имя текущей схемы
current_schema(boolean)	name[]	Имена схем в пути поиска, включая неявные
current_user	name	Имя пользователя текущего исполняемого контекста
inet_client_addr()	inet	Адрес клиентского соединения
inet_client_port()	int	Порт клиентского соединения
inet_server_addr()	inet	Адрес серверного соединения
inet_server_port()	int	Порт серверного соединения
pg_backend_pid()	int	Идентификатор серверного процесса (Process ID) ассоциированного с текущей сессией
pg_conf_load_time()	timestamp with time zone	Время загрузки конфигурации
pg_is_other_temp_schema(oid)	boolean	Является ли схема временной схемой другой сессии
pg_listening_channels()	setof text	Названия каналов, «прослушиваемых» сессией
pg_my_temp_schema()	oid	OID временной схемы сессии, или 0 при отсутствии

Окончание таблицы 82

Функция	Тип результата	Описание
pg_postmaster_start_time()	timestamp with time zone	Время запуска сервера
pg_trigger_depth()	int	Текущий уровень вложенности триггеров PostgreSQL (0 если не вызваны, напрямую или косвенно, вне триггера)
session_user	name	Имя пользователя сессии
user	name	Аналог current_user
version()	text	Версия PostgreSQL

Примечание. Функции current_catalog, current_schema, current_user, session_user и user имеют специальный статус в SQL, и должны вызываться без использования круглых скобок (использование скобок возможно в PostgreSQL с current_schema, но не с другими).

session_user — это пользователь, под именем которого было выполнено соединение с базой данных. Суперпользователь может менять это значение с помощью команды SET SESSION AUTHORIZATION. current_user — это пользователь, относительно которого будут выполняться проверки доступа. Как правило, currrent_user совпадает с session_user, но это может быть изменено сменой роли с помощью SET ROLE. Так же они могут отличаться в процессе выполнения функций с уровнем доступа SECURITY DEFINER. Говоря языком Unix, session_user является <<real user>>, а current user — <<effective user>>.

current_schema возвращает имя схемы, стоящей первой в пути поиска (или значение NULL, если путь поиска неопределен). Эта схема является схемой по умолчанию для поиска и создания объектов, для которых схема не задана явным образом. current_schema(boolean) возвращает массив имен всех схем, занесенных в путь поиска. Если эта функция вызвана с параметром TRUE, то в массив будут включены неявно заданные схемы (например, pg_catalog).

Примечание. Путь поиска может быть изменен в процессе выполнения командой: SET search_path TO schema [, schema, ...]

pg_listening_channels возвращает набор имен каналов, «прослушиваемых» сессией (см. LISTEN).

inet_client_addr возвращает IP-адрес клиента, а inet_client_port его порт. inet_server_addr возвращает IP-адрес серверного конца соединения, а inet_server_port порт этого соединения. Обе функции возвращают NULL в случае установки соединения с помощью сокетов Unix-domain.

pg_my_temp_schema возвращает OID временной схемы текущей сессии, или 0 если таковой нет (если не создавались временные таблицы). pg_is_other_temp_schema

возвращает истину, если указанный OID является идентификатором временной схемы другой сессии. (Это может быть полезно, например, для исключения отображения чужих схем при просмотре каталога.)

pg_postmaster_start_time возвращает время, когда был запущен серверный процесс.

pg_conf_load_time возвращает время, с которое был загружен файл конфигурации. (Если в этот момент текущая сессия уже существовала, это будет время перечитывания конфигурации, и для разных сессий может отличаться. Другими словами это момент времени, в который серверный процесс перечитал свои файлы конфигурации.)

version возвращает сроку описания версии сервера PostgreSQL.

В таблице 83 перечислены функции, позволяющие программно определить права доступа пользователей к объектам базы данных (о правах доступа смотри 2.6).

Таблица 83 – Функции определения прав доступа

Функция	Тип результата	Описание
has_any_column_privilege(user, table, privilege)	boolean	Имеет ли пользователь указанное право доступа к какому-нибудь из столбцов таблицы
has_any_column_privilege(table, privilege)	boolean	Имеет ли текущий пользователь указанное право доступа к какому-нибудь из столбцов таблицы
has_column_privilege(user, table, column, privilege)	boolean	Имеет ли пользователь указанное право доступа к столбцу таблицы
has_column_privilege(table, column, privilege)	boolean	Имеет ли текущий пользователь указанное право доступа к столбцу таблицы
has_database_privilege(user, database, privilege)	boolean	Имеет ли пользователь указанное право доступа к заданной базе данных
has_database_privilege(database, privilege)	boolean	Имеет ли текущий пользователь указанное право доступа к заданной базе данных
has_foreign_data_wrapper_- privilege(user, fdw, privilege)	boolean	Имеет ли пользователь указанное право доступа к обертке внешних данных
has_foreign_data_wrapper_- privilege(fdw, privilege)	boolean	Имеет ли текущий пользователь указанное право доступа к обертке внешних данных
has_function_privilege(user, function, privilege)	boolean	Имеет ли пользователь указанное право доступа к заданной функции
has_function_privilege(function, privilege)	boolean	Имеет ли текущий пользователь указанное право доступа к заданной функции
has_language_privilege(user, language, privilege)	boolean	Имеет ли пользователь указанное право доступа к заданному языку программирования
has_language_privilege(language, privilege)	boolean	Имеет ли текущий пользователь указанное право доступа к заданному языку программирования
has_schema_privilege(user, schema, privilege)	boolean	Имеет ли пользователь указанное право доступа к заданной схеме
has_schema_privilege(schema, privilege)	boolean	Имеет ли текущий пользователь указанное право доступа к заданной схеме

Окончание таблицы 83

Функция	Тип результата	Описание
has_sequence_privilege(user, sequence, privilege)	boolean	Имеет ли пользователь указанное право доступа к последовательности
has_sequence_privilege(sequence, privilege)	boolean	Имеет ли текущий пользователь указанное право доступа к последовательности
has_server_privilege(user, server, privilege)	boolean	Имеет ли пользователь указанное право доступа к внешнему серверу
has_server_privilege(server, privilege)	boolean	Имеет ли текущий пользователь указанное право доступа к внешнему серверу
has_table_privilege(user, table, privilege)	boolean	Имеет ли пользователь указанное право доступа к заданной таблице
has_table_privilege(table, privilege)	boolean	Имеет ли текущий пользователь указанное право доступа к заданной таблице
has_tablespace_privilege(user, tablespace, privilege)	boolean	Имеет ли пользователь указанное право доступа к области хранения данных
has_tablespace_privilege(tablespace, privilege)	boolean	Имеет ли текущий пользователь указанное право доступа к области хранения данных
pg_has_role(user, role, privilege)	boolean	Имеет ли пользователь указанное право доступа к роли
pg_has_role(role, privilege)	boolean	Имеет ли текущий пользователь указанное право доступа к роли

Функция has_table_privilege проверяет возможность доступа пользователя к таблице. Пользователь может быть задан по имени, либо по системному идентификатору (pg_authid.oid), public задает псевдо-роль PUBLIC, если аргумент опущен подразумевается <current_user>. Таблица может быть задана именем или своим идентификатором объекта (oid). (Таким образом, в зависимости от типов и количества аргументов существует шесть вариантов has_table_privilege.) Если таблица задается именем, то можно использовать и полное, и сокращенное имя таблицы. Интересующее право доступа задается в виде текстового выражения, которое должно вычисляться в одно из значений — SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES или TRIGGER. Дополнительно к праву доступа может быть добавлено WITH GRANT OPTION для проверки наличия опции делегирования. Могут быть указаны несколько прав доступа через запятую, в этом случае положительный результат будет при наличии любого из прав доступа. (При этом регистр получившейся текстовой строки не существенен.)

Пример

```
SELECT has_table_privilege('myschema.mytable', 'select');
SELECT has_table_privilege('joe', 'mytable', 'INSERT, SELECT WITH GRANT OPTION');
```

Функция has_sequence_privilege проверяет возможность доступа пользователя к последовательности. Все ее параметры аналогичны параметрам функции has_table_privileges. Задаваемое право доступа должно быть комбинацией из USAGE,

SELECT и UPDATE.

Функция has_any_column_privilege проверяет возможность доступа пользователя к какому-нибудь из столбцов таблицы. Все ее параметры аналогичны параметрам функции has_table_privileges. Задаваемое право доступа должно быть комбинацией из SELECT, INSERT, UPDATE и REFERENCES. Наличие любой из этих привилегий на уровне таблицы неявно применяется ко всем ее столбцам, так что has_any_column_privilege всегда возвращает истину, если ее возвращает has_table_privilege при тех же аргументах. Но так же истина возвращается при наличии требуемого права доступа к хотя бы одному из столбцов.

Функция has_column_privilege проверяет возможность доступа пользователя к столбцу таблицы. Все ее параметры аналогичны параметрам функции has_table_privileges, столбец может быть задан как именем, так и порядковым номером. Задаваемое право доступа должно быть комбинацией из SELECT, INSERT, UPDATE и REFERENCES. Наличие любой из этих привилегий на уровне таблицы неявно применяется ко всем ее столбцам.

Функция has_database_privilege проверяет возможность доступа пользователя к указанной базе данных. Все ее параметры аналогичны параметрам функции has_table_privileges. Задаваемое право доступа должно быть одно из — CREATE, CONNECT, TEMPORARY или TEMP (которое эквивалентно TEMPORARY).

Функция has_function_privilege проверяет возможность доступа пользователя к указанной функции. Параметры аналогичны параметрам функции has_table_privileges. При задании функции ее текстовым именем вместо oid, допустимо использование аргумента как типа regprocedure (смотри 5.18). Задаваемое право доступа должно быть EXECUTE.

Пример

```
SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');
```

Функция has_foreign_data_wrapper_privilege проверяет возможность доступа пользователя к указанной обертке внешних данных. Параметры аналогичны параметрам функции has_table_privileges. Задаваемое право доступа должно быть USAGE.

Функция has_language_privilege проверяет право пользователя на использование указанного процедурного языка. Параметры аналогичны параметрам функции has_table_privileges. Задаваемое право доступа должно быть USAGE.

Функция has_schema_privilege проверяет возможность доступа пользователя к указанной схеме. Параметры аналогичны параметрам функции has_table_privileges. Задаваемое право доступа должно быть одно из CREATE или USAGE.

Функция has_foreign_server_privilege проверяет возможность доступа пользователя к указанному внешнему серверу. Параметры аналогичны параметрам функции has_table_privileges. Задаваемое право доступа должно быть USAGE.

Функция has_tablespace_privilege проверяет возможность доступа пользователя к указанной области хранения данных. Параметры аналогичны параметрам функции has_table_privileges. Задаваемое право доступа должно CREATE.

Функция pg_has_role проверяет возможность доступа пользователя к указанной роли. Параметры аналогичны параметрам функции has_table_privileges. Задаваемое право доступа должно быть одно из MEMBER или USAGE. MEMBER подразумевает явную или неявную принадлежность пользователя роли (т. е. право на ее установку с помощью SET ROLE). USAGE означает доступность прав указанной роли без использования SET ROLE.

В таблице 84 перечислены функции, которые позволяют определить «видимость» объекта при текущем пути поиска. Объект считается «видимым», если схема, которая его содержит, перечислена в пути поиска, и в более ранних схемах нет объектов с тем же именем и типом.

Пример

Следующий запрос выведет список всех «видимых» таблиц:

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

Таблица 84 – Функции определения «видимости» объектов

Функция	Тип результата	Описание
pg_collation_is_visible(collation_oid)	boolean	Проверяет «видимость» способа сортировки
pg_conversion_is_visible(conversion_oid)	boolean	Проверяет «видимость» преобразования кодировки
pg_function_is_visible(function_oid)	boolean	Проверяет «видимость» функций
pg_operator_is_visible(operator_oid)	boolean	Проверяет «видимость» операторов
pg_opclass_is_visible(opclass_oid)	boolean	Проверяет «видимость» классов операторов
pg_opfamily_is_visible(opfamily_oid)	boolean	Проверяет «видимость» семейства операторов
pg_table_is_visible(table_oid)	boolean	Проверяет «видимость» таблиц
pg_ts_config_is_visible(config_oid)	boolean	Проверяет «видимость» конфигурации полнотекстового поиска
pg_ts_dict_is_visible(dict_oid)	boolean	Проверяет «видимость» словаря полнотекстового поиска
pg_ts_parser_is_visible(parser_oid)	boolean	Проверяет «видимость» синтаксического анализатора полнотекстового поиска
pg_ts_template_is_visible(template_oid)	boolean	Проверяет «видимость» шаблона полнотекстового поиска

Окончание таблицы 84

Функция	Тип результата	Описание
pg_type_is_visible(type_oid)	boolean	Проверяет «видимость» типов и доменов

Каждая из этих функций позволяет проверить видимость отдельного типа объектов в базе данных. Функция pg_table_is_visible выполняет проверку «видимости» таблиц, видов или любых других объектов, занесенных в таблицу pg_class. Функции pg_type_is_visible может быть использована и для доменов. Для функций и операторов объекты проверяются по именам вместе с типами аргументов. Для классов операторов проверяются и имена, и соответствующие методы доступа к индексу.

Все эти функции требуют использования идентификаторов объектов для указания проверяемых объектов. При необходимости проверить объект по имени, следует использовать соответствующие псевдонимы типа oid (regclass, regtype, regprocedure, regoperator, regconfig, и regdictionary).

Пример

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

Необходимо заметить, что нет смысла проверять этим способом видимость объектов без указания полного имени.

В таблице 85 перечислены функции, которые позволяют просмотреть информацию в системном каталоге.

Таблица 85 – Функции доступа к системному каталогу

Функция	Тип результата	Описание
format_type(type_oid, typemod)	text	Получение SQL имени для указанного типа
pg_describe_object(catalog_id, object_id, object_sub_id)	text	Получение описания объекта БД
pg_identify_object(catalog_id, object_id, object_sub_id)	type text, schema text, name text, identity text	Получение уникального имени объекта БД Примечание. Доступно в версии PostgreSQL 9.6.
pg_get_constraintdef(constraint_oid)	text	Реконструкция команды для создания ограничения
pg_get_constraintdef(constraint_oid, pretty_- bool)	text	Реконструкция команды для создания ограничения
pg_get_expr(pg_node_tree, relation_oid)	text	Реконструкция внутренней формы выражения, считая, что все параметры относятся к указанной таблице
pg_get_expr(pg_node_tree, relation_oid, pretty_bool)	text	Реконструкция внутренней формы выражения, считая, что все параметры относятся к указанной таблице

Продолжение таблицы 85

Функция	Тип результата	Описание
pg_get_functiondef(func_oid)	text	Получение определения функции
pg_get_function_arguments(func_oid)	text	Получение списка аргументов в функции (со значениями по умолчанию)
pg_get_function_identity_- arguments(func_oid)	text	Получение списка аргументов для идентификации функции (без значений по умолчанию)
pg_get_function_result(func_oid)	text	Получение конструкции RETURNS функции
pg_get_indexdef(index_oid)	text	Реконструкция команды CREATE INDEX для создания индекса
pg_get_indexdef(index_oid, column_no, pretty_bool)	text	Реконструкция команды CREATE INDEX для создания индекса, или определение для одной колонки если column_no не равно нулю
pg_get_keywords()	setof record	Получение списка ключевых слов SQL с их категориями
pg_get_ruledef(rule_oid)	text	Реконструкция команды CREATE RULE для создания правила
pg_get_ruledef(rule_oid, pretty_bool)	text	Реконструкция команды CREATE RULE для создания правила
pg_get_serial_sequence(table_name, column_name)	text	Получение имени последовательности, используемой столбцом тип serial или bigserial
pg_get_triggerdef(trigger_- oid)	text	Получение команды CREATE [CONSTRAINT] TRIGGER для триггера
pg_get_triggerdef(trigger_- oid, pretty_bool)	text	Получение команды CREATE [CONSTRAINT] TRIGGER для триггера
pg_get_userbyid(roleid))	name	Получение имени пользователя или роли по системному идентификатору
pg_get_viewdef(view_name)	text	Устаревшая реконструкция команды CREATE VIEW для создания вида или материализованного вида
pg_get_viewdef(view_name, pretty_bool)	text	Устаревшая реконструкция команды CREATE VIEW для создания вида или материализованного вида
pg_get_viewdef(view_oid)	text	Реконструкция команды CREATE VIEW для создания вида или материализованного вида
pg_get_viewdef(view_oid, pretty_bool)	text	Реконструкция команды CREATE VIEW для создания вида или материализованного вида
pg_get_viewdef(view_oid, wrap_column_int)	text	Реконструкция команды CREATE VIEW для создания вида или материализованного вида; строки с полями расширены до указанного количества, используется режим «pretty»
pg_options_to_table(reloptions)	setof record	Получение набора опций хранения в виде пар ключ/значение
pg_tablespace_databases(tablespace_oid)	setof oid	Получение списка идентификаторов (OID) баз данных, имеющих объекты в указанной области хранения данных
pg_tablespace_location(tablespace_oid)	text	Получение пути в файловой системе для указанной области хранения данных
pg_typeof(any)	regtype	Получение типа любого значения

Окончание таблицы 85

Функция	Тип результата	Описание
`collation for (any)`	`text`	Получение способа сортировки указанного аргумента
`to_regclass(rel_name)`	`regclass`	Получить OID отношения. Примечание. Только в версии СУБД 9.6
`to_regproc(func_name)`	`regprocedure`	Получить OID функции. Примечание. Только в версии СУБД 9.6
`to_regprocedure(func_name)`	`regprocedure`	Получить OID функции. Примечание. Только в версии СУБД 9.6
`to_regoper(operator_name)`	`regoper`	Получить OID оператора. Примечание. Только в версии СУБД 9.6
`to_regoperator(operator_name)`	`regoperator`	Получить OID оператора. Примечание. Только в версии СУБД 9.6
`to_regtype(type_name)`	`regtype`	Получить OID типа. Примечание. Только в версии СУБД 9.6

Функция `format_type` возвращает SQL имя заданного идентификатором типа данных возможно с указанием модификатора типа. Если модификатор неизвестен можно в качестве его указывать `NULL`.

Функция `pg_get_keywords` возвращает список распознаваемых сервером ключевых слов SQL. Столбец `word` содержит само слово. Столбец `catcode` содержит категорию: `U` для не зарезервированных, `C` для имен столбцов, `T` для типов или имен функций, или `R` для зарезервированных. Столбец `catdesc` содержит по возможности локализованное описание категории.

Функции `pg_get_constraintdef`, `pg_get_indexdef`, `pg_get_ruledef`, и `pg_get_triggerdef` реконструируют команды создания ограничений, индексов, правил и триггеров соответственно. Необходимо отметить, что результатом является не исходная, а реконструированная по внутреннему представлению команда. Функция `pg_get_expr` реконструирует внутреннее представление выражения типа значений по умолчанию для столбцов, и может быть использована для получение информации о системном каталоге. Функция `pg_get_viewdef` реконструирует определяющий вид запрос `SELECT`. Большинство из рассматриваемых функций представлено в двух вариантах, один из которых имеет более удобную форму представления результата. Хотя такое представление более удобно, но стандартный вариант более надежен в плане совместимости с будущими версиями PostgreSQL. Не рекомендуется использовать указанную форму в целях создания резервных копий. Передача `FALSE` в качестве аргумента `pretty`, дает тот же результат, что и

использование функции без этого аргумента.

Функция pg_get_functiondef возвращает полную форму команды CREATE OR REPLACE FUNCTION для создания функции. Функция pg_get_function_arguments возвращает список аргументов функции в виде, необходимом для использования внутри конструкции CREATE FUNCTION. Функция pg_get_function_result возвращает конструкцию RETURNS для функций. Функция pg_get_function_identity_arguments возвращает список аргументов, необходимых для идентификации функции, в форме применимой например совместно с ALTER FUNCTION. В этой форме не приводятся значения по умолчанию.

Функция pg_get_serial_sequence позволяет получить имя ассоциированной со столбцом последовательности, или NULL при отсутствии таковой. Первым аргументом является имя таблицы возможно с указанием схемы, вторым — имя столбца. Поскольку в первом аргументе может присутствовать указание схемы, он не рассматривается, как заключенный в двойные кавычки, и умолчанию приводится к нижнему регистру. Второй, так как содержит только имя столбца, рассматривается, как заключенный в двойные кавычки, и сохраняет свое регистровое представление. Возвращаемый результат форматируется для возможности дальнейшего применения в функциях, работающих с последовательностями (смотри 6.16). Полученная ассоциация может быть удалена или изменена командой ALTER SEQUENCE OWNED BY. Функция возможно должна была бы называться pg_get_owned_sequence; подобное название больше отражает что она в основном используется для определения последовательности, ассоциированной со столбцами типа serial или bigserial.

Функция pg_get_userbyid возвращает имя пользователя или роли по указанному OID.

Функция pg_options_to_table возвращает набор опций хранения в виде пар ключ/значение (option_name/option_value), которые располагаются в pg_class.reloptions или pg_attribute.attoptions.

Функция pg_tablespace_databases позволяет получить информации об области хранения данных. Она возвращает набор идентификаторов (OID) баз данных, чьи объекты хранятся в указанной области. Если функция возвратила набор строк, область хранения данных не пуста и не может быть удалена. Для получения списка объектов, расположенных в конкретной области хранения данных, необходимо подключится к полученным базам данных и запросить информацию из таблицы pg_class системного каталога.

Функция pg_describe_object возвращает текстовое описание объекта БД, заданного с помощью OID каталога, OID самого объекта и идентификатора подобъекта (возможно равного нулю). Это описание рассматривается как предназначенное для пользователя и

может быть локализовано в зависимости от конфигурации сервера. Это описание удобно для определения объекта, хранимого в системном каталоге pg_depend.

Функция pg_identify_object возвращает информацию, уникально идентифицирующую объект БД, заданный с помощью OID каталога, OID самого объекта и идентификатора подобъекта (возможно равного нулю). Эта информация не предназначена для пользователя и никогда не локализуется. type идентифицирует тип объекта БД, schema задает имя схемы, которой принадлежит объект, или NULL для типов объектов, не принадлежащим схемам, name задает имя объекта, при необходимости заключенное в кавычки, если оно может быть использовано для уникальной идентификации объекта, в противном случае — NULL, identity — полная информация в формате, зависящем от типа объекта, каждая из частей которого может содержать имя схемы и может быть при необходимости заключено в кавычки.

Функция pg_typeof возвращает идентификатор OID типа данных переданного значения, что может быть полезным для поиска ошибок или создания динамических SQL-запросов. Функция определена, как возвращающая результат типа regtype, являющего псевдонимом типа OID (смотри 5.18). Это дает возможность использовать стандартные операторы сравнения, и при этом иметь текстовое представления на экране.

Пример

```
SELECT pg_typeof(33);

 pg_typeof
-----------
 integer
(1 row)

SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);
 typlen
--------
      4
(1 row)
```

Выражение collation for возвращает способ сортировки переданного значения.

Пример

```
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)
```

Значение должно быть заключено в кавычки и содержать имя схемы. Если способ сортировки для аргумента не задан, возвращается NULL. Для не поддерживающих способов сортировки аргументов выдается ошибка.

Функции to_regclass, to_regproc, to_regprocedure, to_regoper, to_regoperator, и to_regtype преобразуют имена отношения, функции, оператора и типа в типы regclass, regproc, regprocedure, regoper, regoperator и regtype соответственно. Эти функции отличаются от операторов привидения типов тем, что они не принимают числовой OID, и что они возвращаются NULL, а не выводят ошибку, если имя не найдено (или, для to_regproc и to_regoper, если имя совпадает с несколькими объектами).

Функции, перечисленные в таблице 86, выводят комментарии, ранее созданные командой COMMENT. Если для объекта не было задано комментария, то будет выведено неопределенное (NULL) значение.

Таблица 86 – Функции просмотра комментариев

Функция	Тип результата	Описание
col_description(table_oid, column_number)	text	Получение комментария к столбцу таблицы
obj_description(object_oid, catalog_name)	text	Получение комментария к объекту базы данных
obj_description(object_oid)	text	Получение комментария к объекту базы данных (устаревшая)
shobj_description(object_oid, catalog_name)	text	Получение комментария к разделяемому объекту базы данных

Функция col_description возвращает комментарий к столбцу таблицы, задаваемому идентификатором (OID) таблицы и своим именем. Для этой цели нельзя использовать функцию obj_description, поскольку столбцы таблиц не имеют собственных уникальных идентификаторов.

Форма функции obj_description с двумя аргументами возвращает комментарий к объекту базы данных, определяемого его идентификатором и именем содержащей его таблицы системного каталога.

Пример

Функция obj_description возвратит комментарий для таблицы с идентификатором 123456.

```
SELECT obj_description(123456, 'pg_class')
```

Функция obj_description с одним параметром считается устаревшей и не гарантирует возврат правильного комментария (из-за возможной в будущем генерации идентификаторов объектов (строк) для каждой таблицы отдельно).

Функция shobj_description используется аналогично obj_description, но для получения комментариев к разделяемым объектам. Некоторые системные таблицы являются глобальными для баз данных, входящих в один кластер, и их описания так же являются глобальными.

Функции, приведенные в таблице 87, позволяют пользователю получать внутреннюю информацию сервера о транзакциях. Основным использование этих функций является определение того, какие транзакций были завершены между двумя снимками состояния (snapshot — снимок состояния базы данных на начало транзакции) базы данных.

Таблица 87 – Идентификаторы транзакций и снимки

Функция	Тип результата	Описание
txid_current()	bigint	Получение идентификатора текущей транзакции
txid_current_snapshot()	txid_snapshot	Получение идентификатора текущего снимка состояния
txid_snapshot_xmin(txid_snapshot)	bigint	Получение минимального xmin идентификатора транзакции в снимке состояния
txid_snapshot_xmax(txid_snapshot)	bigint	Получение максимального xmax идентификатора транзакции в снимке состояния
txid_snapshot_xip(txid_snapshot)	setof bigint	Получения списка идентификаторов активных транзакций в снимке состояния
txid_visible_in_snapshot(bigint, txid_snapshot)	boolean	Идентификатор транзакции виден в снимке состояния?

Внутренне представление типа идентификатора транзакции ID type (xid), представляет собой 32-целое, переполняющееся через 4 миллиарда транзакций. Однако, функции возвращают результат в 64-битном формате, расширенном счетчиком <<epoch>>, что не дает идентификатору переполнятся в течении всего срока работы системы. Тип данных txid_snapshot, используемых для этих функций, содержит информацию об активности и видимости транзакций на конкретный момент времени. В таблице 88 описаны компоненты этого типа.

Таблица 88 – Компоненты снимка состояния

Компонент	Описание
xmin	Идентификатор ID (txid) самой раннее активная транзакции. Все более ранние транзакции должны быть завершены и видимы, или откачены, и соответственно не видимы
xmax	Наименьший еще не ассоциированный идентификатор. Транзакции с большими идентификаторами еще не начинались и потому не видимы
xip_list	Список активных транзакций в снимке состояния. Список содержит только те транзакции, чьи идентификаторы располагаются между xmin и xmax. Транзакции с идентификаторами попадающими в указанные рамки, но не содержащиеся в списке, являются завершенными и видимыми, или откаченными в зависимости от их статуса завершения. Список так же не содержит идентификаторов вложенных транзакций

Текстовое представление типа txid_snapshot выглядит следующим образом xmin:xmax:xip_list.

Пример

10:20:10,14,15 содержит xmin=10, xmax=20, xip_list=10, 14, 15.

6.26. Функции системного администрирования

В разделе приведены функции управления и мониторинга PostgreSQL.

6.26.1. Функции изменения параметров конфигурации

В таблице 89 приведены функции для запроса и изменения параметров конфигурации, доступных для изменения во время работы сервера.

Таблица 89 – Функции работы с параметрами конфигурации

Функция	Тип результата	Описание
current_setting(setting_name)	text	Получение текущего значения
set_config(setting_name, new_value, is_local)	text	Установка и получение нового значения

Функция current_setting возвращает текущее значение заданного именем setting_name параметра. Функция соответствует команде SQL SHOW.

Пример

```
SELECT current_setting('datestyle');

 current_setting
-----------------
 ISO, MDY
```

```
(1 row)
```

Функция set_config устанавливает значение параметра setting_name в новое new_value. Если установлен аргумент is_local, новое значение применяется только к текущей транзакции. Для установки нового значения в пределах сессии необходимо установить значение аргумента is_local в FALSE. Функция соответствует команде SQL SET.

Пример

```
SELECT set_config('log_statement_stats', 'off', false);

 set_config
------------
 off
(1 row)
```

6.26.2. Функции передачи сигналов

Приведенные в таблице 90 функции позволяют посылать сигналы другим процессам сервера. Использование указанных функции возможно только суперпользователем.

Таблица 90 – Функции посылки сигналов серверу

Функция	Тип результата	Описание
pg_cancel_backend(pid int)	boolean	Прерывание выполнения запроса указанным серверным процессом
pg_reload_conf()	boolean	Перезагрузка сервером конфигурационных файлов
pg_rotate_logfile()	boolean	Ротация журнала сервера
pg_terminate_backend(pid int)	boolean	Завершение указанного серверного процесса

Все функции возвращают истину при успешном выполнении, в противном случае возвращается ложь.

Функции pg_cancel_backend и pg_terminate_backend посылают сигналы (SIGINT и SIGTERM соответственно) указанному идентификатором ID серверному процессу. Идентификатор ID процесса может быть определен по столбцу procpid вида pg_stat_activity, или просмотром списка процессов postgres на сервере с помощью команды ps.

Функция pg_reload_conf посылает сигнал SIGHUP серверу, что вызывает перезагрузку серверными процессами файлов конфигурации.

Функция pg_rotate_logfile сигнализирует менеджеру журнализации сервера о

необходимости немедленно начать новый файл журнала. Команда выполняется только если запущен встроенный менеджер журналов, в потивном случае подобного процесса не существует.

6.26.3. Функции управления резервным копированием

Функции, показанные в таблице 91, помогают при осуществлении on-line (в процессе работы сервера) резервного копирования. Эти функции не могут быть исполнены во время восстановления (за исключением `pg_is_in_backup`, `pg_backup_start_time` и `pg_xlog_location_diff`).

Таблица 91 – Функции управления резервным копированием

Функция	Тип результата	Описание
`pg_create_restore_point(name text)`	`text`	Создание именованной точки восстановления (только для суперпользователя)
`pg_current_xlog_insert_location()`	`text`	Получение текущей позиции вставки в журнале транзакций
`pg_current_xlog_location()`	`text`	Получение текущей позиции записи в журнале транзакций
`pg_start_backup(label text [, fast boolean ])`	`text`	Подготовка к выполнению on-line резервного копирования (только для суперпользователя или роли репликации)
`pg_stop_backup()`	`text`	Завершение выполнения on-line резервного копирования (только для суперпользователя или роли репликации)
`pg_is_in_backup()`	`bool`	Возвращает истину, если находится в процессе on-line резервного копирования
`pg_backup_start_time()`	`timestamp with time zone`	Возвращает время начала on-line резервного копирования
`pg_switch_xlog()`	`text`	Принудительное начало нового журнала транзакций (только для суперпользователя)
`pg_xlogfile_name(location pg_lsn)`	`text`	Конвертирование текстового представления расположения журнала транзакций в имя файла
`pg_xlogfile_name_offset(location pg_lsn)`	`text, integer`	Конвертирование текстового представления расположения журнала транзакций в имя файла и смещении в нем

Окончание таблицы 91

Функция	Тип результата	Описание
pg_xlog_location_diff(location pg_lsn, location pg_lsn)	numeric	Вычисление разницы смещений между двумя транзакциями в журнале транзакций

Функция pg_start_backup принимает один аргумент, являющийся заданной пользователем меткой для резервной копии. Обычно используется имя, под которым резервная копия будет сохранена на диске. Функции сохраняет файл с указанной меткой в директорию кластера баз данных и возвращает тестовое представление позиции начала резервной копии в файле журнала транзакций. Пользователю не следует обращать внимание на это значение, оно необходимо только для дальнейшего использования.

Пример

```
postgres=# select pg_start_backup('label_goes_here');
 pg_start_backup
-----------------
 0/D4445B8
(1 row)
```

Необязательный второй параметр задает режим неотложного исполнения pg_start_backup. Это может вызвать всплеск операций ввода/вывода, что может затормозить исполнение параллельно выполняющихся запросов.

Функция pg_stop_backup удаляет созданный функцией pg_start_backup файл метки, и создает файл историй выполнения резервных копий в архивной области журнала транзакций. Файл истории содержит метку резервной копии, позиции начала и завершения резервного копирования в журнале транзакции, и время начала и завершения резервного копирования. Результатом функции является позиция завершения резервного копирования в журнале транзакций (которая также не представляет особого интереса). После фиксации позиции завершения, текущее место вставки журнала транзакций автоматически позиционируется в новый файл журнала транзакций, что бы старый журнал транзакций мог быть архивирован для завершения выполнения резервного копирования.

Функция pg_switch_xlog начинает новый файл транзакций, обеспечивая возможность архивирования текущего файла журнала транзакции (в случае использования непрерывного архивирования). Результатом является позиция в новом журнале транзакций. Если с последнего вызова этой функции не было никакой активности. Новый журнал транзакций не создается, и возвращается позиция в предыдущем журнале.

Функция pg_create_restore_point создает именованную запись в журнале тран-

закций, которая может быть использована как цель восстановления, и возвращает позицию в журнале транзакций. Данное имя может быть впоследствии использовано в recovery_target_name для указания точки восстановления. Не рекомендуется создавать несколько точек восстановления с одним именем, так как процесс восстановления будет остановлен по достижении первой точки с таким именем.

Функция pg_current_xlog_location отображает текущую позицию записи в журнале транзакций в формате, используемом предыдущими описанными функциями. Аналогично, функция pg_current_xlog_insert_location отображает текущую позицию вставки в журнале транзакции. Позиция вставки является логическим концом журнала транзакции в каждый конкретный момент времени, тогда как позиция записи указывает на конец последней произведенной из внутреннего буфера сервера записи. Позиция записи — конечная позиция, которая может быть использована вне сервера, и обычно используется для архивирования частично-завершенных журналов транзакций. Позиция вставки сделана доступной в основном для отладки сервера. Обе операции обеспечивают только чтение указанных показателей и не требуют для выполнения прав суперпользователя.

Возможно использование функции pg_xlogfile_name_offset для получения соответствующего имени файла журнала транзакций и позиции в нем из результата ранее описанных функций.

Пример

```
postgres=# SELECT * FROM pg_xlogfile_name_offset(pg_stop_backup());
         file_name          | file_offset
----------------------------+-------------
 00000001000000000000000D |     4039624
(1 row)
```

Аналогично, функция pg_xlogfile_name извлекает только имя файла журнала транзакций. Когда запрашиваемая позиция попадает точно на границу файлов журналов, обе функции возвращают имя предыдущего файла журнала транзакций. Такое поведение обычно является желаемым для менеджера архивирования, так как последним доступным для архивирования файлом журнала транзакций как раз является предыдущий.

Функция pg_xlog_location_diff вычисляет разницу смещений между двумя транзакциями в журнале транзакций. Это может быть использовано совместно с pg_stat_replication или некоторыми функциями из таблицы 91 для получения времени запаздывания репликации.

Более детально об использовании этих функций см. раздел 24.3.

6.26.4. Функции управления восстановлением

Функции, показанные в таблице 92, предоставляют информацию о текущем статусе восстановления, и могут быть использованы как во время восстановления, так и в режиме нормального функционирования.

Таблица 92 – Функции статуса восстановления

Функция	Тип результата	Описание
`pg_is_in_recovery()`	`bool`	Возвращает истину, если находится в процессе восстановления
`pg_last_xlog_receive_location()`	`pg_lsn`	Возвращает последнюю позицию в журнале транзакций, полученную и синхронизированную потоковой репликацией. В ходе потоковой репликации это значение монотонно возрастает. После завершения восстановления значение остается неизменным, указывая на последнюю полученную и синхронизированную в ходе восстановления запись журнала. Если потоковая репликация выключена или не была запущена, возвращается `NULL`.
`pg_last_xlog_replay_location()`	`pg_lsn`	Возвращает последнюю позицию в журнале транзакций, выполненную в процессе восстановления. В ходе восстановления это значение монотонно возрастает. После завершения восстановления значение остается неизменным, указывая на последнюю выполненную в ходе восстановления запись журнала. Если сервер был запущен нормально без восстановления, возвращается `NULL`.
`pg_last_xact_replay_timestamp()`	`timestamp with time zone`	Возвращает временную метку последней транзакции, выполненной в процессе восстановления. Это время завершения или отката данной транзакции на основном сервере. Если во время восстановления транзакции не выполнялись, возвращается `NULL`, иначе в ходе восстановления это значение монотонно возрастает. После завершения восстановления значение остается неизменным, указывая временную метку последней выполненной в ходе восстановления транзакции. Если сервер был запущен нормально без восстановления, возвращается `NULL`.

Функции, показанные в таблице 93, управляют процессом восстановления и могут быть выполнены только во время восстановления.

Таблица 93 – Функции управления восстановлением

Функция	Тип результата	Описание
`pg_is_xlog_replay_paused()`	`bool`	Возвращает истину, если процессе восстановления приостановлен
`pg_xlog_replay_pause()`	`void`	Немедленная приостановка восстановления. Примечание. Вызывать данную функцию могут только суперпользователи в СУБД версии 9.6.
`pg_xlog_replay_resume()`	`void`	Перезапуск восстановления после приостановки. Примечание. Вызывать данную функцию могут только суперпользователи в СУБД версии 9.6.

Пока восстановление приостановлено никакие изменения в БД не применяются. В режиме горячего восстановления все новые запросы видят одно целостное состояние (snapshot) базы данных, и никакие конфликты не генерируются до продолжения восстановления.

Если потоковая репликация выключена, приостановленное состояние может продолжатся бесконечно без каких-либо проблем. В ходе потоковой репликации записи журнала транзакций продолжают поступать пока достаточно места на диске, что зависит от продолжительности паузы, скорости заполнения журнала и доступного места на диске.

6.26.5. Функции синхронизации снимков (snapshot)

PostgreSQL позволяет сессиям синхронизировать их снимки. Снимок (snapshot) определяет, какие данные доступны транзакции, использующей снимок. Синхронизация снимков необходима, когда две или более сессии должны видеть одно состояние базы данных. Если две сессии начинают свои транзакции независимо, всегда возможна ситуация, когда третья транзакция завершается между выполнением двух команд `START TRANSACTION`, так что одна сессия видит ее результат, а другая — нет.

Для решения этой проблемы, PostgreSQL позволяет транзакции экспортировать снимок, который она использует. До тех пор пока существует экспортирующая транзакция, другие транзакции могут импортировать ее снимок, и таким образом гарантируется видимость того же самого состояния базы данных. При этом любые изменения в базе данных, выполненные одной из этих транзакций, остаются невидимыми другими, как это

и должно быть для незавершенных транзакций. Так что транзакции синхронизированы по выполненным до них изменениям, но работают нормально с с изменениями, внесенными ими самими.

Снимки экспортируются функцией pg_export_snapshot, приведенной в таблице 94, а импортируются командой SET TRANSACTION.

Таблица 94 – Функции синхронизации снимков

Функция	Тип результата	Описание
pg_export_snapshot()	text	Сохраняет текущий снимок и возвращает его идентификатор

Функция pg_export_snapshot сохраняет текущий снимок и возвращает текстовый идентификатор снимка. Эта строка должна быть передана (вне базы данных) клиентам, желающим импортировать снимок. Снимок доступен для импорта только пока существует, экспортировавшая его транзакция. При необходимости транзакция может экспортировать более одного снимка, но это имеет смысл только для READ COMMITTED транзакций, поскольку при уровнях изоляции REPEATABLE READ и более высоких, транзакция использует в течении своего существования один снимок. Транзакция, экспортировавшая снимок, не может быть подготовлена командой PREPARE TRANSACTION.

Детальная информация по использованию экспортированных снимков приведена в описании команды SET TRANSACTION.

6.26.6. Функции для управления репликацией

Примечание. Данные функции используются только в СУБД версии 9.6.

Функции, представленные в таблице 95, используются для контролирования и взаимодействия репликации. См. 15.2.5 и 15.2.7 для получения информации о описанных ниже возможностях. Использовать данные функции разрешено только суперпользователям.

Многие из этих функций имеют эквивалентные команды в протоколе репликации.

Функции, описанные в 6.26.5, 6.26.4 и 6.26.3 также применимы для репликации.

Таблица 95 – SQL Функции управления репликацией

Функция	Тип результата	Описание
pg_create_physical_-replication_-slot(slot_name name)	(slot_name name, xlog_-position pg_lsn)	Создает новый слот физической репликации с именем slot_name. Потоковое изменения физического слота возможны только при использовании протокола репликации. Соответствует команде репликации протокола CREATE_REPLICATION_SLOT ... PHYSICAL.

pg_drop_replication_slot(slot_name name)	void	Удаляет слот физической или логической репликации slot_name. Аналогична команде протокола репликации DROP_REPLICATION_SLOT.
pg_create_logical_replication_slot(slot_name name, plugin name)	(slot_name name, xlog_position pg_lsn)	Создает новый логический (декодирования) слот репликации с именем slot_name, используя плагин plugin. Вызов этой функции соответствует команде протокола репликации CREATE_REPLICATION_SLOT ... LOGICAL.
pg_logical_slot_get_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[]	(location pg_lsn, xid xid, data text)	Вовращает изменения в слоте slot_name, начиная с точки, с которой эти изменения произошли. Если upto_lsn и upto_nchanges равны NULL, логическое декодирование продолжится до окончания WAL. Если upto_lsn не NULL, декодирование будет включать только транзации, завершенные до указанного LSN. Если upto_nchanges не равно NULL, декодирование остановится, когда количество выведенных строк путем декодирования превышает указанное значение. Стоит отметить, что фактическое количество возвращаемых строк может быть больше, так как этот предел проверяется только после добавления строки, полученные при декодировании каждой новой транзакции.
pg_logical_slot_peek_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])	(location text, xid xid, data text)	Аналогична по поведению функции pg_logical_slot_get_changes(), за исключеним того, что изменения не уничтожаются; то есть, они будут возвращены снова на последующих вызовах.
pg_logical_slot_get_binary_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])	(location pg_lsn, xid xid, data bytea)	Аналогична по поведению функции pg_logical_slot_get_changes(), за исключением того, что изменения возвращаются типа bytea.
pg_logical_slot_peek_binary_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])	(location pg_lsn, xid xid, data bytea)	Аналогична по поведению функции pg_logical_slot_get_changes(), за исключением того, что изменения возвращаются типа bytea и что изменения не расходуются; то есть, они будут возвращены снова на последующих вызовов.

6.26.7. Функции управления объектами БД

Функции, приведенные в таблице 96, рассчитывают реальное использования дискового пространства объектами базы данных.

Таблица 96 – Функции получения размеров объектов базы данных

Функция	Тип результата	Описание
pg_column_size(any)	int	Количество байт под хранение одного значения столбца (возможно сжатого)
pg_database_size(oid)	bigint	Место на диске, занимаемое базой данных (по идентификатору)
pg_database_size(name)	bigint	Место на диске, занимаемое базой данных (по имени)
pg_indexes_size(regclass)	bigint	Место на диске, занимаемое индексами указанной таблицы
pg_relation_size(relation regclass, fork text)	bigint	Место на диске, занимаемое указанной частью, основной частью или картой свободного места таблицы или индекса
pg_relation_size(relation regclass)	bigint	Сокращенное название для pg_relation_size(..., 'main')
pg_size_pretty(bigint)	text	Конвертирование размера в байта в более удобный для восприятия формат с указанием единицы измерения
pg_size_pretty(numeric)	text	Конвертирование размера в байта в более удобный для восприятия формат с указанием единицы измерения
pg_table_size(regclass)	bigint	Место на диске, занимаемое указанной таблицей без учета индексов (но включая TOAST расширения, основную часть, карту свободного места и карту видимости)
pg_tablespace_size(oid)	bigint	Место на диске, занимаемое указанной областью хранения данных (по идентификатору)
pg_tablespace_size(name)	bigint	Место на диске, занимаемое указанной областью хранения данных (по имени)
pg_total_relation_size(regclass)	bigint	Общее мест она диске занимаемое указанной таблицей, ее расширениями и индексами

Функция pg_column_size показывает сколько будет занимать одно значение столбца.

Функция pg_total_relation_size принимает в качестве аргумента идентификатор или имя таблицы, индекса или расширения таблицы, и возвращает размер в байтах всех данных, включая ассоциированные индексы и расширения. Эта функция эквивалентна pg_table_size + pg_indexes_size.

Функция pg_table_size принимает в качестве аргумента идентификатор или имя таблицы и возвращает место на диске, занимаемое указанной таблицей без учета индексов (но включая TOAST расширения, основную часть, карту свободного места и карту

видимости).

Функция pg_indexes_size принимает в качестве аргумента идентификатор или имя таблицы и возвращает место на диске, занимаемое индексами указанной таблицы.

Функции pg_database_size и pg_tablespace_size принимают в качестве аргумента идентификатор или имя базы данных или области хранения данных, и возвращает общее занимаемое ими на диске место.

Функция pg_relation_size принимает OID или имя таблицы, индекса или toast-таблицы и возвращает размер на диске в байтах одного отношения. (Обратите внимание, что в большинстве случаев удобнее использовать более высокого уровня функции pg_total_relation_size или pg_table_size, которые возвращают суммарный размер отношения.) Вызванная с одним аргументом, она возвращает размер основной памяти отношения. Вызванная со вторым аргументом, она выдает детальную информацию о хранении отношения:

- 'main' – возвращает размер головной записи отношения;
- 'fsm' – возвращает размер Free Space Map, связанного с отношением;
- 'vm' – возвращает размер Visibility Map;
- 'init' – возвращает размер инициализации отношения, если есть, связанной с отношением.

Примечание. Вывод детальной информации о структуре памяти отношения в функции pg_relation_size() возможен только в СУБД версии 9.6.

Функция pg_size_pretty может быть использована для форматирования результата рассматриваемых функций в более удобный для восприятия вид с использованием сокращений единиц измерений типа kB, MB, GB или TB.

Описанные выше функции работы с таблицами и индексами принимают аргумент типа regclass, который представляет OID таблицы или индекса из системного каталога pg_class. Нет необходимости определять OID вручную, поскольку функция ввода типа regclass позволяет указать имя таблицы в виде строкового литерала, заключенного в одинарные кавычки. При этом для совместимости с обычными SQL именами, имена таблиц приводятся к нижнему регистру, если они не были заключены в двойные кавычки.

Если переданный в выше описанные функции в качестве аргумента OID не относится к существующему объекту, возвращается NULL.

Функции, приведенные в таблице 97, позволяют определить элементы файловой системы, ассоциированные с объектами БД.

Таблица 97 – Функции получения расположения объектов базы данных на диске

Функция	Тип результата	Описание
pg_relation_filenode(relation regclass)	oid	Filenode указанного отношения
pg_relation_filepath(relation regclass)	text	Путь к файлу указанного отношения
pg_filenode_relation(tablespace oid, filenode oid)	regclass	Найти отношение, находящейся в указанном табличном пространстве и указанным файловым дескриптором

Функция pg_relation_filenode принимает в качестве аргумента идентификатор или имя таблицы, индекса, расширения TOAST и возвращает текущий идентификатор файла "filenode", ассоциированный с указанным объектом БД. filenode является базовым компонентом имени файла, используемого для хранения содержимого объекта БД. Для большинства таблиц результатом является значение pg_class.relfilenode, но для некоторых системных каталогов значение relfilenode равно нулю, и для получения корректного значения должна быть использована эта функция. Функция возвращает NULL, если переданное в качестве аргумента отношение не хранится в файловой системе, это справедливо например для представлений.

Функция pg_relation_filepath схожа с pg_relation_filenode, но возвращает полный путь (относительно каталога кластера PGDATA) к файлу отношения.

Функция pg_filenode_relation является обратной для функции pg_relation_filenode. По переданным OID «табличного пространства» и «файловому дескриптору» функция возвращает OID отношения, связанного с ними. Для таблицы в табличном пространстве по умолчанию базы данных, табличное пространство может быть определено как 0.

6.26.8. Функции доступа к файлам

Приведенные в таблице 98 функции обеспечивают доступ к файлам, расположенным на сервере баз данных. Доступ разрешен только к файлам, расположенным в пределах папки кластера баз данных и папки log_directory. Рекомендуется использовать относительные пути в пределах папки кластера баз данных и папки log_directory. Использование указанных функции возможно только суперпользователем.

Таблица 98 – Функции доступа к файлам

Функция	Тип результата	Описание
pg_ls_dir(dirname text)	setof text	Список содержимого указанной папки

Окончание таблицы 98

Функция	Тип результата	Описание
`pg_read_file(filename text, offset bigint, length bigint)`	`text`	Чтение текстового файла
`pg_read_binary_file(filename text [, offset bigint, length bigint])`	`bytea`	Чтение бинарного файла
`pg_stat_file(filename text)`	`record`	Получение информации о файле

Функция `pg_ls_dir` возвращает список всех элементов в папке, исключая специальные элементы «`.`» и «`..`».

Функция `pg_read_file` возвращает часть текстового файла, начиная от указанного смещения `offset`, длинной `length` байт (меньше, если достигнут конец файла). При отрицательном указании смещения, оно отсчитывается от конца файла. Если смещение и длина опущены, возвращается весь файл. Получаемые из файла байты интерпретируются как строка в кодировки сервера; если байт не соответствует кодировки, вызывается ошибка.

Функция `pg_read_binary_file` схожа с `pg_read_file` за исключением того, что результатом является значение типа `bytea`, при этом контроль кодировки не производится. Совместно с функцией `convert_from` эта функция моэет быть использована для чтения файла в заданной кодировке:

```
SELECT convert_from(pg_read_binary_file('file_in_utf8.txt'), 'UTF8');
```

Функция `pg_stat_file` возвращает запись, содержащую размер указанного файла, метку времени последнего доступа, метку времени последней модификации, метку времени последнего изменения статуса (только для платформы Unix), метку времени создания (только для платформы Windows), и признак того, файл это или папка.

Примеры:

1. `SELECT * FROM pg_stat_file('filename');`
2. `SELECT (pg_stat_file('filename')).modification;`

6.26.9. Функции работы с прикладными блокировками

Функции, приведенные в таблице 99, обеспечивают работу с прикладными блокировками.

Таблица 99 – Функции работы с прикладными блокировками

Функция	Тип результата	Описание
`pg_advisory_lock(key bigint)`	`void`	Установка эксклюзивной блокировки уровня сессии
`pg_advisory_lock(key1 int, key2 int)`	`void`	Установка эксклюзивной блокировки уровня сессии

Продолжение таблицы 99

Функция	Тип результата	Описание
`pg_advisory_lock_shared(key bigint)`	`void`	Установка разделяемой блокировки уровня сессии
`pg_advisory_lock_shared(key1 int, key2 int)`	`void`	Установка разделяемой блокировки уровня сессии
`pg_advisory_unlock(key bigint)`	`boolean`	Снятие эксклюзивной блокировки уровня сессии
`pg_advisory_unlock(key1 int, key2 int)`	`boolean`	Снятие эксклюзивной блокировки уровня сессии
`pg_advisory_unlock_all()`	`void`	Снятие всех блокировок, установленных в текущей сессии
`pg_advisory_unlock_shared(key bigint)`	`boolean`	Снятие разделяемой блокировки уровня сессии
`pg_advisory_unlock_shared(key1 int, key2 int)`	`boolean`	Снятие разделяемой блокировки уровня сессии
`pg_advisory_xact_lock(key bigint)`	`void`	Установка эксклюзивной блокировки уровня транзакции
`pg_advisory_xact_lock(key1 int, key2 int)`	`void`	Установка эксклюзивной блокировки уровня транзакции
`pg_advisory_xact_lock_shared(key bigint)`	`void`	Установка разделяемой блокировки уровня транзакции
`pg_advisory_xact_lock_shared(key1 int, key2 int)`	`void`	Установка разделяемой блокировки уровня транзакции
`pg_try_advisory_lock(key bigint)`	`boolean`	Установка эксклюзивной блокировки уровня сессии, если возможно
`pg_try_advisory_lock(key1 int, key2 int)`	`boolean`	Установка эксклюзивной блокировки уровня сессии, если возможно
`pg_try_advisory_lock_shared(key bigint)`	`boolean`	Установка разделяемой блокировки уровня сессии, если возможно
`pg_try_advisory_lock_shared(key1 int, key2 int)`	`boolean`	Установка разделяемой блокировки уровня сессии, если возможно
`pg_try_advisory_xact_lock(key bigint)`	`boolean`	Установка эксклюзивной блокировки уровня транзакции, если возможно
`pg_try_advisory_xact_lock(key1 int, key2 int)`	`boolean`	Установка эксклюзивной блокировки уровня транзакции, если возможно
`pg_try_advisory_xact_lock_shared( key bigint)`	`boolean`	Установка разделяемой блокировки уровня транзакции, если возможно

Окончание таблицы 99

Функция	Тип результата	Описание
`pg_try_advisory_xact_lock_shared( key1 int, key2 int)`	`boolean`	Установка разделяемой блокировки уровня транзакции, если возможно

Функция `pg_advisory_lock` осуществляет установку прикладной блокировки, идентифицируемой одиночным 64-разрядным значением или двумя 32-разрядными (необходимо отметить, что эти два ключевых пространства не пересекаются). Если указанная блокировка уже установлена в другой сессии, функции задерживается до того времени, пока она станет доступной. Блокировка является эксклюзивной. При установке блокировки используется стек, так что, количество снятий блокировки должно равняться количеству ее установки.

Функция `pg_advisory_lock_shared` действует аналогично `pg_advisory_lock`, за исключением того, что блокировка может быть разделена между сессиями. При этом реально блокируются только попытки установки эксклюзивной блокировки на указанный ресурс.

Функция `pg_try_advisory_lock` похожа на `pg_advisory_lock`, но при этом не ожидает освобождение занятой блокировки. Если блокировка возможна в данным момент времени, функция устанавливает блокировку и возвращает истину, в противном случае возвращается ложь.

Функция `pg_try_advisory_lock_shared` действует аналогично `pg_try_advisory_lock`, но для разделяемых блокировок.

Функция `pg_advisory_unlock` снимает ранее установленную блокировку. Функция возвращает истину, если блокировка была успешно снята. Если же в действительности блокировки не была установлена, возвращается ложь, в дополнение к этому сервер может выдавать SQL предупреждение.

Функция `pg_advisory_unlock_shared` действует аналогично `pg_advisory_unlock`, но для разделяемых блокировок.

Функция `pg_advisory_unlock_all` снимает все блокировки, установленные в текущей сессии. Указанная функция неявно вызывается в конце сессии, даже если связь с клиентом была потеряна.

Функция `pg_advisory_xact_lock` действует аналогично `pg_advisory_lock`, за исключением того, что блокировка автоматически освобождается по завершению текущей транзакции и не может быть освобождена явно.

Функция `pg_advisory_xact_lock_shared` действует аналогично `pg_advisory_lock_shared`, за исключением того, что блокировка автоматически освобождается по завершению текущей транзакции и не может быть освобождена явно.

Функция pg_try_advisory_xact_lock действует аналогично pg_try_advisory_lock, за исключением того, что блокировка автоматически освобождается по завершению текущей транзакции и не может быть освобождена явно.

Функция pg_try_advisory_xact_lock_shared действует аналогично pg_try_advisory_lock_shared, за исключением того, что блокировка автоматически освобождается по завершению текущей транзакции и не может быть освобождена явно.

6.27. Триггерные функции

В настоящий момент PostgreSQL поддерживает одну встроенную триггерную функцию, suppress_redundant_updates_trigger, которая предотвращает выполнения обновлений, которые в действительности не меняют значения данных, в отличие от стандартного поведения, при котором действия обновления выполняются, даже если данные не меняются. Такое поведение делает обновление более быстрым, поскольку не требует проверок, что удобно в некоторых случаях.

В идеале следует избегать обновлений, которые в действительности не меняют данные. Ненужные обновления могут требовать большого количества излишнего времени, особенно при большом количестве обновлений индексов и пространства в удаленных строках, которое со временем должно быть освобождено. Однако определение подобной ситуации в прикладном приложении зачастую нелегко, или даже невозможно, и создание выражений, определяющих это, легко приводит к ошибкам. Альтернативой этому является применение функции suppress_redundant_updates_trigger, пропускающей обновления, которые не меняют данных. В тоже время это должно выполнятся с осторожностью. Триггер затрачивает на каждую строку хоть и малое, но время, так что при большом количестве изменяемых данных, использование этого триггера может замедлить процесс обновления.

Триггерная функция suppress_redundant_updates_trigger может быть добавлена к таблице следующим образом:

```
CREATE TRIGGER z_min_update
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE PROCEDURE suppress_redundant_updates_trigger();
```

В большинстве случаев желательно вызывать этот триггер последним для каждой строки. Необходимо помнить, что триггера вызываются в порядке их имен, так что следует выбирать имя для этого триггера большим, чем все остальные триггера, ассоциированные с таблицей.

Информация по созданию триггеров приведена в описании команды CREATE TRIGGER.

6.28. Событийные триггерные функции

Примечание. Событийные триггеры представлены в PostgreSQL с версии 9.6.

В настоящий момент PostgreSQL поддерживает одну встроенную событийную вспомогательную триггерную функцию pg_event_trigger_dropped_objects.

Функция pg_event_trigger_dropped_objects возвращает список объектов, удаленных командной, для которой было вызвано событие sql_drop. При использовании в ином контексте функция pg_event_trigger_dropped_objects вызывает ошибку. Функция pg_event_trigger_dropped_objects возвращает следующие столбцы:

Таблица 100 – Столбцы pg_event_trigger_dropped_objects

Имя	Тип	Описание
classid	Oid	OID каталога, к которому принадлежит объект
objid	Oid	OID объекта в пределах каталога
objsubid	int32	Дополнительный идентификатор внутри объекта (например, номер атрибута для столбцов)
object_type	text	Тип объекта
schema_name	text	Имя схемы, которой принадлежит объект, или NULL. Заключение в кавычки не применяется.
object_name	text	Имя объекта, если сочетание этого имени со именем схемы позволяет однозначно идентифицировать объект, иначе NULL. Заключение в кавычки не применяется, имя приводится без указания схемы.
object_identity	text	Текст, позволяющий однозначно идентифицировать объект, с указанием схемы. Каждая часть при необходимости заключается в кавычки.

Функция pg_event_trigger_dropped_objects может быть использована как событийный триггер следующим образом:

```
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 PROCEDURE test_event_trigger_for_drops();
```

7. ЗАПУСК И ИСПОЛЬЗОВАНИЕ СЕРВЕРА СУБД POSTGRESQL

В разделе описаны установка сервера управления базами данных PostgreSQL и его взаимодействие с ОС.

Далее описаны базовые процедуры работы с СУБД PostgreSQL с использованием ее приложений, команд, и настроек по умолчанию.

ВНИМАНИЕ! Для управления работой СУБД PostgreSQL в составе ОС настоятельно рекомендуется использовать специальные средства управления СУБД, входящие в состав ОС (см. **??**), поскольку настройки по умолчанию и расположение необходимых для работы СУБД файлов могут отличаться от базовых.

7.1. Учетная запись пользователя

Как и любой другой сервис, сервер СУБД PostgreSQL рекомендуется запускать от имени специальной учетной записи пользователя в ОС. Такой пользователь должен являться владельцем только тех данных (каталогов и файлов), которые управляются сервером СУБД, и не должен использоваться для запуска других процессов сервисов. Например, настоятельно не рекомендуется использование учетной записи пользователя `nobody`, т. к. установка исполняемых файлов, владельцем которых является данный пользователь, в случае компрометации процесса сервера СУБД приведет к тому, что удаленный анонимный пользователь получит возможность модифицировать исполняемые файлы.

Для добавления учетной записи пользователя используется команда `useradd` или `adduser`. Как правило, применяется учетная запись пользователя с именем `postgres`, но может быть выбрано любое другое имя.

Примечание. При установке СУБД из состава ОС необходимые каталоги, конфигурационные файлы и специальная учетная запись сервера `postgres` создаются автоматически.

7.2. Создание кластера БД

Перед началом работы необходимо выделить на диске область для хранения БД, называемую «кластером БД» или «каталог кластера» (catalog cluster). Кластер БД является набором БД, управляемых одним субъектом доступа (одной учетной записью пользователя ОС), от имени которого запущен процесс сервера СУБД. После инициализации в кластере БД содержится одна БД с именем `postgres`, которая является БД для использования по умолчанию разными утилитами, пользователями и приложениями. Для функционирования самого сервера СУБД существование БД с именем `postgres` не является необходимым условием, но многими внешними утилитами предполагается ее наличие. Другой БД, создаваемой при инициализации каждого кластера, является БД с именем `template1`. Она

применяется в качестве шаблона при создании БД и не должна использоваться для реальной работы. Дополнительные сведения по созданию новых БД в кластере приведены в 10.

В терминах ФС кластер БД является выделенным каталогом, в котором хранятся все данные. Указанный каталог называется «каталогом данных» или «областью данных». Место размещения данных выбирается произвольно. Значение по умолчанию отсутствует. Наиболее часто для хранения данных используется каталог `/usr/local/pgsql/data` или `/var/lib/pgsql/data`. Инициализация кластера БД производится утилитой `initdb`, установленной с СУБД PostgreSQL. Место размещения кластера БД задается опцией `-D` при выполнении команды `initdb`:

```
$ initdb -D /usr/local/pgsql/data
```

Данная команда должна выполняться от имени учетной записи пользователя PostgreSQL (см. 7.1).

В качестве альтернативы опции `-D` может быть установлено значение переменной окружения `PGDATA`.

Утилита `initdb` может быть запущена с помощью утилиты утилиты `pg_ctl` следующим образом:

```
$ pg_ctl -D /usr/local/pgsql/data initdb
```

Это может быть более понятным, если утилита `pg_ctl` уже используется для запуска и остановки сервера (см. 7.3), так что `pg_ctl` может являться единой командой управления экземпляром сервера СУБД.

Если указанного каталога кластера не существует, то при выполнении команды `initdb` осуществляется попытка его создания. При использовании непривилегированной учетной записи у пользователя сервера СУБД PostgreSQL будут отсутствовать права для создания каталога. В этом случае необходимо создать этот каталог от имени учетной записи `root`, а потом указать в качестве владельца пользователя СУБД PostgreSQL. Для выполнения данной операции может быть использована следующая последовательность команд:

```
root# mkdir /usr/local/pgsql/data
root# chown postgres /usr/local/pgsql/data
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

В случае, если каталог данных уже инициализирован, команда `initdb` завершит работу.

Поскольку каталог данных содержит все данные, хранимые в БД, необходимо, чтобы он был защищен от несанкционированного доступа. Поэтому `initdb` удаляет права доступа к этому каталогу для всех пользователей, кроме учетной записи пользователя СУБД

PostgreSQL.

Однако, несмотря на то, что содержимое каталога защищено, настройка аутентификации клиента по умолчанию позволяет любому локальному пользователю подключиться к БД или даже стать суперпользователем СУБД. В случае, если локальные пользователи не являются доверенными лицами, рекомендуется воспользоваться одной из следующих опций команды `initdb` для задания пароля суперпользователя СУБД: `-W`, `--pwprompt` или `--pwfile`. Кроме того, можно использовать опции `-A md5` или `-A password` для отключения используемого по умолчанию режима аутентификации `trust`, или, по завершении выполнения команды `initdb`, отредактировать файл `pg_hba.conf` перед первым запуском сервера СУБД PostgreSQL. Другим возможным подходом является использование режима аутентификации `peer` или настройка разрешений в ФС, задающих ограничения для соединений. Дополнительная информация приведена в 9.

Кроме того, команда `initdb` устанавливает для кластера БД региональные настройки по умолчанию. Обычно для этого берутся региональные настройки окружения и применяются к инициализированной БД, но возможно определить для БД и другие региональные настройки. Подробнее об этом можно прочитать в 12.1. Порядок сортировки, используемый в БД, определяется посредством команды `initdb` и впоследствии не может быть изменен без полной выгрузки всех данных, повторного выполнения команды `initdb` и перезагрузки данных. Использование форматов представления региональных настроек, отличных от C и POSIX, может привести к снижению производительности сервера СУБД.

Также команда `initdb` устанавливает для кластера БД кодировку по умолчанию. Обычно она выбирается соответствующей региональным настройкам. Подробнее об этом можно прочитать в 12.3.

7.2.1. Сетевая ФС

Во многих случаях применения кластер БД создается на сетевой ФС (Network File System, NFS). Данная операция выполняется либо непосредственно через сервис NFS, либо с применением сетевой системы хранения данных (Network Attached Storage, NAS) — устройства с внутренним использованием NFS. В СУБД PostgreSQL отсутствуют специальные средства для поддержки NFS. Таким образом, СУБД PostgreSQL предполагает, что NFS ведет себя точно так же, как и локально подсоединенные устройства хранения (Direct Attached Storage, DAS). Использование в реализации клиента и сервера NFS нестандартной семантики может привести к возникновению проблемы достоверности данных. В частности, при асинхронной записи на NFS-сервер. Во избежание указанных проблем следует монтировать NFS синхронно (без кэширования). Кроме того, не рекомендуется использовать режим `soft` при монтировании NFS. Сети хранения данных (Storage Area Networks, SAN) вместо NFS используют протоколы доступа низкого уровня.

7.3. Запуск сервера

Для получения доступа к БД необходимо запустить сервер СУБД с помощью команды `postgresql`, которая указывает место расположения данных. Данная операция может быть выполнена с использованием опции `-D`:

```
$ postgresql -D /usr/local/pgsql/data
```

Выполнение команды осуществляется при условии, что выполнен вход от имени учетной записи сервера PostgreSQL. Если опция `-D` не используется, то сервер в качестве каталога данных будет использовать значение переменной окружения `PGDATA`. Если значение указанной переменной также не задано, запуск сервера завершится с ошибкой:

В общем случае, лучше осуществлять запуск сервера СУБД в фоновом режиме, используя обычный синтаксис оболочки командной строки:

```
$ postgresql -D /usr/local/pgsql/data >logfile 2>&1 &
```

Целесообразно для осуществления аудита и диагностирования проблем обеспечить сохранение стандартного потока вывода `stdout` и стандартного потока ошибок `stderr`. Дополнительная информация приведена в 13.3.

Команда `postgres` также использует набор опций командной строки. Дополнительная информация приведена в 8.

Поскольку синтаксис командной строки может быть очень громоздким, то для упрощения решения ряда задач используется команда `pg_ctl`. Например, следующая команда запускает сервер в фоновом режиме и обеспечивает вывод в указанный файл журнала:

```
pg_ctl start -l logfile
```

Опция `-D` в данном случае используется также, как и в команде `postgres`. Также с помощью `pg_ctl` можно остановить сервер.

Для запуска сервера СУБД при загрузке компьютера используются скрипты автоматического запуска. Для установки данных скриптов необходимо иметь права суперпользователя (`root`).

Когда сервер СУБД PostgreSQL запущен, идентификатор его процесса (process identifier — PID) хранится в файле `postmaster.pid` в директории данных для исключения повторного запуска сервера для одной и той же директории данных. Кроме того, идентификатор процесса может быть использован для остановки сервера СУБД.

7.3.1. Ошибки, возникающие при запуске сервера

Существует несколько причин, по которым сервер может не запуститься. Необходимо проверить журнал сервера СУБД или осуществить запуск сервера вручную (без перенаправления стандартного потока вывода или стандартного потока ошибок) и проанализировать возникающие сообщения об ошибках. Далее рассмотрены наиболее распространенные сообщения об ошибках.

Следующее сообщение обычно означает, что запуск сервера СУБД осуществляется с использованием порта, который занят другим работающим сервером:

```
LOG:  could not bind IPv4 socket: Address already in use
HINT: Is another postmaster already running on port 5432?
If not, wait a few seconds and retry.
FATAL: could not create TCP/IP listen socket
```

Перевод сообщения:

```
LOG: Не удается привязать IPv4-сокет: Адрес уже используется.
HINT: Проверьте, используется ли порт 5432 другим процессом postmaster?
Если нет, подождите несколько секунд и повторите попытку.
FATAL: Не удается создать TCP/IP-сокет.
```

В случае, когда сообщение об ошибке не содержит пояснение «Address already in use» (адрес уже используется) или схожее с ним, то возникновение ошибки может быть обусловлено другой причиной. Например, при попытке запуска сервера с использованием порта с зарезервированным номером может быть получено следующее сообщение об ошибке:

```
$ postgres -p 666
LOG: could not bind IPv4 socket: Permission denied
HINT: Is another postmaster already running on port 666?
If not, wait a few seconds and retry.
FATAL: could not create TCP/IP listen socket
```

Перевод сообщения:

```
$ postgres -p 666
LOG: не удается привязать IPv4-сокет: Доступ запрещен.
HINT: Возможно, порт 666 занят другим процессом postmaster.
Если нет, повторите попытку через несколько секунд.
FATAL: Не удается создать TCP/IP-сокет.
```

Следующее сообщение об ошибке возможно означает, что ограничение ядра на объем разделяемой памяти меньше, чем рабочая область, которую пытается создать PostgreSQL (в данном примере — 4011376640 Байт):

```
FATAL: could not create shared memory segment: Invalid argument
DETAIL: Failed system call was shmget(key=5440001, size=4011376640, 03600).
```

Перевод сообщения:

```
FATAL: Не удалось создать сегмент разделяемой памяти: Недопустимый аргумент.
DETAIL: Ошибочный системный вызов shmget(key=5440001, size=4011376640, 03600).
```

Данное сообщение может означать, что поддержка разделяемой памяти в стиле System V не сконфигурирована в ядре. В качестве временного решения сервер СУБД может быть запущен с меньшим, чем нормальное, числом разделяемых буферов

(shared_buffers). В дальнейшем необходимо осуществить пересборку ядра для увеличения допустимого объема разделяемой памяти. Кроме того, данное сообщение может быть получено при попытке запуска нескольких серверов на одном компьютере в случае, когда общий размер запрашиваемой разделяемой памяти превышает ограничение ядра.

Следующее сообщение об ошибке не означает отсутствие свободного места на диске. Сообщение означает, что ограничение ядра на количество System V семафоров меньше, чем число семафоров, которое хочет создать PostgreSQL:

```
FATAL:  could not create semaphores: No space left on device
DETAIL: Failed system call was semget(5440126, 17, 03600).
```

Перевод сообщения:

```
FATAL: Невозможно создать семафоры: Нет места на диске.
DETAIL: Ошибочный системный вызов semget(5440126, 17, 03600).
```

Наиболее вероятная причина возникновения сообщения об ошибке типа «illegal system call» (запрещенный системный вызов) заключается в отсутствии в ядре поддержки разделяемой памяти и семафоров. В данном случае необходимо осуществить пересборку ядра с поддержкой разделяемой памяти и семафоров.

Дополнительная информация о настройке возможностей System V IPC приведена в 7.4.1.

7.3.2. Ошибки, возникающие на стороне клиента

Ошибки, возникающие на стороне клиента, характеризуются разнообразием и зависимостью от особенностей клиентских приложений. Однако ряд ошибок, возникающих на стороне клиента, находится в прямой зависимости от способа запуска сервера СУБД.

Рассмотрим следующее сообщение об ошибке:

```
psql: could not connect to server: Connection refused
Is the server running on host "server.joe.com" and accepting
TCP/IP connections on port 5432?
```

Перевод сообщения:

```
psql: Не удалось установить соединение с сервером. Соединение закрыто.
Запущен ли сервер на узле "server.joe.com" и принимает ли он
соединения по TCP/IP на порт 5432?
```

Данная ошибка является стандартной ошибкой типа «невозможно найти сервер, с которым необходимо связаться». Общим условием возникновения ошибок является отсутствие в конфигурации сервера разрешений на установку соединений TCP/IP.

Следующее сообщение об ошибке возникает при попытке подсоединения к локальному серверу с использованием доменного сокета Unix:

```
psql: could not connect to server: No such file or directory
Is the server running locally and accepting
```

```
connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

Перевод сообщения:

```
psql: Не удалось установить соединение с сервером:
отсутствует указанный файл или директория.
Запущен ли сервер локально и принимает ли он соединения на доменный
сокет Unix "/tmp/.s.PGSQL.5432"?
```

Последняя строка сообщения позволяет определить корректность параметров, используемых клиентом при установке соединения. В случае, если сервер вообще не запущен, в сообщении об ошибке будет указано «Connection refused» (соединение отклонено) или «No such file or directory» (отсутствует указанный файл или директория), как было показано выше. Следует отметить, что «Connection refused» в данном случае не означает, что сервер получил запрос на соединение и отверг его. Сообщение об ошибке, возникающее в подобном случае, описано в 9.4 . Другие сообщения об ошибках, такие как «Connection timed out» (соединение разорвано, т. к. за определенное время не был получен отклик от сервера) могут указывать на более серьезные проблемы, например отсутствие связи по сети.

7.4. Управление ресурсами ядра

Иногда PostgreSQL может превышать пределы использования системных ресурсов ОС, особенно в случае функционирования нескольких экземпляров сервера СУБД на одной системе, или при очень больших БД. В разделе описаны ресурсы ядра ОС, используемые PostgreSQL, и способы устранения проблем их использования.

7.4.1. Разделяемая память и семафоры

Общим наименованием для разделяемой памяти и семафоров является термин «System V IPC». Данный термин включает еще и очереди сообщений, которые не используются в СУБД PostgreSQL. Почти все современные ОС предоставляют данные возможности, но не во всех возможности их использования включены по умолчанию или настроены со значениями параметров так, что это позволяло бы обеспечить условия для работы сервера СУБД.

При появлении сообщения об ошибке «Illegal system call», возникающего при запуске сервера, требуется пересборка ядра, иначе СУБД PostgreSQL работать не будет.

В случае, когда требования PostgreSQL к ресурсам превышают одно из установленных в IPC ограничений, сервер не запустится, и будет выдано сообщение об ошибке, содержащее описание возникшей проблемы и инструкции по ее решению (см. 7.3.1). Параметры ядра, имеющие отношение к данному вопросу, называются в различных системах одинаково (см.таблицу 101), но методы их настройки отличаются. Ниже представлены предложения по установке значений параметров для ряда платформ. Следует учесть,

что для изменения значений приведенных параметров зачастую необходимо выполнить перезагрузку ОС, в отдельных случаях может потребоваться перекомпиляция ядра.

Таблица 101 – Параметры System V IPC

Параметр	Описание	Допустимое значение
SHMMAX	Максимальный размер сегмента разделяемой памяти, байты	Не менее 1КБ (больше, если запущено несколько копий сервера)
SHMMIN	Минимальный размер сегмента разделяемой памяти, байты	1
SHMALL	Общий объем доступной разделяемой памяти, байты или страницы	Для байтов — SHMMAX, для страниц — ceil(SHMMAX/PAGE_SIZE)
SHMSEG	Максимальное число сегментов разделяемой памяти на один процесс	Необходим только один сегмент, но значение по умолчанию гораздо выше
SHMMNI	Максимальное число сегментов разделяемой памяти в системе	SHMSEG + ресурсы для прочих приложений
SEMMNI	Максимальное число идентификаторов для семафоров	По меньшей мере ceil((max_connections + autovacuum_max_workers + 4) / 16)
SEMMNS	Максимальное число семафоров в системе	ceil((max_connections + autovacuum_max_workers + 4) / 16) * 17 + еще место для других приложений
SEMMSL	Максимальное число семафоров в наборе	Не менее 17
SEMMAP	Число записей в карте семафоров	Может потребоваться увеличение значения параметра SEMMAP, чтобы оно по порядку приблизилось к SEMMNS
SEMVMX	Максимальное значение семафора	Не менее 1000 (по умолчанию обычно — 32767, не рекомендуется менять без необходимости)

Операция ceil является операцией округления до ближайшего большего целого.

PostgreSQL требует незначительное количество байт разделяемой памяти «System V» (обычно 48 Байт для 64-битных платформ) для каждой копии сервера. В большинстве современных ОС это количество может быть выделено. Однако, в случае исполнения множества копий сервера, или других приложений, использующих разделяемую память «System V», может возникнуть необходимость увеличения максимального количества байт на сегмент разделяемой памяти SHMMAX или ограничение на общий объем разделяемой памяти в системе (SHMALL). Во многих случаях SHMALL измеряется не в байтах, а в страницах.

Наименьшее количество проблем возникает с минимальным размером сегментов разделяемой памяти (SHMMIN), который для PostgreSQL должен составлять не менее 32 (обычно установлен 1). Значения параметров — максимальное количество сегментов в системе (SHMMNI) или на один процесс (SHMSEG) — приводят к возникновению ошибок,

только если значения установлены, равными 0.

PostgreSQL использует один семафор на каждое допустимое соединение (max_connections) и на допустимый рабочий процесс автовакуума (autovacuum_max_workers), группируя их в наборы по 16. В каждом таком наборе есть 17-й семафор, который содержит «волшебное число», для обнаружения коллизий с наборами семафоров, используемыми другими приложениями. Максимальное число семафоров в системе задается переменной SEMMNS, значение которой должно быть не менее max_connections плюс autovacuum_max_workers плюс еще один дополнительный на каждые 16 допустимых соединений плюс рабочие процессы (см. таблицу 101). Параметр SEMMNI задает предельное число наборов семафоров, которое одновременно может быть в системе. Таким образом, этот параметр должен иметь значение не менее ceil((max_connections + autovacuum_max_workers + 4) / 16). Если функция semget выдает сообщение «No space left on device» (недостаточно места), в качестве временного решения возможно уменьшение числа допустимых соединений.

Иногда требуется увеличение значения параметра SEMMAP, чтобы оно по порядку приблизилось к SEMMNS. Данный параметр определяет размер карты ресурсов семафоров, в которой для каждого непрерывного блока семафоров необходима отдельная запись. Как только набор семафоров освобождается, он также добавляется в уже существующую запись, соответствующую свободному блоку, или для него создается новая запись. В случае заполнения карты семафоров, освобождаемые семафоры теряются до перезагрузки. Фрагментация пространства семафоров может через некоторое время привести к снижению числа доступных семафоров ниже требуемого значения.

Значение параметра SEMMSL, определяющего количество семафоров в наборе, для PostgreSQL должно быть не менее 17.

Прочие настройки, относящиеся к «semaphore undo», такие как SEMMNU и SEMUME для PostgreSQL не имеют значения.

По умолчанию максимальный размер сегмента равен 32 МБ, что является достаточным только для конфигураций PostgreSQL с низкими требованиями к системным ресурсам. Значения по умолчанию для прочих параметров не требуют внесения изменений. Максимальный размер сегмента разделяемой памяти может быть изменен командой sysctl. Например, чтобы разрешить 16 ГБ, необходимо выполнить:

```
$ sysctl -w kernel.shmmax=17179869184
$ sysctl -w kernel.shmall=4194304
```

Для сохранения настроек после перезагрузки используется файл /etc/sysctl.conf.

В состав старых дистрибутивов программа sysctl может не входить. В этом случае

эквивалентные изменения можно произвести, работая с ФС /proc:

```
$ echo 17179869184 >/proc/sys/kernel/shmmax
$ echo 4194304 >/proc/sys/kernel/shmall
```

7.4.2. Ограничения на системные ресурсы

Особенно важными являются ограничения числа процессов на одного пользователя и объема памяти, доступного для одного процесса. Для каждого из названных параметров устанавливается «жесткий» и «мягкий» предел. «Мягкий» предел — это текущее значение, которое пользователь может поднимать вплоть до «жесткого» предела. «Жесткий» предел может изменить только суперпользователь. За настройку данных параметров отвечает системный вызов setrlimit. В командной строке значения предела можно контролировать с помощью встроенной команды ulimit (Bourne shells) или limit (csh).

Следует обратить внимание на значение следующих параметров: maxproc, openfiles и datasize.

Пример

```
default:\
...
        :datasize-cur=256M:\
        :maxproc-cur=256:\
        :openfiles-cur=256:\
...
```

Суффикс -cur задает установку значения «мягкого» предела. Для установки значения «жесткого» предела используется суффикс -max.

Ядро также может иметь системные ограничения на некоторые ресурсы.

/proc/sys/fs/file-max определяет максимальное число открытых файлов, поддерживаемое ядром. Это значение можно изменить, указав другое число в файле, или, добавив присваивание в /etc/sysctl.conf. Максимальное число файлов для одного процесса фиксируется при компиляции ядра.

Сервер PostgreSQL использует один процесс на каждое соединение, следовательно должна быть обеспечена возможность создания процессов в количестве, как минимум, равном допустимому количеству соединений дополнительно к числу процессов, которое необходимо для функционирования остальной части системы. Проблемы с данным ограничением возникают только при необходимости обеспечить запуск нескольких серверов в одной системе.

По умолчанию ограничение на число открытых файлов установлено в значение, которое позволяет нескольким пользователям сосуществовать в одной системе и без использования недопустимого дробления системных ресурсов. Подобный подход пригоден

при запуске нескольких серверов в одной системе. В случае использования выделенного сервера может появиться необходимость увеличить значение ограничения.

В случае, если несколько процессов одновременно открывают большое количество файлов, ограничение системы может быть превышено. Для решения данной проблемы без изменения системного ограничения на число открытых файлов существует возможность задать значение параметра конфигурации PostgreSQL `max_files_per_process` для установки ограничения на число открытых файлов.

7.4.3. Избыточное использование памяти (Linux Memory Overcommit)

В Linux 2.4 и более старших версий использование виртуальной памяти по умолчанию является не оптимальным для PostgreSQL. Применение ядром memory overcommit может привести к остановке ядром сервера PostgreSQL (мастер-процесса сервера) в случае, когда потребности другого процесса в памяти вызовут нехватку виртуальной памяти в системе.

В данном случае будет выдано следующее сообщение ядра:

```
Out of Memory: Killed process 12345 (postgres).
```

Данное сообщение означает, что процесс `postgres` был остановлен по причине нехватки памяти. Существующие соединения с БД будут функционировать, новые соединения устанавливаться не будут. Для восстановления нормального функционирования необходимо перезапустить PostgreSQL.

Во избежание возникновения указанной проблемы необходимо запускать PostgreSQL в системе, для которой существует гарантия, что ни один процесс не займет всю память. В случае недостатка памяти можно увеличить размер области подкачки (`swap`) поскольку остановка процесса вследствие нехватки памяти (OOM – out-of-memory) осуществляется только тогда, когда исчерпана как физическая память, так и область подкачки `swap`.

В том случае, если PostgreSQL является причиной нехватки виртуальной памяти в системе, существует возможность избежать этого путем изменения конфигурационных настроек СУБД, связанных с использованием памяти, такие как `shared_buffers` и `work_mem`. В других случаях причиной может быть большое разрешенное количество соединений с сервером. В большинстве случаев следует уменьшать значения параметра `max_connections` и использовать стороннее ПО, обеспечивающее использование соединений с СУБД.

В Linux, начиная с версии 2.6, в качестве дополнительной меры существует возможность настроить ядро для предотвращения чрезмерного использования (overcommit) памяти. Данная мера не является полной гарантией от срабатывания остановки процессов OOM, однако, значительно снижает вероятность подобного события и, следовательно, приводит к более устойчивому поведению системы. Для этого необходимо выбрать `strict overcommit mode` посредством утилиты `sysctl`:

```
sysctl -w vm.overcommit_memory=2
```

или добавить эквивалентную запись в /etc/sysctl.conf. Для изменения значения параметра vm.overcommit_ratio необходимо руководствоваться документацией на ядро Documentation/vm/overcommit-accounting.

7.4.4. Большие страницы Linux

Примечание. Поддержка больших страниц реализована только в СУБД версии 9.6.

Использование больших страниц уменьшает накладные расходы при использовании больших смежных участков памяти, как PostgreSQL делает. Для того, чтобы использовать эту возможность, вам необходимо ядро с CONFIG_HUGETLBFS=y и CONFIG_HUGETLB_PAGE=y. Вам также необходимо настроить системные настройки vm.nr_hugepages. Чтобы оценить количество необходимых больших страниц, необходимых для старта PostgreSQL без поддержки PostgreSQL этого механизма, проверьте значение VmPeak файловой системе proc:

```
$ head -1 /path/to/data/directory/postmaster.pid
4170
$ grep ^VmPeak /proc/4170/status
VmPeak:  6490428 kB
```

6490428 / 2048 (PAGE_SIZE равен 2MB в этом случае) приближенно равен 3169.154 больших страниц, соответственно, вам необходимо как минимум 3170 больших страниц:

```
$ sysctl -w vm.nr_hugepages=3170
```

Иногда ядро не позволяет выделять желаемое число больших страниц, в таком случае стоит попробовать выполнить эту команду или перезагрузить систему. Не забудьте добавить запись в файл /etc/sysctl.conf для сохранения этой настройки после перезагрузкок системы.

Поведение больших страниц в PostgreSQL по умолчанию – это использовать их, когда возможно и возвращаться к нормальным страницам при ошибках. Для включения возможности использования больших страниц, вам необходимо установить параметр huge_pages в on. Отметим, что в этом случае PostgreSQL может завершаться с ошибкой при старте, если недостаточно больших страниц для его запуска.

7.5. Остановка сервера

Сервер БД может быть остановлен несколькими способами. Способ отключения выбирается с помощью различных сигналов, посылаемых мастер-процессу postgres.

1) SIGTERM

«Умное» отключение (Smart Shutdown). Получив сигнал SIGTERM, сервер запрещает

установку новых соединений, но дает возможность существующим сессиям закончить работу. Сервер выключается только после нормального закрытия всех сессий. В случае, когда сервер находится в состоянии резервного копирования, ожидается выход из данного состояния. В данном режиме остановки сервера допускается создание новых подключений, но только для суперпользователей. Подобная возможность позволяет суперпользователю установить соединение с сервером и завершить резервное копирование.

2) SIGINT

«Быстрое» отключение (Fast Shutdown). Сервер запрещает создание новых подключений и посылает всем запущенным процессам сигнал SIGTERM, который заставит их прервать текущие транзакции и немедленно выйти. Затем сервер ожидает завершения всех процессов и завершает работу. В случае когда сервер находится в состоянии резервного копирования, копирование прерывается.

3) SIGQUIT

«Немедленное» отключение (Immediate Shutdown). Мастер-процесс postgres посылает сигнал SIGQUIT всем дочерним процессам и выходит немедленно без собственного правильного завершения. Работа всех дочерних процессов завершается при получении сигнала SIGQUIT, что приводит к необходимости восстановления данных при следующем запуске (путем воспроизведения журнала WAL). Данный режим остановки сервера рекомендуется использовать только в экстренных случаях.

Примечание. В СУБД версии 9.6 «немедленное» отключение работает иначе: сервер отправляет SIGQUIT всем дочерним процессам и ждет их заверешения. Тем процессам, которые не завершились в течение 5 секунд самостоятельно, будет отправлен сигнал SIGKILL мастер-процессом postgres, которые прекращают свою работу без ожидания.

Интерфейс для отправки рассмотренных сигналов серверу предоставляет утилита pg_ctl. Кроме того, может быть использована команда kill. PID процесса postgres можно определить с использованием утилиты ps или в файле postmaster.pid в директории данных.

Пример

Выполняем быстрое выключение:

```
$ kill -INT 'head -1 /usr/local/pgsql/data/postmaster.pid'
```

Следует отметить, что использование для остановки сервера сигнала SIGKILL не рекомендуется, поскольку в данном случае сервер не освобождает разделяемую память и семафоры, и приведет к необходимости выполнения названных операций вручную перед

запуском нового сервера. Кроме того, сигнал SIGKILL приводит к завершению работы процесса postgres, не позволяя ему отослать сигнал на выключение своим дочерним процессам, что приводит к необходимости завершать дочерние процессы вручную.

Закрыть одну сессию, позволяя всем остальным продолжать работу, можно с помощью pg_terminate_backend() (см. 6.26.2) или отправив сигнал SIGTERM дочернему процессу, соответствующему данной сессии.

7.6. Обновление кластера PostgreSQL

В разделе описана процедура обновления баз данных при переходе на новую версию PostgreSQL.

Номер основной версии PostgreSQL представлен первыми двумя цифрами номера версии, например 8.4. Дополнительный номер версии PostgreSQL представлен третьей группой цифр, например 8.4.2 означает второй дополнительный выпуск версии 8.4. Дополнительный выпуск никогда не содержит изменения внутреннего формата хранения данных и всегда совместим с более ранними или поздними выпусками той же основной версии, так что выпуск 8.4.2 совместим с 8.4.1 и 8.4.6. Для перехода между дополнительными выпусками достаточно заменить исполняемые модули при остановленном сервере и перезапустить его. Каталог данных при этом остается без изменений.

Более сложен переход между разными основными версиями PostgreSQL, поскольку при этом меняется внутренний формат хранения данных. Традиционным методом переноса данных на новую версию является сохранение резервной копии данных в одной версии и восстановление в новой, хотя этот метод может быть медленным. Наиболее быстрым методом является обновление с использованием pg_upgrade в соответствии с 7.6.2. Методы репликации также можно использовать, как описано ниже.

Новая основная версия как правило содержит заметные пользователю несовместимости, возможно требующие изменения в прикладном ПО. Все изменения в новой версии приведены в описании изменения версии, при этом особое внимание должно быть уделено пункту «Миграция». При переходе через несколько основных версий следует ознакомится с подобными разделами для каждой промежуточной версии.

Хорошей практикой является проверка клиентских приложений на новой версии перед полным переходом на нее. При переходе на новую основную версию PostgreSQL все изменения можно разделить на следующие категории:

- Администрирование — возможности для мониторинга и управления сервером часто изменяются и улучшаются в каждой последующей версии.
- SQL — обычно добавляются новые команды SQL, без изменения в их поведении, если это специально не указано.

- Прикладной программный интерфейс (Library API) — обычно в библиотеки типа `libpq` добавляется новый функционал, если специально не указано иное.
- Системный каталог — системный каталог изменяется только для отражения новых функций управления базами данных.
- Прикладной программный C интерфейс сервера (Server C-language API) — включает изменения в функциях, доступны для разработки серверных расширений на языке C. Подобные изменения связаны с внутренними функциями сервера.

7.6.1. Обновление с помощью `pg_dumpall`

Для переноса данных из одной версии PostgreSQL в другую, необходимо использовать `pg_dump`; методы резервного копирования на уровне файловой системы не будут работать. (Есть проверки, которые предотвращают использование данных каталога с несовместимой версии PostgreSQL, так что вреда не будет при попытке чтения неподдерживаемой версии PostgreSQL каталога данных).

Рекомендуется применять утилиты `pg_dump` и `pg_dumpall` из новой версии PostgreSQL, для использования всех возможных их усовершенствований. Существующие версии утилит резервирования позволяют читать данных со всех версий сервера с 7.0.

Следующие инструкции подразумевают расположение существующего экземпляра сервера в `/usr/local/pgsql`, а каталога данных в `/usr/local/pgsql/data`. При выполнении должны быть указаны соответствующие каталоги.

1) При создании резервной копии следует убедится, что база данных не находится в режиме обновления. Это не сказывается на целостности резервной копии, но изменения естественным образом в нее включены не будут. При необходимости могут быть изменены права доступа к конфигурационному файлу `/usr/local/pgsql/data/pg_hba.conf` (или эквивалентному) для предотвращения к нему доступа других лиц.

Для сохранения копии кластера следует выполнить:

```
pg_dumpall > outputfile
```

При создании резервной копии может быть использована утилита `pg_dumpall` из существующей версии, но для лучшего результата рекомендуется использовать эту утилиту из более новой, поскольку она может содержать некоторые улучшения. На этом шаге рекомендуется установить параллельно сервер PostgreSQL новой версии, чтобы сократить время простоя при обновлении сервера.

2) Остановка сервера:

```
pg_ctl stop
```

Для систем, в которых PostgreSQL запускается при старте, остановку следует выполнять следующим образом:

```
/etc/init.d/postgresql stop
```

3) При восстановлении резервной копии, следует переименовать или удалить старые каталоги. Лучшим является переименование, на случай возвращения к старым данным при возможных проблемах. Следует учитывать, что каталоги могут занимать значительное дисковое пространство. Для переименования каталога, используется такая команда как:

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

(Каталог данных должен быть скопирован как единое целое с сохранением внутренних относительных путей.)

4) Установка новое версии PostgreSQL согласно инструкции по установке.

5) Создание при необходимости нового кластера. Это должно быть выполнено от имени предопределенного системного пользователя сервера баз данных (при обновлении, подобный пользователь уже существует).

```
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

6) Восстановление предыдущих изменений в конфигурационных файлах `pg_hba.conf` и `postgresql.conf`.

7) Запуск сервера баз данных от имени предопределенного системного пользователя сервера баз данных:

```
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
```

8) Восстановление резервной копии кластера с помощью новой версии утилиты `psql`:

```
/usr/local/pgsql/bin/psql -d postgres -f outputfile
```

Время неработоспособности сервера может быть уменьшено в случае установки новой версии в другие каталоги и одновременный запуск старой и новой версии кластера на разных портах. После этого перенос данных может быть осуществлен следующим образом: `pg_dumpall -p 5432 | psql -d postgres -p 5433`

7.6.2. Методы обновления с помощью `pg_upgrade`

Модуль `pg_upgrade` позволяет осуществить переход «по-месту» с одной версии PostgreSQL на следующую. Обновление может быть выполнено в течение нескольких минут, в частности, при использовании опции `--link`. Это также требует действий, подобные

pg_dumpall выше, например, запуска/остановки сервера, запуска initdb.

7.6.3. Методы обновления с помощью средств репликации

Также возможно применение некоторых средств репликации, таких как Slony, для создания резервного сервера с новой версией PostgreSQL. Это возможно, поскольку Slony поддерживает репликацию между разным версиями PostgreSQL. Резервный сервер PostgreSQL может располагаться как на том же самом, так и на другом компьютере. После синхронизации резервного сервера с основным (запущенным на старой версии PostgreSQL), резервный сервер делается основным, а старый останавливается. Такой подход позволяет сократить время неработоспособности сервера при обновлении до нескольких секунд, требующихся для переключения.

7.7. Предотвращение подмены сервера

Во время работы сервера нарушитель не может выполнить подмену сервера СУБД. В случае когда сервер СУБД выключен локальный пользователь может подменить его, запустив свой собственный сервер. Сервер нарушителя имеет возможность прочесть пароли и запросы, посылаемые клиентами, но не имеет возможности ответить, поскольку директория PGDATA защищена настроенными правами доступа. Возникновение подобной ситуации возможно, т.к. любой пользователь может запустить сервер БД. Клиент не сможет распознать ложный сервер, не использующий особый вариант конфигурации.

Наиболее простой способ запретить появление ложных серверов для локальных соединений заключается в использовании директории доменных сокетов Unix (unix_socket_directory), имеющей разрешение на запись только для доверенного локального пользователя. Подобный подход предотвращает создание нарушителем своего файла сокетов в данной директории. Некоторые приложения могут использовать директорию /tmp для файла сокетов и могут быть подвержены спуфингу. Во время загрузки системы создайте символическую ссылку /tmp/.s.PGSQL.5432, которая будет указывать на перемещенный файл сокетов. Для сохранения названной ссылки измените скрипт очистки /tmp.

По TCP сервер должен принимать только соединения, указанные с помощью параметра hostssl в файле pg_hba.conf (см. 9.1) и должен иметь файлы SSL server.key (файл ключа) и server.crt (соответствующий ключу файл сертификата) (см. 7.9). TCP-клиент должен соединяться с использованием sslmode='verify-ca' или sslmode='verify-full' и иметь необходимые файлы ключа и сертификата.

7.8. Возможности маскирующих преобразований

СУБД PostgreSQL позволяет использовать маскирующее преобразование на нескольких уровнях и обеспечивает защиту от нарушения конфиденциальности данных в случае получения нарушителем доступа к носителям с БД, при возникновении ошибок администрирования и незащищенных сетей.

- Маскирующее преобразование паролей

По умолчанию все пароли пользователей БД хранятся в преобразованном виде, полученном с использованием хэш-функции MD5, следовательно администратор не имеет возможности узнать пароль пользователя. Использование преобразования хэш-функции MD5 при аутентификации клиента обеспечивает невозможность появления на сервере пароля в открытом виде даже временно, поскольку пароль преобразуется на стороне клиента перед отправкой.

- Маскирующее преобразование отдельных столбцов

Библиотека функций `pgcrypto` позволяет хранить определенные поля в замаскированном виде. Данная возможность полезна для обеспечения конфиденциальности конкретных данных. Клиент вводит ключ, данные преобразуются в открытый вид сервере и затем отправляются клиенту.

В открытом виде данные и ключ присутствуют на сервере короткое время при преобразовании и передаче между клиентом и сервером. Существует короткий временной интервал, предоставляющий возможность для перехвата данных или ключей пользователю, имеющему полный доступ к серверу СУБД, например, системному администратору.

- Маскирующее преобразование раздела данных

Шифрование хранилища может быть выполнена на уровне файловой системы или на уровне блоков. Опции шифрования файловой системы Linux системы включают в себя eCryptfs и EncFS, в то время как FreeBSD использует PEFS. Уровень блоков или полное шифрование диска включают DM-crypt + LUKS на Linux и GEOM модули geli и GBDE на FreeBSD. Многие другие операционные системы поддерживают этой функциональности, в том числе Windows.

Данный механизм предотвращает возможность чтения данных с диска в случае кражи дисков или всего компьютера. Однако данный механизм не обеспечивает защиту от атак в то время когда ФС смонтирована и ОС предоставляет доступ к данным в открытом виде. Для монтирования ФС необходимо определить способ передачи ключа ОС, избегая хранения ключа на узле, на котором монтируется диск.

- Маскирующее преобразование паролей в сети

Метод аутентификации с использованием хэш-функции MD5 дважды преобразовы-

вает пароль на стороне клиента перед отправкой на сервер. Сначала производится MD5-преобразование на основе имени пользователя, затем на основе случайной комбинации (`salt`), присланной сервером при подключении к БД. По сети передается именно это дважды преобразованное значение. Такое двойное преобразование предотвращает не только вскрытие пароля, но и повторные подключения к БД с использованием одинаково преобразованного пароля.

- Маскирующее преобразование данных в сети

Соединение с использованием протокола SSL преобразовывает все переданные по сети данные: пароль, запросы и возвращаемые данные. Администратор может задать в файле `pg_hba.conf` перечень узлов, которые могут использовать открытые соединения (параметр `host`), и перечень узлов, требующих использования соединений по протоколу SSL (параметр `hostssl`). Кроме того, клиенты при соединении с сервером могут требовать обязательного использования протокола SSL. `Stunnel` или `SSH` также могут быть использованы для маскирующего преобразования передаваемых данных.

- Аутентификация узла с использованием SSL

Существует возможность обеспечения обмена клиента и сервера SSL-сертификатами. Данный механизм требует дополнительного конфигурирования с обеих сторон, обеспечивая более надежную идентификацию и аутентификацию по сравнению с проверкой подлинности лишь на основе пароля. Кроме того, данный механизм предотвращает атаки с подменой сервера, направленные на получение паролей клиентов и атаки с использованием «посредника», внедряемого в сеть между сервером и клиентом, а также осуществляющего чтение и передачу всех данных между клиентом и сервером.

- Маскирующее преобразование на стороне клиента

Клиент может использовать маскирующее преобразование данных для исключения их попадания на сервер СУБД в открытом виде. Маскирующее преобразование данных осуществляется на стороне клиента перед отправкой на сервер, а обратное преобразование данных, полученных в ответе от сервера СУБД осуществляется клиентом перед использованием.

7.9. Безопасные TCP/IP-соединения с использованием протокола SSL

В СУБД PostgreSQL существует поддержка использования соединений по протоколу SSL с использованием маскирующих преобразований для при взаимодействии клиента и сервера для повышения безопасности соединения. Для применения данного механизма необходимо наличие в ОС клиента и сервера установленного OpenSSL и соответствующая

опция для поддержки SSL в СУБД PostgreSQL должна быть включена при сборке.

Для использования SSL при запуске сервера СУБД PostgreSQL, скомпилированного с поддержкой SSL, необходимо установить в `on` значение параметра `ssl` в конфигурационном файле `postgresql.conf`. Сервер будет принимать стандартные и SSL-соединения, применяя один порт TCP, и договариваясь с каждым подключающимся клиентом об использовании SSL. По умолчанию названная возможность является опцией клиента, в 9.1 описано конфигурирование сервера для применения SSL на некоторых или всех соединениях.

СУБД PostgreSQL использует общий системный конфигурационный файл OpenSSL. По умолчанию файл называется `openssl.cnf` и находится в директории, которую посредством команды `openssl version -d`. Данная установка по умолчанию может быть изменена указанием в переменной окружения `OPENSSL_CONF` имени конфигурационного файла.

OpenSSL поддерживает широкий диапазон типов маскирующих преобразований и алгоритмов аутентификации различной длины. В то время как список типов маскирующих преобразований может быть определен в конфигурационном файле OpenSSL, вы можете отдельно указать типы маскирующих преобразований, которые будут использоваться сервером БД, изменив `ssl_ciphers` в `postgresql.conf`.

Следует отметить, что существует возможность снижения издержек на маскирующие преобразования, используя для аутентификации маскирующие преобразования типа `NULL-SHA` или `NULL-MD5`. Однако в таком случае внедренный посредник сможет читать и передавать сообщения между клиентом и сервером. Кроме того, затраты на маскирующие преобразования малы по сравнению с затратами на аутентификацию. Следовательно, использование маскирующих преобразований типа `NULL` не рекомендуется.

Для запуска в режиме SSL необходимо, наличие в директории данных файлов сертификата сервера `server.crt` и закрытого ключа `server.key`. Для файлов ключей могут быть заданы другие имена конфигурационными параметрами `ssl_cert_file` и `ssl_key_file`.

В Unix-системах необходимо запретить доступ к `server.key` для группы и всех при помощи команды `chmod 0600 server.key`. В случае защиты закрытого ключа парольной фразой, сервер выведет приглашение для ввода парольной фразы и не будет запущен, пока фраза не будет введена.

В некоторых случаях сертификат сервера может быть подписан "промежуточным"сертификатом, а не тем, которому непосредственно доверяют клиенты. Чтобы использовать такой сертификат, добавьте этот промежуточный сертификат в файл `server.crt`, далее в сертификат узла, выдавшего промежуточный сертификат, и так далее до удостоверяющего центра, т.о, "корневому"или "промежуточным"сертификатам будут доверять клиенты,

т.к. клиентам выдал сертификаты удостоверяющий центр.

7.9.1. Использование сертификатов клиента

Для проверки предоставляемого клиентом сертификата, необходимо поместить доверенные сертификаты, полученные от удостоверяющих центров (CA – certificate authorities), в файл `root.crt` в директории данных и присвойте параметру `clientcert` значение `1` в соответствующей строке `pg_hba.conf`. Сертификат будет запрашиваться у клиента при создании SSL-соединения. Сервер проверит, подписан ли сертификат клиента на одном из доверенных сертификатов. Если промежуточные центры сертификации появляются в `root.crt`, файл должен также содержать цепочек сертификатов их корневых центров сертификации. При установке соединения с помощью файла `root.crl`, будет проверяться и список отозванных сертификатов (Certificate Revocation List – CRL).

Опция `clientcert` в файле `pg_hba.conf` доступна для всех методов аутентификации, но только для строк, определенных в `hostssl`. Когда `clientcert` не задан или установлен в 0, сервер проверяет представляемые сертификаты клиентов с помощью списка удостоверяющих центров (CA), если таковой задан, но в этом случае сервер не требует обязательного представления клиентских сертификатов.

Примечание. `root.crt` перечисляет удостоверяющие центры верхнего уровня, рассматриваемые как доверенные для подписывания клиентских сертификатов. В принципе нет необходимости указывать удостоверяющий центр, подписавший сертификат сервера, хотя в большинстве случаев этот центр должен быть доверенным и для клиентских сертификатов.

При использовании для аутентификации пользователей клиентских сертификатов, можно применять метод аутентификации `cert` (см. 9.3.9).

7.9.2. Использование на сервере файлов для применения SSL

В таблице 102 приведен список файлов для использования SSL на сервере. В таблице приведены имена файлов по умолчанию, действительные имена могут отличаться.

Таблица 102 – Использование на сервере файлов для применения SSL

Файл	Содержимое	Действие
`ssl_cert_file` (`$PGDATA/server.crt`)	Сертификат сервера	Запрашивается клиентом
`ssl_key_file` (`$PGDATA/server.key`)	Закрытый ключ сервера	Подтверждает, что сертификат сервера отправлен владельцем, но не гарантирует, что владельцу сертификата можно доверять
`ssl_ca_file` (`$PGDATA/root.crt`)	Доверенные удостоверяющие центры	Для проверки подписи сертификата клиента

Окончание таблицы 102

Файл	Содержимое	Действие
ssl_crl_file ($PGDATA/root.crl)	Отозванные сертификаты	Сертификат клиента не должен быть в списке

Файлы server.key, server.crt, root.crt и root.crl проверяются только во время запуска сервера, поэтому для того, чтобы внесенные в них изменения вступили в силу, необходимо перезапустить сервер.

7.9.3. Создание самоподписанного сертификата

Для создания самоподписанного сертификата сервера необходимо выполнить следующую команду OpenSSL:

```
openssl req -new -text -out server.req
```

Следует заполнить информацию, запрашиваемую openssl. В качестве «Common Name» должно быть указано сетевое имя сервера, пароль может быть не задан. Программа создает ключ, защищенный ключевой фразой длиной не менее 4-х символов. Для удаления этой ключевой фразы (для автоматического запуска сервера), выполняются команды:

```
openssl rsa -in privkey.pem -out server.key
rm privkey.pem
```

Для разблокирования ключа необходимо ввести старую ключевую фразу. Затем выполнить:

```
openssl req -x509 -in server.req -text -key server.key -out server.crt
```

Все необходимые данные будут запрошены в интерактивном режиме.

Далее необходимо скопировать файлы закрытого ключа и сертификата сервера в определенные в параметрах конфигурации места и установить права доступа на файл закрытого ключа командой:

```
chmod og-rwx server.key
```

Дополнительная информация о создании закрытых ключей и сертификатов приведена в документации на OpenSSL.

Самоподписанный сертификат можно использовать при тестировании, однако в процессе реального функционирования для идентификации клиентами сервера рекомендуется использовать сертификат, подписанный в удостоверяющем центре.

7.10. Безопасные TCP/IP-соединения с использованием протокола SSH

Для маскирования данных передаваемых в рамках сетевого соединения между клиентами и сервером PostgreSQL может быть использован SSH. Подобный подход при правильной настройке позволяет организовать безопасное сетевое соединение даже для клиентов, которые не могут использовать SSL.

Необходимо обеспечить наличие сервера SSH и доступ к нему от имени некоторой

учетной записи в системе, под управлением которой функционирует сервер PostgreSQL. Для организации защищенного туннеля на клиентской машине выполняется команда:

```
ssh -L 63333:localhost:5432 joe@foo.com
```

Число 63333 в аргументе -L задает номер порта на клиентской стороне туннеля (выбирается произвольно). Internet Assigned Numbers Authority (IANA) выделила для частного использования порты с номерами от 49152 до 65535. В случае установки соединения на порт 63333 соединение перенаправляется по защищенному каналу и устанавливается на стороне сервера на номер порта 5432 узла localhost. Для подключения к БД с использованием данного туннеля клиенту необходимо подключиться к порту 63333 на локальной машине:

```
psql -h localhost -p 63333 postgres
```

Для сервера СУБД на узле foo.com подключение будет выглядеть, как локальное подключение пользователя joe, и он будет использовать для аутентификации процедуру, которая была определена для этого пользователя и узла. Следует отметить, что соединение между сервером SSH и сервером PostgreSQL не защищается. Однако данное обстоятельство не приводит к увеличению риска для безопасности информации поскольку сервер SSH и сервер СУБД PostgreSQL функционируют в одной системе на одном узле.

Для установки туннельного соединения необходимо иметь права на установление соединения через ssh как пользователь joe@foo.com аналогично использованию ssh для установки терминальной сессии.

Возможно использовать переадресацию портов командой:

```
ssh -L 63333:foo.com:5432 joe@foo.com
```

В данном случае сервер СУБД будет видеть соединение входящим на интерфейс foo.com, который по умолчанию закрыт значением параметра listen_addresses = 'localhost'.

Для установки соединения с сервером СУБД с использованием узла аутентификации используется команда:

```
ssh -L 63333:db.foo.com:5432 joe@shell.foo.com
```

Следует отметить, что соединение shell.foo.com с db.foo.com не будет защищаться туннелем SSH. SSH предоставляет незначительные возможностей по настройке в случае различных сетевых ограничений. Дополнительная информации содержится в документации на SSH.

8. КОНФИГУРИРОВАНИЕ СЕРВЕРА

Поведение сервера СУБД в процессе функционирования определяется конфигурационными параметрами. Далее рассмотрен порядок настройки конфигурационных параметров.

8.1. Настройка параметров

8.1.1. Имена и значения параметров

Имена параметров используются без учета регистров. Параметр может принимать значение одного из пяти типов: `boolean`, `integer`, `floating point`, `string` или `enum`. Тип параметра определяет допустимый синтакс:

- *Логический* (`Boolean`): Значения типа могут быть записаны как `as`, `on`, `off`, `true`, `false`, `yes`, `no`, `1` или `0` (все значения к регистру нечувствительны);
- *Строковый* (`String`): Заключенное значение в одинарные кавычки, удвоив при этом любые одиночные кавычки в пределах значения. Однако, кавычки может быть опущены, если значение простое число или идентификатор;
- *Числовой* (`integer` и `float`): десятичная точка допускается только для параметров с плавающей точкой. Не используются разделители тысяч. Кавычки не требуются.
- *Числовой с суффиксом* (`numeric c uint`): Некоторые числовые параметры имеют суффикс, т.к. они описывают количество памяти или времени. Суффиксом может быть килобайт, блок (обычно восемь килобайт), милисекунда, секунда или минута. При неуказанном значение для такого параметра будет применяться значению по умолчанию, указанное в `pg_settings.unit`. Для удобства настройки могут быть предоставлены с суффиксом явно, например, `120 mc` для значения времени, и они будут преобразованы к той единице измерения, которая используется в этом параметре. Обратите внимание, что значение должно быть написано в виде строки (с кавычками), чтобы использовать суффикс. Имя суффикса чувствителен к регистру, в значении параметра не может быть пробелов между числовым значением и устройством;
  - Корректными суффиксами для единиц памяти являются: `kB` (килобайт), `MB` (мегабайт), `GB` (гигабайт), `TB` (терабайт). Множителем для памяти является `1024`, а не `1000`;
  - Корректными суффиксами для единиц времени являются: `ms` (миллисекунда), `s` (секунда), `min` (минута), `h` (час), `d` (день).
- *Перечисляемый* (`Enumerated`): Параметры перечисляемого типа представлены в виде строковых параметров, но разрешено использовать только одно из множества значений. Разрешенные значения для такого параметра могут быть найдены в `pg_–`

settings.enumvals. Значения перечисляемого типа нечувствительны к регистру.

8.1.2. Установка параметров с помощью конфигурационного файла

Одним из способов установки параметров является редактирование файла postgresql.conf, который обычно находится в каталоге данных, в который initdb записывает данный файл по умолчанию. Далее приведен пример файла postgresql.conf.

Пример

```
# This is a comment
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

Каждый параметр устанавливается в отдельной строке. Знак равенства между именем и значением является не обязательным. Пробелы (за исключением значения параметра в кавычках) и пустые строки игнорируются. Перед комментарием ставится знак #. Значения параметров, которые не являются простыми идентификаторами или числами, должны заключаться в одинарные кавычки. Для включения знака одинарной кавычки в значение параметра необходимо добавлять двойные кавычки (предпочтительный вариант) или обратный слэш.

Параметры, установленные в этом случае, являются параметрами по умолчанию для кластера. Если есть какие-либо настройки, установленные в активной сессией, то они переопределят параметры по умолчанию. Следующие разделы описывают способы изменеия администратором или пользователем эти значения по умолчанию.

Конфигурационный файл проверяется каждый раз, когда основной процесс сервера получает сигнал SIGHUP, который можно послать посредством выполнения команды pg_ctl reload или SQL функции pg_reload_conf(). Главный процесс сервера также передает данный сигнал всем запущенным процессам сервера для получения существующими сессиями нового значения. Альтернативой является отправка данного сигнала непосредственно одному из процессов сервера. Некоторые параметры могут быть изменены только при запуске сервера, изменения их значений в конфигурационном файле будут игнорироваться до перезапуска сервера.

В дополнение к postgresql.conf, каталог данных PostgreSQL содержит файл postgresql.auto.conf, который имеет тот же формат, что и postgresql.conf, но не должен быть отредактирован вручную. Этот файл содержит настройки, указанные с помощью команды ALTER SYSTEM. Этот файл автоматически считывается, как и postgresql.conf. Настройки в postgresql.auto.conf переопределяют настройки postgresql.conf.

Примечание. Возможность изменения конфигурационных параметров с помощью команды ALTER SYSTEM существует только в СУБД версии 9.6.

8.1.3. Измение параметров конфигурации с помощью SQL

Примечание. Возможность установки параметров конфигурации с помощью SQL-запросов ALTER DATABASE и ALTER ROLE есть только в версии СУБД 9.6.

PostgreSQL обеспечивает три SQL команды для установки параметров по умолчанию. Уже опомянутая команда ALTER SYSTEM обеспечивает SQL-доступ к средствам изменения глобальных умолчанию; эта возможность эквивалентна редактированию postgresql.conf. В дополнение к этому, существует две команды, которые позволяют установить параметры по умолчанию для базы данных и роли:

- ALTER DATABASE позволяет переопределить глобальные настройки в контексте базы данных;
- ALTER ROLE позволяет и глобальным, и настройкам для базы данных быть переопределнным пользовательскими значениями.

Значения, установленные с помощью ALTER DATABASE и ALTER ROLE применяются при старте новой сессии. Они переопределяют значения из конфигурационных файлов или командной строки и представляют собой значения по умолчанию для остальных сессий. Отметим, что некоторые настройки могут быть изменены только при перезапуске сервера и не могут быть переопределены одной из этих команды (или несколькими указанными ниже).

PostgreSQL обеспечивает клиента двумя дополнительными SQL командами для взаимодействия с локальными настройками сессии, как только он подключился к базе данных:

- Команда SHOW позволяет посмотреть текущее значение всех параметров. Соответствующей функцей является current_setting(setting_name text).;
- Команда SET позволяет проводить изменения параметров, которые могут быть изменены в сессии; это не влияет на другие сессии. Соответствующей функцией является set_config(setting_name, new_value, is_local).

В дополнение системное представление pg_settings могут быть использовано для просмотра и изменения локальных настроек сессии:

- Запрос на чтение этого представления эквивалентно команде SHOW ALL, но он возвратит больше информации. Оно также более расширяемо, поэтому можно указывать условия для фильтрации или использовать для объединения с другими отношениями.
- Использование UPDATE на представлении и указание в нем столбец setting аналогично использованию команды SET. Например, эквивалентом

```
SET configuration_parameter TO DEFAULT;
```

является:

```
  UPDATE pg_settings SET setting = reset_val
 WHERE name = 'configuration_parameter';
```

8.1.4. Установка параметров конфигурации с помощью командной оболочки

Кроме того для установки конфигурационных параметров можно передать их значения как опции команды `postgres` в командной строке:

```
postgres -c log_connections=yes -c log_destination='syslog'
```

Значения, указанные в командной строке, переопределяют значения, указанные в файле `postgresql.conf`. Следует отметить, что хотя задание значений параметров с помощью опций в командной строке является удобным, использование данного способа не позволит изменять значения «на лету», редактируя файл `postgresql.conf`, и следовательно может привести к снижению гибкости администрирования.

Параметры командной строки могут быть использованы при установке значений параметров для отдельной сессии с помощью переменной окружения `PGOPTIONS` на стороне клиента:

```
env PGOPTIONS='-c geqo=off' psql
```

Другие клиенты и библиотеки могут поставлять механизмы установки через командную оболочку или иные механизмы, которые позволяют пользователям изменять настройки сессии без использовования SQL команд.

8.1.5. Управление содержимым конфигурационного файла

PostgreSQL обеспечивает несколько возможностей для разделения конфигурационного файла `postgresql.conf` на файлы. Эти возможности обычно полезны при работе с несколькими связанными, но неодинаковыми конфигурациями серверов.

Кроме значений параметров, в файле `postgresql.conf` содержатся инструкции включения, указывающие файл, который необходимо читать и обрабатывать для добавления его содержимого в конфигурационный файл в указанном месте. Инструкции включения выглядят следующим образом:

```
include 'filename'
```

В случае когда не используется абсолютный путь, считается, что файл лежит в том же каталоге, что и этот конфигурационный файл. Включения могут быть вложенными.

Существует директива `include_if_exists`, которая действует аналогично `include` за исключением поведения в случае отсутствия указанного файла. Обычно `include` рассматривает это как ошибку, тогда как `include_if_exists` в том же случае не прерывает обработку конфигурационного файла, а только выводит предупреждающее

сообщение.

postgresql.conf может содержать и директиву include_dir, указывающую целый каталог конфигурационных файлов для включения:

```
include_dir 'directory'
```

В случае когда не используется абсолютный путь, применяются те же правила, что и для отдельного файла: путь рассматривается относительно каталога основного конфигурационного файла. Внутри указанного каталога включаются только обычные файлы с суффиксом .conf. Файлы, имена которых начинаются с точки (.), также включаются для избежания ошибок связанных с сокрытием этих файлов. Файлы рассматриваются в порядке их имен, отсортированных в по правилам локали C, т.е. цифры располагаются ранее букв, а заглавные буквы ранее срочных.

Включение файлов и каталогов может быть использовано для логического разделения конфигурации на части, вместо одного громоздкого файла postgresql.conf. Рассмотрим компанию, имеющую два сервера, отличающихся количеством памяти. Оба могут разделять общие части конфигурации, таких как настройки протоколирования. Тогда как параметры, относящиеся к использованию памяти, могут различаться. Также могут присутствовать и другие отличия в настройках, специфичные для каждого сервера. Одним и способов решения подобной ситуации является разделение изменений на три файла, которые могут быть включены в конец файла postgresql.conf.

```
include 'shared.conf'
include 'memory.conf'
include 'server.conf'
```

Обе системы могут иметь одинаковый shared.conf. Каждый сервер с одинаковым количеством памяти может иметь одинаковый memory.conf; может быть задан один для серверов с 8Гб оперативной памяти, а другой для серверов с 16Гб. server.conf может содержать специфичные для каждого сервера настройки.

Другим подходом является создание конфигурационного каталога, содержащего настройки. Например включение каталога conf.d в файл postgresql.conf может быть выполнено следующим образом:

```
include_dir 'conf.d'
```

При этом файлы в каталоге conf.d могут быть поименованы так:

```
00shared.conf
01memory.conf
02server.conf
```

Это позволяет прозрачным образом определить порядок загрузки этих файлов. Порядок важен поскольку будет использовано только последнее обнаруженное при чтении конфигурации значение. Некоторые настройки в conf.d/02server.conf могут перекрыть

значения в conf.d/01memory.conf.

Может быть использовано и боле прозрачное именование этих файлов:

```
00shared.conf
01memory-8GB.conf
02server-foo.conf
```

Такой способ сортировка позволяет использовать уникальные для каждого варианта конфигурационного файла. Это позволяет устранить неоднозначность в случае когда, несколько серверов разделяют конфигурационные файлы, расположенные в одном месте, например при использовании версионного хранилища. (Использование версионного хранилища для конфигурационных файлов также является хорошей практикой.)

8.2. Расположение файлов

В дополнение к файлу postgresql.conf в PostgreSQL используется еще два редактируемых вручную конфигурационных файла, которые контролируют аутентификацию клиента (смотри 9). По умолчанию, все эти три файла находятся в директории данных кластера БД. Параметры, описываемые в данном разделе, позволяют для упрощения администрирования и резервного копирования конфигурационных файлов поместить названные файлы в любое другое место.

- data_directory (string) определяет директорию для хранения данных. Значение этого параметра может быть задано только при запуске сервера.
- config_file (string) определяет основной конфигурационный файл сервера (обычно он называется postgresql.conf). Значение этого параметра может быть задано только в командной строке postgres.
- hba_file (string) определяет конфигурационный файл для аутентификации по узлам (обычно он называется pg_hba.conf). Значение этого параметра может быть задано только при запуске сервера.
- ident_file (string) определяет конфигурационный файл для аутентификации по методу ident (обычно он называется pg_ident.conf). Значение этого параметра может быть задано только при запуске сервера.
- external_pid_file (string) определяет имя дополнительного файла с идентификатором процесса, который сервер создает для использования программами администрирования сервера. Значение этого параметра может быть задано только при запуске сервера.

При установке по умолчанию значения перечисленных выше параметров не задаются явным образом. Вместо этого директория данных определяется ключом -D в командной строке или переменной окружения PGDATA, а все конфигурационные файлы находятся в директории данных.

Для размещения конфигурационных файлов в другой директории, опция -D команды postgres или переменная окружения PGDATA должны указывать на директорию для конфигурационных файлов, а параметр data_directory, прописанный в файле postgresql.conf (или в командной строке), должен показывать, где на самом деле находится директория данных. Следует отметить, что параметр data_directory переопределяет значения -D и PGDATA при указании директории данных и не переопределяет значения -D и PGDATA при указании расположения конфигурационных файлов.

Существует возможность определять имена конфигурационных файлов и их местоположение по отдельности, используя параметры config_file, hba_file и/или ident_file. Значение config_file можно задать только в командной строке postgres, остальные значения параметров можно указать в главном конфигурационном файле. Определение названных трех параметров и data_directory явным образом избавляет от необходимости использовать -D или PGDATA.

При использовании относительного пути для задания значений этих параметров, путь будет отсчитываться от директории, в которой запущен postgres.

8.3. Соединения и аутентификация

8.3.1. Параметры соединения

- listen_addresses (string) определяет TCP/IP адреса, по которым сервер должен ожидать соединения от клиентских приложений. Значение формируется в виде перечня разделенных запятой имен узлов и/или числовых IP-адресов. Специальный знак * соответствует всем доступным IP-адресам. Значение 0.0.0.0 позволяет задать все IPv4 адреса, а :: все IPv6 адреса. Если список пуст, сервер не слушает ни один IP-интерфейс. В этом случае установка соединения с сервером возможна только с использованием доменных сокетов Unix. По умолчанию значением параметра является localhost, которое позволяет создавать только локальные loopback-соединения. Настройка аутентификации клиента (см. 9) позволяет гибко настроить, кто может устанавливать соединения, тогда как listen_addresses какой именно интерфейс принимает попытки соединения, что может помочь предотвратить вредоносные попытки соединения по незащищенному интерфейсу. Значение параметра может быть задано только при запуске сервера.
- port (integer) определяет TCP-порт, на котором сервер должен ожидать соединения от клиентских приложений (по умолчанию 5432). Следует отметить, что для всех IP-адресов, указанных в listen_addresses (string), используется один и тот же порт. Значение параметра может быть задано только при запуске сервера.
- max_connections (integer) определяет максимальное число одновременных

соединений с сервером СУБД. По умолчанию обычно 100 соединений, но значение может быть уменьшено если настройки ядра не поддерживают данное значение (что определяется при выполнении initdb). Значение параметра может быть задано только при запуске сервера. Для резервного сервера должно быть указано одинаковое или большее чем на основным сервере значение этого параметра, в противном случае он может не принимать входящие соединения.

- superuser_reserved_connections (integer) определяет число соединений, зарезервированных для подключения суперпользователей PostgreSQL. В случае когда число активных соединений становится равным max_connections минус superuser_reserved_connections, новые соединения устанавливаются только с суперпользователями. По умолчанию выделено три соединения. Значение параметра должно быть меньше, чем значение параметра max_connections. Значение параметра можно задать только при запуске сервера.

- unix_socket_directory (string) определяет директорию для доменного сокета Unix, на котором сервер ожидает соединения с клиентскими приложениями. Для ожидания соединения может быть использовано несколько сокетов, расположенных в разных директориях, разделенных запятыми. Пробельные символы игнорируются, для их использования имена директорий должны быть заключены в двойные кавычки. Если список пуст, сервер не ожидает соединений на доменных сокетах Unix, в этом случае для установки соединений могут быть использованы только TCP/IP сокеты. По умолчанию используется значение /tmp, которое может быть изменено при сборке. Значение параметра задается только при запуске сервера. Дополнительно к самому файлу сокета, именуемому .s.PGSQL.nnnn, где nnnn является номером порта, для каждого файла сокета из unix_socket_directories создается файл блокировок .s.PGSQL.nnnn.lock. Указанные файлы не должны удаляться вручную.

- unix_socket_group (string) определяет группу, владеющую доменным сокетом Unix. Владельцем сокета всегда является пользователь, запускающий сервер. Используется в комбинации с параметром unix_socket_permissions, как дополнительный механизм контроля доступа для соединений в домене Unix. По умолчанию значение параметра пустая строка, что означает группу по умолчанию для текущего пользователя. Значение параметра можно задать только при запуске сервера.

- unix_socket_permissions (integer) определяет права доступа для доменного сокета Unix. Доменные сокеты Unix обычно используют права доступа файловой системы Unix. Значение параметра должно быть задано числом в форме, принимаемой системными вызовами chmod и umask. Для использования стандартного

восьмеричного формата число должно начинаться с нуля. По умолчанию значение параметра 0777, которое определяет, что соединение разрешено для всех. Разумной альтернативой является задание значения параметра 0770 (только пользователь и группа, смотри также unix_socket_group) или 0700 (только пользователь). Следует отметить, что для доменного сокета Unix имеет значение только разрешение на запись, и следовательно нет необходимости устанавливать или удалять разрешение на чтение или выполнение. Данный механизм контроля доступа не зависит от механизма, описанного в 9. Значение параметра может быть задано только при запуске сервера. Этот параметр не имеет смысла на системах, таких как Solaris, так как в них игнорируются полностью права на сокет. Там можно достичь аналогичного эффекта, указав unix_socket_directories права на просмор директории ограниченному какой-либо группе. Этот параметр также не имеет смысла на Windows, которая не имеет Unix-сокетов.

- bonjour (boolean) разрешает экземпляру сервера представляться с помощью Bonjour. По умолчанию выключено. Значение параметра может быть задано только при запуске сервера.

- bonjour_name (string) определяет имя для широковещательной трансляции приветствия. Пустая строка (значение по умолчанию) соответствует имени компьютера. Параметр игнорируется в случае когда сервер скомпилирован без поддержки трансляции приветствия. Значение параметра может быть задано только при запуске сервера.

- tcp_keepalives_idle (integer) определяет время бездействия (в секундах), после которого следует выполнить проверку активности клиента по протоколу TCP. Значение 0 определяет использование настроек по умолчанию операционной системы. В случае когда опция TCP_KEEPIDLE не поддерживается системой, значение параметра должно быть 0. Параметр игнорируется при соединениях через доменные сокеты Unix.

- tcp_keepalives_interval (integer) определяет в секундах для систем с поддержкой опции сокета TCP_KEEPINTVL время ожидания ответа на пакет, подтверждающий активность, перед его повторной отправкой. Значение 0 определяет использование настроек по умолчанию операционной системы. В случае когда опция TCP_KEEPINTVL не поддерживается системой, значение параметра должно быть 0. Параметр игнорируется при соединениях через доменный сокет Unix.

- tcp_keepalives_count (integer) определяет для систем с поддержкой опции сокета TCP_KEEPCNT число пакетов, подтверждающих активность, может быть потеряно, прежде чем соединение будет считаться разорванным. Значение 0 означает,

что нужно использовать системные настройки по умолчанию. В случае когда опция TCP_KEEPCNT не поддерживается системой, значение параметра должно быть 0. Параметр игнорируется при соединениях через доменный сокет Unix.

8.3.2. Безопасность и аутентификация

- authentication_timeout (integer) определяет в секундах максимальное время выполнения аутентификации клиента. В случае когда потенциальный клиент не выполняет протокол аутентификации в установленное время, сервер разрывает соединение, что предотвращает использование ресурсов соединений зависшими клиентами. Значение по умолчанию одна минута (1m). Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.
- ssl (boolean) разрешает соединения по протоколу SSL (см. 7.9). Значение по умолчанию off. Значение параметра может быть задано только при запуске сервера. Использование протокола SSL возможно только для TCP/IP-соединений.
- ssl_ca_file (string) задает файл, содержащий SSL сертификат удостоверяющего центра сервера (CA). По умолчанию значение не задано, что означает отсутствие загрузки CA файла и проверки сертификатов клиентов. (В предыдущих версиях имя было жестко задано (root.crt). Относительные пути считаются от каталога данных. Значение параметра может быть задано только при запуске сервера.
- ssl_cert_file (string) задает файл, содержащий SSL сертификат сервера. По умолчанию значение равно server.crt. Относительные пути считаются от каталога данных. Значение параметра может быть задано только при запуске сервера.
- ssl_crl_file (string) задает файл, содержащий список отозванных сертификатов (CRL). По умолчанию значение не задано, что означает отсутствие загрузки CLR файла. (В предыдущих версиях имя было жестко задано (root.crl). Относительные пути считаются от каталога данных. Значение параметра может быть задано только при запуске сервера.
- ssl_cert_file (string) задает файл, содержащий приватный ключ сервера. По умолчанию значение равно server.key. Относительные пути считаются от каталога данных. Значение параметра может быть задано только при запуске сервера.
- ssl_renegotiation_limit (integer) определяет количество данных, передаваемых через защищенное SSL соединение, до смены ключей сессии. Смена ключей уменьшает шансы нарушителя произвести криптоанализ по большому количеству передаваемых данных, но может снижать производительность. Для значения

используется сумма отправленных и полученных данных. Если параметру задано значение 0 — смены ключей не производится. Значение по умолчанию — 512МБ.

- `ssl_ciphers (string)` определяет список маскирующих преобразований, используемых для создания защищенных соединений по протоколу SSL. Перечень поддерживаемых маскирующих преобразований приведен в руководстве по `openssl`. Значением по умолчанию является `HIGH:MEDIUM:+3DES:!aNULL`. Это обычно требуется, когда необходимы специфические меры безопасности. Объяснение значения по умолчанию:
  - `HIGH` – алгоритмы шифрования, использующие шифры `HIGH` группы (например, AES, Camellia, 3DES);
  - `MEDIUM` – алгоритмы шифрования, использующие шифры `MEDIUM` группы (например, RC4, SEED);
  - `+3DES` – порядок OpenSSL умолчанию для `HIGH` проблематичен, потому что это 3DES выше, чем AES128. Это неправильно, потому что 3DES предлагает меньше защиты, чем AES128, и это также гораздо медленнее. `+3DES` сортирует его после всех других `HIGH` и `MEDIUM` шифров;
  - `!aNULL` – отключает анонимный алгоритмы щифрования, которые не производят аутентификацию. Такие алгоритмы шифрования уязвимы для MITM-атак (man in the middle) и, следовательно, не должны использоваться.

Доступные шифры могут варьироваться в зависимости от версии OpenSSL на вашей платформе. Используйте команду OpenSSL `openssl ciphers -v 'HIGH:MEDIUM:+3DES:!aNULL'`, чтобы увидеть алгоритмы шифрования, доступной для текущей установленной версии OpenSSL. Обратите внимание, что этот список фильтруется во время выполнения в зависимости от типа сервера ключей.

- `ssl_prefer_server_ciphers (bool)` определяет, может ли сервер использовать SSL предпочтения сервера, а не клиента. По умолчанию `true`.
- `ssl_ecdh_curve (string)` определяет имя кривой для использования в обмене ключами в алгоритме ECDH. Эта кривая должна поддерживаться всеми клиентами. Имя не должно быть таким же, как используемая сервером ключ эллептической кривой. По умолчанию `prime256v1`. Имена OpenSSL общих кривых: prime256v1 (NIST P-256), secp384r1 (NIST P-384), secp521r1 (NIST P-521). Полный список доступных кривых может быть получен с помощью команды `openssl ecparam -list_curves`. Не все из них поддерживаются в протоколе TLS.
- `password_encryption (boolean)` определяет необходимость маскирующего преобразования пароля указанного в командах `CREATE USER` или `ALTER USER` без

использования ENCRYPTED или UNENCRYPTED. Значение по умолчанию on (пароль маскируется).

- krb_server_keyfile (string) определяет расположение файла с ключом Kerberos для сервера (смотри 9.3.3 и 9.3.4). Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

- krb_srvname (string) определяет имя сервиса Kerberos (9.3.4). Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

- krb_caseins_users (boolean) определяет необходимость учета регистра букв при чтении имен пользователей Kerberos и GSSAPI. Значение по умолчанию off (регистр учитывается). Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

- db_user_namespace (boolean) разрешает задание индивидуального набора пользователей для БД. Значение по умолчанию off (запрещено). Значение параметра может быть задано в файле postgresql.conf или в командной строке сервера. Если значение параметра on (разрешено), имена новых пользователей необходимо вводить в формате username@dbname (имя_пользователя@имя_БД). При создании соединения клиентом передается имя пользователя username, к которому добавляется символ @ и имя БД. Сформированное специфическое для БД обрабатывается сервером. Следует отметить, что для использования имени пользователя, содержащего символ @, в SQL-окружении необходимо квотировать имя пользователя. Значение параметра on позволяет создавать обычных глобальных пользователей. Просто необходимо добавлять символ @ при определении имени пользователя на стороне клиента. При обработке подобного имени пользователя сервер будет убирать символ @. Параметр db_user_namespace делает различными представления имени пользователя для сервера и для клиента. При аутентификации всегда проверяется вариант сервера, а не клиента, следовательно необходимо подбирать соответственные методы аутентификации. Так как md5 использует имя пользователя как параметр для маскирующего преобразования как для клиента, так и для сервера, то аутентификация md5 не может использоваться при значении on параметра db_user_namespace. Параметр используется временно и впоследствии будет удален.

8.4. Потребление ресурсов

8.4.1. Память

- shared_buffers (integer) определяет объем памяти, используемой сервером СУБД для буферов разделяемой памяти. Значение по умолчанию 128Мб, но может быть уменьшено если настройки ядра не поддерживают данное значение (что определяется при выполнении initdb). Значение параметра должно быть не менее 128Кб. Изменение значения BLCKSZ приводит к изменению данного минимума. Для обеспечения хорошей производительности необходимо установить значение намного выше минимального. Значение параметра может быть задано только при запуске сервера. Для выделенных серверов баз данных с объемом оперативной памяти более 1 ГБ, рекомендуемое значение для shared_buffers может начинаться от 25% памяти системы. Существуют некоторые рабочие нагрузки, при которых большие значения установки для shared_buffers являются эффективными, но поскольку PostgreSQL использует средства кеширования ОС, использование для этого параметра значения более 40% оперативной памяти может быть хуже, чем меньшее. Большие значения shared_buffers обычно требуют соответствующего увеличения параметра checkpoint_segments, для того, чтобы растянуть процесс фиксации большого количества новых или измененных данных в течение более длительного периода времени. Для систем с памятью менее 1 ГБ подходят малые значения процентов использования оперативной памяти, для оставления ее ОС.
- huge_pages (enum) включает/отключает использования больших страниц. Корректными значениями являются: try, on и off. На текущий момент поддержка больших страниц реализована только в Linux. Значение параметра будет игнорироваться на других системах. Использование больших страниц приведет к меньшему числу страниц страниц и уменьшение процессорного времени на управление памятью, а также повышение производительности. Для более подробной информации см 7.4.4. Если этот параметр установлен в try, то PostgreSQL будет пытаться использовать механизм больших страниц, возвращаясь к нормальным при ошибке. Если параметр установлен в on, то возможен отказ запуска PostgreSQL, если в системе недостаточно больших страниц. При off использование больших страниц отключено.

Примечание. Данный параметр может быть установлен только в СУБД версии 9.6.

- temp_buffers (integer) определяет максимальное число временных буферов, используемых каждой сессией БД только для доступа к временным таблицам. Значение по умолчанию 8 МБ. Значение параметра может быть изменено в ходе сессии, но

только до момента первого использования временных таблиц. Последующие попытки изменить значение в данной сессии не сработают. Сессия по необходимости будет занимать временные буферы до достижения предела, заданного в temp_buffers. Установка большого значения для сессии, в которой не требуется много буферов, то будет стоить 64 байта на каждую единицу увеличения temp_buffers (место, необходимое для дескриптора буфера). На каждый используемый буфер требуется дополнительно 8192 байт (или BLCKSZ байт).

- max_prepared_transactions (integer) определяет максимальное число транзакций, одновременно находящихся в состоянии «готово» (смотри команду PREPARE TRANSACTION). Значение 0 (по умолчанию) отключает возможность подготовки транзакций. Значение параметра может быть задано только при запуске сервера. В случае когда подготовленные транзакции не используются значение параметра можно установить равным 0. В противном случае для исключения ошибок на этапе подготовки рекомендуется установить значение параметра max_prepared_transactions как минимум равным значению параметра max_connections. Для резервного сервера должно быть указано одинаковое или большее чем на основным сервере значение этого параметра, в противном случае он может не принимать входящие соединения.

- work_mem (integer) определяет объем памяти, используемой для внутренних операций сортировки и хэш-таблиц до вывода во временные файлы на диске. Значение по умолчанию 4 МБ. Следует отметить, что в сложном запросе параллельно может быть запущено несколько операций сортировки и хэширования, каждая из которых до вывода во временные файлы может использовать объем памяти равный значению параметра. Кроме того, операции сортировки и хэширования могут одновременно выполняться в нескольких сессиях. Таким образом, при установке значения параметра следует учесть, что объем реально используемой памяти может оказаться в несколько раз больше, чем определено значением параметра work_mem. Операции сортировки используются в командах ORDER BY, DISTINCT и операций соединения методом слияния (merge joins). Хэш-таблицы используются при операциях соединения методом хэширования (hash joins), операциях агрегирования, основанных на хэшировании и обработке IN подзапросов на основе хэширования.

- autovacuum_work_mem (integer) определяет максимальный размер памяти, который может быть использован каждым процессом очистки (autovacuum). Если выставлен в -1, то значение maintenance_work_mem не будет использовано. Настройка не имеет эффекта при поведении VACUUM, запущенного в другом контекстах. Примечание. Данный параметр используется только в СУБД версии 9.6.

- maintenance_work_mem (integer) определяет максимальный объем памяти, используемой при выполнении служебных операций, таких как VACUUM, CREATE INDEX и ALTER TABLE ADD FOREIGN KEY. Значение по умолчанию 16 МБ. Поскольку в сессии возможно одновременное выполнение только одной подобной операции и сервер СУБД обычно не осуществляет конкурентное выполнение данных операций, можно установить значение параметра намного выше, чем значение параметра work_mem. Большие значения улучшат производительность при проведении операции VACUUM и восстановлении БД из дампа. Следует отметить, что значение по умолчанию должно быть не слишком большим, поскольку при выполнении автовакуума выделенная память может не освобождаться пока число операций автовакуума не достигнет значения, заданного параметром autovacuum_max_workers. Это может быть полезно для контролирования с помощью установки autovacuum_work_mem.
- max_stack_depth (integer) определяет максимальную безопасную глубину стека для сервера. Идеальным значением параметра является реальный предельный размер стека, заданный ядром ОС, уменьшенный на допуск безопасности примерно в 1 МБ. Допуск безопасности необходим, так как глубина стека проверяется не при каждом действии сервера, а только в случае потенциально рекурсивных действий, таких как оценка выражения. Значение по умолчанию 2 МБ достаточно мало и не должно приводить к сбоям. Однако, значение может оказаться слишком малым для выполнения сложных функций. Значение параметра могут изменять только суперпользователи. Установка для параметра max_stack_depth значения, превышающего ограничение ядра ОС, может привести при выполнении рекурсивной функции к непредвиденному завершению обслуживающего процесса сервера. На платформах, в которых PostgreSQL может определить ограничение ядра, возможно установить небезопасное значение параметра. Однако не все платформы предоставляют такую информацию, следовательно рекомендуется осторожно выбирать значение параметра.
- dynamic_shared_memory_type (enum) определяет реализацию динамически разделяемой памяти, которая используется сервером. Возможные значения: posix (для POSIX разделяемой памяти, выделяемой с помощью shm_open), sysv (для System V разделяемой памяти, выделяемой с помощью shmget), windows (для разделяемой памяти Windows), nmap (для симуляции разделяемой памяти, использующей файлы в директории data) и none (для отключения использования). Не все значения поддерживаются на платформах. Первое поддерживаемое значение является значением по умолчанию на платформе. Для использования опции nmap,

которая не является настройкой по умолчанию на любой платформе, как правило, не рекомендуется, поскольку операционная система может написать измененных страниц на диск несколько раз, увеличивая нагрузки на систему ввода-вывода; Однако, это может быть полезно для отладки, когда каталог pg_dynshmem хранится на RAM диске, или когда другие объекты разделяемой памяти не доступны.

8.4.2. Диск

- temp_file_limit (integer) задает максимальное значение дискового пространства, которое может быть использовано в одной сессии для временных файлов, таких как временные файлы сортировок или файлов поддержки курсоров. Транзакции, пытающиеся превысить указанный предел, будут прерваны. Значение задается в килобайтах, при значении -1 (по умолчанию) ограничение отсутствует. Только суперпользователь может изменять этот параметр. Этот параметр ограничивает общее пространство, используемое в любой момент всеми временными файлами, используемых конкретной сессией PostgreSQL. Следует отметить, что место на диске, используемое для специальных временных таблиц, в отличие от временных файлов, используемых при выполнении запроса, не учитываются в этом ограничении.

8.4.3. Использование ресурсов ядра

- max_files_per_process (integer) определяет максимальное допустимое для каждого подпроцесса сервера число одновременно открытых файлов. Значение по умолчанию 1000. В случае когда ядро устанавливает безопасный предел числа открытых файлов для каждого процесса необходимость использования параметра отсутствует. На некоторых платформах (особенно в BSD-системах) ядро разрешает отдельным процессам открывать намного больше файлов, чем предел поддерживаемый системой, при этом сразу несколько процессов могут пытаться открыть файлы в количестве превышающем возможности системы. Сообщение «Too many open files» (Открыто слишком много файлов) указывает на необходимость уменьшения значения параметра. Значение параметра может быть задано только при запуске сервера.

8.4.4. Задержка операции вакууминга на основе оценки стоимости

Во время выполнения команд VACUUM и ANALYZE система использует внутренний счетчик для отслеживания оценочной стоимости различных выполняемых операций ввода-вывода. По достижении накопленной стоимостью предела, определенного значением параметра vacuum_cost_limit, процесс, выполняющий операцию «засыпает» на некоторое время, определенное значением параметра vacuum_cost_delay. Затем счетчик обнуляется, и выполнение команды продолжается.

Рассмотренная особенность позволяет администраторам понизить влияние на операций ввода-вывода команд VACUUM и ANALYZE на прочие действия СУБД. Существует множество ситуаций, не требующих быстрого выполнения команд, подобных VACUUM и ANALYZE. Однако обычно очень важно, чтобы влияние выполнения указанных команд на возможность системы выполнять другие операции с БД не было значительным. Задержка операции вакууминга на основе оценки стоимости предоставляет администраторам возможность решения рассмотренной проблемы. По умолчанию эта возможность отключена. Для ее использования необходимо присвоить параметру vacuum_cost_delay значение, отличное от нуля.

- vacuum_cost_delay (integer) определяет в миллисекундах время, на которое «засыпает» процесс по достижении предела стоимости. Значение по умолчанию 0 означает, что возможность задержки операции вакууминга на основе оценки стоимости отключена. Следует отметить, что во многих системах эффективным значением задержки является 10. Значение vacuum_cost_delay округляется в сторону ближайшего большего числа кратного 10. При использовании вакууминга на основе оценки стоимости подходящие значения vacuum_cost_delay обычно малы, возможно 10 или 20 миллисекунд. Управление потреблением ресурсов осуществляется изменением других относящихся к вакуумингу параметров.
- vacuum_cost_page_hit (integer) определяет оценочную стоимость операции вакууминга для одного буфера, находящегося в кэше буферов разделяемой памяти (стоимость блокировки буферной области, поиска по разделяемой хэш-таблице и сканирования содержимого найденной страницы). Значение по умолчанию 1.
- vacuum_cost_page_miss (integer) определяет оценочную стоимость вакууминга буфера, который должен быть прочитан с диска (стоимость попытки заблокировать буферную область, произвести поиск по разделяемой хэш-таблице, считать нужный блок с диска и просканировать его содержимое). Значение по умолчанию 10.
- vacuum_cost_page_dirty (integer) определяет оценочную стоимость заполнения при модификации операцией вакууминга пустого блока (стоимость дополнительных операций ввода-вывода для переноса содержимого блока обратно на диск). Значение по умолчанию 20.
- vacuum_cost_limit (integer) определяет предел суммарной стоимости, превышение которого приводит к «засыпанию» процесса вакууминга. Значение по умолчанию 200.

Примечание. Следует отметить, что существуют определенные проводящие к критическим блокировкам действия, которые должны быть завершены как можно

скорее. Задержки операции вакууминга на основе оценки стоимости во время таких действий не производятся. Следовательно, суммарная стоимость может оказаться значительно выше установленного предела. Во избежание необоснованно длинных задержек в подобных случаях, реальное время задержки вычисляется следующим образом: `vacuum_cost_delay * accumulated_balance / vacuum_cost_limit`. При этом максимальное значение ограничено `vacuum_cost_delay * 4`.

8.4.5. Процесс для записи в фоновом режиме

Существует отдельный процесс сервера `background writer` (процесс для записи в фоновом режиме), функцией которого является запись содержимого разделяемых буферов. Данный процесс позволяет процессам сервера, обрабатывающим запросы пользователей, не ждать (или ждать редко) возможности для записи. Однако, общее количество операций ввода-вывода за единицу времени увеличивается поскольку повторно обновленная страница могла иным способом быть записана только один раз за контрольный интервал, а `background writer` осуществляет запись страницы несколько раз за тот же интервал. Далее описаны параметры для настройки поведения процесс для записи в фоновом режиме для локальных нужд.

- `bgwriter_delay (integer)` определяет в миллисекундах время задержки между рабочими циклами процесса `background writer`. В каждом цикле `background writer` осуществляет запись некоторого числа, задаваемого прочими параметрами, заполненных буферов. Далее процесс «засыпает» на время, определяемое значением параметра `bgwriter_delay`, и затем следует новый рабочий цикл. Значение по умолчанию `200` миллисекунд (`200ms`). Следует отметить, что для многих систем эффективное разрешение задержки составляет 10 миллисекунд. Значение округляется в сторону ближайшего большего числа кратного `10`. Значение параметра может быть задано только в файле `postgresql.conf` или в командной строке сервера.
- `bgwriter_lru_maxpages (integer)` определяет предельное число буферов, записываемых процессом `background writer` в одном цикле. При установке нулевого значения фоновая запись будет отключена за исключением действий, связанных с контрольными точками. Значение по умолчанию `100`. Значение параметра может быть задано только в файле `postgresql.conf` или в командной строке сервера.
- `bgwriter_lru_multiplier (floating point)` определяет оценку числа новых буферов, которые понадобятся процессам сервера в последующих циклах. Количество обновленных буферов, записываемых в каждом цикле зависит от данной оценки. Оценка необходимого процессам сервера в следующем цикле количества

новых буферов вычисляется как среднее для последних циклов число использованных буферов умноженное на значение параметра bgwriter_lru_multiplier. Обновленные буферы записываются до тех пор пока не будет достигнуто необходимое количество чистых доступных для повторного использования буферов. При этом за один цикл не может быть записано больше буферов, чем определено значением параметра bgwriter_lru_maxpages. Значение 1.0 означает, что нужно записывать ровно то количество буферов, которое было вычислено по приведенной выше формуле. Большие значения позволяют создать запас буферов на случай резкого повышения требований процессов сервера. Указание меньшего значения означает, что некоторые записи будут выполнены процессами сервера. По умолчанию установлено значение 2.0. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

Указание меньших значений для bgwriter_lru_maxpages и bgwriter_lru_multiplier сокращает число дополнительных операций ввода-вывода, вызванных процессом background writer, однако при этом повышается вероятность того, что процессы сервера будут выполнять запись самостоятельно, задерживая выполнение интерактивных запросов.

8.4.6. Асинхронное поведение

- effective_io_concurrency (integer) определяет число одновременно выполняемых операций чтения-записи на диск ожидаемых сервером СУБД. Увеличение значения повышает число операций ввода-вывода, которое любая сессия PostgreSQL может попытаться инициировать параллельно. Допустимые значения находятся в диапазоне от 1 до 1000. Значение 0 запрещает выполнение асинхронных операций ввода-вывода. Для задания начального значения параметра можно использовать число отдельных устройств, входящих в массивы RAID 0 или RAID 1, используемых БД. Устройства для контрольных сумм не должны не учитываться. Однако, если СУБД часто занята множеством запросов, получаемых от различных конкурирующих сессий, для поддержания загрузки дискового массива могут быть использованы более низкие значения параметра. Значения превышающие необходимое для поддержания загрузки дискового массива приводят увеличению загрузки процессора. Для более экзотических систем, таких как хранилища, основанные на памяти или RAID-массивы, ограниченные пропускной способностью шины, возможно, корректным значением может являться число доступных путей ввода-вывода. Оптимальное значение определяется экспериментальным путем. Асинхронный ввода-вывод зависит от эффективности функции posix_fadvise, которая отсутствует в некоторых операционных системах. В подобном случае любое

значение параметра, отличное от 0, приводит к возникновению ошибки. В некоторых операционных системах (например, Solaris) названная функция присутствует, но в действительности ничего не делает.

- max_worker_processes (integer) устанавливает маскимальное число фоновых процессов, которые система может использовать. Данный параметр может быть указан до запуска сервера. Когда запускается ведомый сервер, вы должны установить этот параметр в такое же или большее значение данного параметра ведущего сервера. Иначе запросы не будут доходить до ведомого сервера.

Примечание. Данный параметр используется только в СУБД версии 9.6.

8.5. Журнал упреждающей регистрации записываемых данных (WAL)

8.5.1. Настройки

- wal_level (enum) определяет какая информация записывается в журнал (WAL). По умолчанию задано значение minimal, при котором записывается только информация, необходимая для восстановления после сбоя или внезапной остановки сервера. archive добавляет информацию для архивирования WAL журнала, а hot_standby добавляет еще информацию для возможности исполнения запросов на чтение на резервном сервере, наконец, logical добавляет необходимую информацию для логического декодирования. Каждый уровень включает в себя информацию предыдущих. Значение параметра может быть задано только при старте сервера.

Примечание. Значение данного параметра logical используется только в СУБД версии 9.6.

ВНИМАНИЕ! Использование уровня logical не рекомендуется, т.к. в данном режиме изменения отправляются в виде высокоуровневых запросов, что может стать причиной утечки информации.

На уровне minimal некоторые массовые операции могут не фиксироваться в журнале для ускорения их выполнения. Операции, которые могут быть затронуты этим:

CREATE TABLE AS

CREATE INDEX

CLUSTER

COPY в таблицы, созданные или очищенные в той же транзакции

Но в этом случае в журнале не содержится полной информации для воссоздания данных из резервной копии и журнала WAL, так что для архивирования WAL (archive_mode) должны использоваться уровни archive или hot_standby и потоковая репликация.

На уровне hot_standby дополнительно к информации archive записывается информация, необходимая для воссоздания статуса запущенных транзакций. Для возможности исполнения на резервном сервере запросов ни чтение, параметр wal_level должен быть установлен в значение hot_standby как на основном сервере, так и на резервном.

На уровене logical дополнительно к информации на уровне hot_standby записывается инофрмация, необходимая для извлечения логических изменений из WAL. Использование уровня logical увеличивает размер WAL частично за счет таблиц, сконфирурированных для REPLICA IDENTITY FULL и запросов UPDATE и DELETE, которые исполнялись.

- fsync (boolean) определяет будет ли сервер СУБД PostgreSQL пытаться обеспечить гарантию физической записи всех обновлений на диск, выполняя системные вызовы fsync(), или различными другими методами, приведенными далее в описании параметра wal_sync_method. Это гарантирует восстановление кластера данных до целостного состояния после аппаратного сбоя или сбоя ОС.

Несмотря на то, что отключение fsync() зачастую позволяет повысить производительность, это может привести к невосстановимым повреждениям данных в случае отказа питания или системного сбоя. Единственной причиной допустимого отключения fsync() может быть возможность восстановления всей базы данных с внешнего источника данных. Примером может является начальная загрузка данных в новый кластер из резервной копии, использования кластера для обработки пакета данных, после чего база данных будет отброшена и пересоздана, или для копии базы данных только для чтения, которая часто пересоздается и не используется для обеспечения отказоустойчивости. Само по себе высокое качество аппаратных комплектующих не является веской причиной для отключения fsync().

Для надежного восстановления при смене значения fsync() с off на on необходимо сбросить все модифицированные ядром буфера на надежное хранилище. Это может быть выполнено во время остановки кластера или установки fsync() запуском команды initdb --sync-only, запуском команды sync, перемонтированием файловой системы или перезапуском сервера.

Во многих случаях, отключение synchronous_commit для некритических транзакций может обеспечить больший выигрыш в производительности по сравнению с отключением fsync без сопутствующего риска повреждения данных.

Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера. При отключении параметра следует рассмотреть возможность отключения параметра full_page_writes.

- synchronous_commit (enum) определяет, должна ли операция завершения транзакции ожидать переноса записей журнала WAL на диск, прежде чем вернуть клиенту сообщение об успешном выполнении. Допустимые значения on, remote_write, local, и off.

Безопасное значение по умолчанию on (включен). При значении параметра off (выключен) возможно наличие задержки между сообщением клиенту об успешном выполнении транзакции и реальным моментом гарантированной защиты транзакции от сбоя или отказа на сервере. Максимальная длина задержки в три раза больше значения параметра wal_writer_delay. В отличие от fsync, отключение параметра не создает риск для непротиворечивости БД: сбой может привести к потере последних выполненных транзакций, но состояние БД будет соответствовать состоянию при котором транзакции были прерваны корректно. Таким образом, отключение synchronous_commit может оказаться полезным, если производительность важнее, чем уверенность в завершенности транзакций.

Если задан параметр synchronous_standby_names рассматриваемый параметр определяет будет ли завершение транзакции ожидать репликации записей журнала WAL транзакции на резервный сервер. При значении on, завершение (commit) транзакции ожидает ответа от текущего синхронного резервного сервера о получение записей журнала для транзакции и сброса их на диск. Это гарантирует, что транзакция не будет потеряна, кроме случая одновременного разрушения хранилища баз данных на всех серверах. При значении remote_write, завершение транзакции ожидает ответа от текущего синхронного резервного сервера о получение записей журнала для транзакции и записи с помощью ОС, но данные при этом могут еще не сохраниться на физическом носителе. Это значение достаточно для сохранения данных в случае сбоя экземпляра сервера PostgreSQL, но не в случае сбоя ОС.

В случае использования синхронной репликации, целесообразно ожидать сохранения на носителе данных журнала как на основном, так и на резервном сервере, или же разрешить транзации завершаться асинхронно. При значении local завершение транзакции ожидает только локального сохранения на основном сервере, без учета синхронной репликации. Если параметр synchronous_standby_names не задан, а значения on, remote_write, local обеспечивают одинаковый уровень синхронизации: завершение транзакции ожидает только локального сохранения.

Значение параметра может быть изменено в любое время: поведение каждой транзакции будет определяться значением параметра в момент проведения транзакции. Следовательно, возможно и полезно совершать одни транзакции синхронно, а другие асинхронно. Например, при необходимости асинхронно провести транзакцию,

состоящую из множества запросов при значении по умолчанию on включено, следует выполнить внутри транзакции кманду SET LOCAL synchronous_commit TO OFF.
- wal_sync_method (enum) определяет метод, используемый для принудительной записи обновлений журнала WAL на диск. При значении параметра fsync off (выключен) использование параметра wal_sync_method (enum) не имеет смысла, вследствие отсутсвия принудительной записи обновлений на диск. Возможные значения параметра:

- open_datasync (запись файлов WAL с опцией O_DSYNC функции open());
- fdatasync (вызывать fdatasync() при каждом завершении транзакции);
- fsync (вызывать fsync() при каждом завершении транзакции);
- fsync_writethrough (вызывать fsync() при каждом завершении транзакции для принудительной сквозной записи через дисковый кэш);
- open_sync (запись файлов WAL с опцией O_SYNC функции open()).

Не все значения параметра доступны для на всех платформах. По умолчанию из списка берется первый метод, поддерживаемый платформой. Опции open_* также по возможности используют O_DIRECT. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.
- full_page_writes (boolean) определяет при значении on (включен), что сервер PostgreSQL пишет полное содержимое каждой страницы в журнал WAL во время первой модификации этой страницы после контрольной точки. Данная возможность необходима, поскольку сбой или отказ операционной системы в процессе записывания страницы может привести к появлению на диске страницы со смешанными старыми и новыми данными. Построчные изменения данных, которые обычно хранятся в журнале WAL, могут быть не достаточны для полного восстановления страницы после сбоя или отказа операционной системы. Хранение полного образа страницы гарантирует, что страница может быть восстановлена правильно, но приводит к увеличение количества данных, которые должны быть записаны в журнал WAL. Поскольку чтение журнала WAL всегда начинается с контрольной точки, запись удобно осуществлять при первом изменении каждой страницы после контрольной точки. Следовательно, одним из способов уменьшения накладных расходов на запись страниц является увеличение интервала между контрольными точками.
Отключение параметра ускоряет работу, но может привести к повреждению БД в случае сбоя или отказа операционной системы или отключения питания. Риски аналогичны рискам связанным с отключением fsync, хотя в данном случае они меньше.
Отключение параметра не влияет на использование архива журнала WAL для

пошагового восстановления (Point-In-Time Recovery – PITR) (смотри 14.3).
Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера. Значение по умолчанию on.

- wal_log_hints (boolean). Если этот параметр on, сервер PostgreSQL пишет все сожержимое каждой страницы диска в WAL во время первого изменения этой страницы после контрольной точки, даже для некритичных изменений так называемых hint bits. Если контрольные суммы данных используются, то всегда обновятся hint bits WAL-журнала и этот параметр игнорируется. Вы можете использовать этот параметр, чтобы проверить, сколько дополнительных логов WAL бы произойти, если ваша база данных использовала контрольные суммы данных. Этот параметр может быть задан только при запуске сервера. Значением по умолчанию является off.

Примечание. Данный параметр используется в СУБД версии 9.6.

- wal_buffers (integer) определяет объем разделяемой памяти, используемой для данных журнала WAL, еще не сохраненных на диске. Значение по умолчанию -1 выбирает величину равной 1/32 (около 3%) от shared_buffers, но не менее 64Кб и не более размера сегмента WAL (обычно 16Мб). Значение может быть задано вручную, если автоматически выбирается слишком малое или большое значение, но любое значение, меньшее 32Кб, воспринимается как 32Кб. Значение параметра может быть задано только при запуске сервера.

Содержимое буферов WAL записываются на диск по каждому завершению транзакции, поэтому слишком большие значения вряд ли принесут значительную пользу. Однако, установка значение как минимум в несколько мегабайт может увеличить производительность записи на сервере, клиенты которого завершают большое количество транзакций одновременно. В большинстве случаев автоматических выбор по значению -1 дает приемлемые результаты.

- wal_writer_delay (integer) определяет в миллисекундах длительность задержки между рабочими циклами процесса, записывающего WAL. В каждом цикле процесс осуществляет запись журнала WAL на диск. Далее процесс «засыпает» на количество миллисекунд, указанное в значении параметра wal_writer_delay, и цикл повторяется. Значение по умолчанию 200 миллисекунд (200ms). Следует отметить, что что для многих систем эффективное разрешение задержки составляет 10 миллисекунд. Значение округляется в сторону ближайшего большего числа кратного 10. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

- commit_delay (integer) определяет в микросекундах время задержки между помещением записи о завершении транзакции в буфер журнала WAL и переносом

буфера на диск. Это может повысить производительность, поскольку позволяет нескольким транзакциям завершиться с одним сбросом на диск журнала WAL, при загрузке системы, достаточной для возникновения в пределах временного интервала задержки дополнительных транзакций готовящихся к завершению. Однако, это также увеличивает задержку для каждого сброса журнала WAL на диск.

При отсутствии других транзакций, готовящихся к завершению, временная задержка не используется. Следовательно, задержка выполняется при наличии некоторого определенного значением параметра `commit_siblings` числа транзакций активных в момент создания процессом сервера записи о завершении транзакции. Если параметр `fsync` выключен, временная задержка не используется. Значение по умолчанию `0` (нет задержки). Только суперпользователь может изменить этот параметр. В версиях PostgreSQL, ранее 9.3, `commit_delay` действовал по другому и менее эффективно: он действовал только при завершении транзакций, а не при любых сбросах журнала WAL, и ожидал полное время задержки в случае недавнего сброса журнала. Начиная с версии 9.3, первый готовый к сбросу процесс, ожидает установленный интервал, тогда как остальные ждут только до момента сброса журнала первым процессом.

- `commit_siblings (integer)` определяет минимальное число конкурирующих открытых транзакций, необходимое для выполнения задержки на время, заданное значением параметра `commit_delay`. Увеличение значения повышает вероятность завершения по крайней мере еще одной транзакции во время задержки. Значение по умолчанию `5` транзакций.

8.5.2. Контрольные точки

- `checkpoint_segments (integer)` определяет максимальное число сегментов файла журнала между автоматическими контрольными точками журнала WAL (размер каждого сегмента обычно составляет 16 мегабайт). Значение по умолчанию `3`. Увеличение значения параметра может привести к увеличению времени восстановления после сбоя или отказа операционной системы. Значение параметра может быть задано только в файле `postgresql.conf` или в командной строке сервера.
- `checkpoint_timeout (integer)` определяет в секундах максимальное время между автоматическими контрольными точками журнала WAL. Значение по умолчанию пять минут (`5min`). Увеличение значения может привести к увеличению времени восстановления после сбоя или отказа операционной системы. Значение параметра может быть задано только в файле `postgresql.conf` или в командной строке сервера.
- `checkpoint_completion_target (floating point)` определяет в долях кон-

трольного интервала целевую продолжительность контрольных точек. Значение по умолчанию 0.5. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

- checkpoint_warning (integer) определяет необходимость записи в журнал сервера сообщения в случае когда контрольные точки, полученные наполнением файлов сегментов контрольных точек, оказываются на более близком расстоянии в секундах, чем задано. Сообщение означает, что значение параметра checkpoint_segments следует увеличить. Значение по умолчанию 30 секунд (30s). Если значение checkpoint_timeout меньше чем checkpoint_warning сообщения не генерируются. Значение 0 отключает сообщения. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

8.5.3. Архивирование

- archive_mode (boolean) определяет при значении on (включен), что заполненные сегменты WAL могут быть отправлены в архивное хранилище, посредством соответствующей команды заданной в значении параметра archive_command. archive_mode и archive_command являются разными переменными, таким образом archive_command может быть изменена без выхода из режима архивирования. Значение параметра может быть задано только при запуске сервера. archive_mode не может быть включено при значении wal_level равном minimal.

- archive_command (string) определяет команду для выполнения архивирования заполненного сегмента серий файлов журнала WAL. Вместо %p в строке указывается путь к файлу, который необходимо заархивировать, вместо %f записывается имя файла. Путь указывается относительно рабочей директории сервера, которой является директория кластера БД. Для вставки знака % используется %%. Дополнительная информация приведена в 14.3.1.

Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера. При значении параметра archive_mode off (выключен) параметр archive_command (string) игнорируется. Значение параметра по умолчанию archive_command пустая строка. Данное значение при значении параметра archive_mode on (включен) определяет, что архивирование журнала WAL временно отключается, но сервер продолжает собирать файлы сегмента журнала WAL в ожидании поступления команды на архивирование. Важно, чтобы команда возвращала нулевой результат тогда и только тогда, когда она выполнена успешно.

- archive_timeout (integer) определяет в секундах предельное время существования старых данных не в архиве. Команда archive_command вызывается только для заполненных сегментов журнала WAL. Следовательно, если сервер постоянно

или периодически генерирует незначительное количество информации для журнала WAL, то возможна длительная задержка между завершением транзакции и безопасной ее записью в архивное хранилище. Значение параметра `archive_timeout` позволяет заставлять сервер периодически переключаться на новый файл сегмента журнала WAL. При значении параметра больше нуля, сервер будет переключаться на новый файл по истечении указанного числа секунд с момента последнего переключения. Следует отметить, что архивные файлы, преждевременно закрытые при вынужденном переключении, будут иметь длину равную длине полностью заполненных файлов. Следовательно, задание для параметра `archive_timeout` чрезмерно малого значения приведет к значительному увеличению размера архива. Обычно для параметра `archive_timeout` используется значение, равное приблизительно одной минуте. Значение параметра может быть задано только в файле `postgresql.conf` или в командной строке сервера.

8.6. Репликация

Приведенные в этом разделе конфигурационные параметры управляют встроенными средствами потоковой репликации. Сервера могут быть основными (Master) или резервными (Standby). Основные сервера отсылают, а резервные получают данные репликации. При использовании каскадной репликации, резервные сервера могут быть не только получателями, но и отправителями. Параметры приведены в основном для отсылающих и резервных серверов, хотя некоторые параметры могут иметь значения только для основного сервера. Параметры при необходимости могут отличаться внутри кластера.

8.6.1. Отсылающие сервера

Следующие параметры могут быть заданы для любого сервера, отсылающего данные репликации одному или более резервным серверам. Основной (master) сервер всегда является отсылающим, поэтому для него эти параметры должны быть всегда заданы. Роль и значение параметров не изменяются, после того как резервный сервер становится основным.

- `max_wal_senders (integer)` определяет максимально число одновременных соединений с резервными серверами или клиентов создания резервных копий (т.е. максимальное число одновременно запущенных процессов отсылки записей журнала WAL). Значением по умолчанию является 0, означающее запрет репликации. Значение параметра `max_wal_senders` не может превышать `max_connections`. Внезапное отключение потокового клиента может привести к прекращению репликации по тайм-ауту, поэтому этот параметр должен быть установлен немного выше, чем максимальное число ожидаемых клиентов, т.к. отключенные клиенты могут под-

ключиться снова. Параметр может быть задан только при запуске сервера. Параметр wal_level должен быть установлен в archive или hot_standby для возможности принятия соединений от резервных серверов.

- wal_keep_segments (integer) определяет минимальное количество старых сегментов файла журнала, сохраняемое в каталоге pg_xlog, предназначенных для забора их резервным сервером в процессе потоковой репликации. Каждый сегмент обычно составляет 16Мб. В случае сбоя резервного сервера за пределами wal_keep_segments сегментов, отправляющий сервер может удалить сегмент WAL, необходимый резервному серверу. В конечном итоге соединение с резервным сервером разрывается с ошибкой. (Резервный сервер может имеет возможность восстановится путем забора потерянного сегмента из архива, если включен режим архивирования WAL.

Параметр устанавливает только минимальное количество сегментов, оставляемых в pg_xlog; система может требовать большее их количество для архивирования журнала WAL или для восстановления от контрольной точки. Если значение wal_keep_segments равно нулю (по умолчанию), система не хранит дополнительных сегментов для целей резервирования, таким образом, количество старых сегментов WAL, доступных резервным серверам, зависит положения предыдущей контрольной точки и статуса архивирования WAL. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

- max_replication_slots (integer) определяет максимальное число слотов репликации (см. 15.2.6), которые сервер может использовать. По умолчанию равен 0. Этот параметр может быть установлен только при запуске сервера. Параметр wal_level должен быть установлен в значение archive или выше для возможности использования слотов реплкации. Установка на меньшее значение, чем существующее значение слотов репликации повлечет ошибку при запуске сервера.

Примечание. Данный параметр используется только в СУБД версии 9.6.

- wal_sender_timeout (integer) определяет время в миллисекундах до прерывания соединений репликации в случае их неактивности. Параметр предназначен для определения отсылающим сервером сбоев резервных серверов или сетевых проблем. Нулевое значение отключает указанный механизм. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера. Значение по умолчанию 60 секунд.

8.6.2. Основной (master) сервер

Следующие параметры могут быть заданы для основного сервера, отправляющего данные репликации одному или нескольким резервным. Следует отметить, что дополнитель-

но к этим параметрам на основном сервере должен быть соответствующим образом задан параметр wal_level, а так же возможно настроено архивирование журнала WAL. Для резервных серверов значения этих параметров игнорируется, но могут быть установлены на случай перевода резервного сервера в основной.

- synchronous_standby_names (string) определяет список разделенных запятой имен резервных серверов, поддерживающих синхронную репликацию. В каждый момент времени существует не более одного синхронного резервного сервера; ожидающие завершения транзакции будут завершены только после получения подтверждения их получение указанным сервером. Синхронный резервный сервер выбирается первым из списка и с ним устанавливается соединение для передачи данных в режиме реального времени (состояние потоковой репликации отражается в представлении pg_stat_replication). Следующие за ним в списке имена являются именами потенциальных синхронных резервных серверов. В случае отключения по какой-либо причине текущего синхронного резервного сервера, он немедленно будет заменен следующим по списку. Использование нескольких подобных резервных серверов может значительно улучшить отказоустойчивость.

  Именем резервного сервера для этих целей является параметр application_name резервного сервера, указанный в primary_conninfo его получателя WAL. Не существует механизма обеспечения уникальности имен. В случае обнаружения повторения имен, невозможно предсказать какой именно сервер будет выбран в качестве синхронного резервного. Специальное значение * означает любое application_name, включая имя получателя WAL по умолчанию.

  Если параметр synchronous_standby_names не задан, синхронная репликация отключена и завершения транзакций не ожидают репликации. По умолчанию параметр не задан. Даже если включена синхронная репликация, отдельные транзакции могут быть сконфигурированы как не ожидающие репликации, путем установки значений local или off параметру synchronous_commit.

  Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

- vacuum_defer_cleanup_age (integer) определяет число транзакций на которое будут отложены очистка ненужных версий записей механизмами типа VACUUM. Значением по умолчанию является ноль транзакций, означающее немедленное удаление ненужных записей если они больше не видны ни одной открытой транзакции. Для основного сервера, обслуживающего сервера горячего резерва, может быть задано отличное от нуля значение. Это предоставит больше времени для завершения запросов на резервных серверах без возникновения конфликтов, связанных

с ранней очисткой записей. Но поскольку значение задается числом транзакций основного сервера, сложно предсказать точно, сколько будет предоставлено времени резервным серверам. Значение параметра может быть задано только в файле `postgresql.conf` или в командной строке сервера.

Альтернативой использованию этого параметра может быть задание параметра `hot_standby_feedback` для резервных серверов.

8.6.3. Резервные (standby) сервера

Следующие параметры управляют поведением резервных серверов, получающих данные репликации. Для основного сервера значения этих параметров игнорируются.

- `hot_standby (boolean)` определяет могут или нет устанавливаться соединения для выполнения запросов в процессе восстановления. Значением по умолчанию является `off`. Параметр может быть задан только при старте сервера и имеет значение только в процессе восстановления из архива или в режиме резервирования.
- `max_standby_archive_delay (integer)` в режиме горячего резервирования, параметр определяет время, которое должен ожидать резервный сервер до отмены запросов, конфликтующих с требующими применения записями WAL. `max_standby_archive_delay` применяется когда данные WAL только получены из архива (и следовательно еще не применены). Значение по умолчанию 30 секунд. Без указания спецификатора значение задается в миллисекундах. Значение `-1` позволяет резервному серверу ожидать завершения конфликтующих запросов неограниченное время. Значение параметра может быть задано только в файле `postgresql.conf` или в командной строке сервера.

  Следует отметить, что `max_standby_archive_delay` не является максимальным временем выполнения запроса до его прерывания; скорее это максимальное допустимое время для применения данных сегмента WAL. Таким образом, если один из запросов вызвал значительную задержку ранее в сегменте WAL, последующим конфликтующим запросам останется меньше времени для успешного завершения.
- `max_standby_streaming_delay (integer)` в режиме горячего резервирования, параметр определяет время, которое должен ожидать резервный сервер до отмены запросов, конфликтующих с требующими применения записями WAL. `max_standby_streaming_delay` применяется когда данные получены с помощью потоковой репликации. Значение по умолчанию 30 секунд. Без указания спецификатора значение задается в миллисекундах. Значение `-1` позволяет резервному серверу ожидать завершения конфликтующих запросов неограниченное время. Значение параметра может быть задано только в файле `postgresql.conf` или в командной строке сервера.

Следует отметить, что max_standby_streaming_delay не является максимальным временем выполнения запроса до его прерывания; скорее это максимальное допустимое время для применения данных сегмента WAL. Таким образом, если один из запросов вызвал значительную задержку ранее в сегменте WAL, последующим конфликтующим запросам останется меньше времени для успешного завершения.

- wal_receiver_status_interval (integer) определяет минимальную частоту отправки сведений о ходе процесса репликации принимающим процессом резервного сервера основному или вышестоящему серверу, где эта информация может быть просмотрена с помощью представления pg_stat_replication. Резервный сервер должен сообщать позицию последней записанной им транзакции в журнале WAL, последнюю позицию, сброшенную на диск, и последнюю примененную позицию. Параметр задает максимальный интервал в секундах между такими сообщениями. Изменения отсылаются по каждому изменению позиции записи, или как минимум через указанный период. Таким образом, примененная позиция может немного отставать от действительной. Установка параметра в ноль полностью отключает обновление информации о ходе процесса репликации. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера. Значение по умолчанию 10 секунд.

- hot_standby_feedback (boolean) определяет будет ли отправлять резервный сервер основному или вышестоящему серверу информацию о выполняющихся запросах. Параметр может быть использован для устранения прерываний запросов из за удаления записей, но при этом база данных на основном сервере может увеличиваться в размерах для некоторых видов рабочей нагрузки. Сообщения посылаются не чаще, чем задано параметром wal_receiver_status_interval. Значение по умолчанию off. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

При каскадной репликации эти сообщения передаются выше по цепочке пока не достигнут основного сервера. Резервные сервера могут сами не генерировать таких сообщений, а только передавать выше сообщения, полученные от нижестоящих.

- wal_receiver_timeout (integer) завершает соединение для репликации в случае его неактивности в течение указанного количества миллисекунд. Используется для определения резервным сервером сбоя вышестоящего сервера или сети. Нулевое значение отключает указанный механизм. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера. Значение по умолчанию 60 секунд.

8.7. Планирование запросов

8.7.1. Настройка метода планировщика

Рассматриваемые здесь конфигурационные параметры предоставляют предварительный способ влияния на планы, выбранные оптимизатором запросов. В случае когда план, выбранный по умолчанию оптимизатором, для определенного запроса не является оптимальным, может быть найдено временное решение с использованием одного из параметров для указания оптимизатору на необходимость выбрать другой план. Постоянное выключение одного из рассматриваемых здесь параметров не рекомендуется. Для улучшения качества планов, выбираемых оптимизатором, следует включить применение стоимостных констант планировщика, чаще выполнять команду `ANALYZE`, увеличить значение конфигурационного параметра `default_statistics_target` и увеличить объем статистики, собираемой для определенных столбцов с помощью команды `ALTER TABLE SET STATISTICS`.

- `enable_bitmapscan (boolean)` разрешает или запрещает планировщику запросов использовать типы планов с побитовым сканированием (bitmap-scan). Значение по умолчанию `on`.
- `enable_hashagg (boolean)` разрешает или запрещает планировщику запросов использовать типы планов с агрегированием на основе хэширования (hashed aggregation). Значение по умолчанию `on`.
- `enable_hashjoin (boolean)` разрешает или запрещает планировщику запросов использовать типы планов с операциями соединения на основе хэширования (hash-join). Значение по умолчанию `on`.
- `enable_indexscan (boolean)` разрешает или запрещает планировщику запросов использовать типы планов с индексным сканированием (index-scan). Значение по умолчанию `on`.
- `enable_indexonlyscan (boolean)` разрешает или запрещает планировщику запросов использовать типы планов с только с индексным сканированием (index-only-scan). Значение по умолчанию `on`.
- `enable_material (boolean)` разрешает или запрещает планировщику запросов использовать типы планов с использованием материализации. Не существует возможности полностью подавить материализацию, но установка значения `off` указывает планировщику использовать материализацию только там, где это необходимо для корректного результата. Значение по умолчанию `on`.
- `enable_mergejoin (boolean)` разрешает или запрещает планировщику запросов использовать типы планов с операциями соединения методом слияния (merge-join). Значение по умолчанию `on`.
- `enable_nestloop (boolean)` разрешает или запрещает планировщику запро-

сов использовать планы с операциями соединения с вложенными циклами (nested-loop join). Полностью заблокировать операции соединения с вложенными циклами невозможно, но при установке для параметра запрещающего значения планировщик не будет использовать данный метод в случае доступности какого-либо другого метода. Значение по умолчанию `on`.

- `enable_seqscan (boolean)` разрешает или запрещает планировщику запросов использовать типы планов с операциями последовательного чтения (sequential scan). Полностью заблокировать операции последовательного чтения невозможно, но при установке для параметра запрещающего значения планировщик не будет использовать данный метод в случае доступности какого-либо другого метода. Значение по умолчанию `on`.

- `enable_sort (boolean)` разрешает или запрещает планировщику запросов использовать шаги явной сортировки (explicit sort steps). Полностью заблокировать явную сортировку невозможно, при установке для параметра запрещающего значения планировщик не будет использовать данный метод в случае доступности какого-либо другого метода. Значение по умолчанию `on`.

- `enable_tidscan (boolean)` разрешает или запрещает планировщику запросов использовать типы планов с операциями TID-сканирования (TID scan). Значение по умолчанию `on`.

8.7.2. Стоимостные константы планировщика

Описанные здесь параметры стоимости измеряются на произвольной шкале. Важны только их относительные значения, следовательно масштабирование значений всех параметров умножением на один и тот же множитель не повлияет на выбор планировщика. Традиционно единицей стоимости выбирается стоимость последовательной выборки страниц, следовательно значение параметра `seq_page_cost` обычно задается равным 1.0, а значения остальных параметров задаются относительно значения данного параметра. Может быть использована любая другая шкала, например, реальное время выполнения запроса в миллисекундах на определенной машине.

Примечание. Следует отметить, что, к сожалению, не существует строго определенного метода нахождения идеальных значений для переменных стоимости. Наилучшим является использование средних значений для всех запросов, к рабочей конфигурации СУБД. Задание значений их на основе данных всего нескольких экспериментов является очень рискованным.

- `seq_page_cost (floating point)` определяет оценку планировщика для стоимости выборки дисковой страницы, как части из серий последовательных выборок. Значение по умолчанию равно `1.0`. Значение может быть переопределено для

таблиц и индексов, расположенных в конкретном табличной пространстве, одноименным параметром (см. описание команды ALTER TABLESPACE).

- random_page_cost (floating point) определяет оценку планировщика для стоимости не последовательно выбранной страницы диска. Значение по умолчанию равно 4.0. Значение может быть переопределено для таблиц и индексов, расположенных в конкретном табличной пространстве, одноименным параметром (см. описание команды ALTER TABLESPACE).

Уменьшение значения относительно seq_page_cost приведет к тому, что система будет предпочитать индексное сканирование. Увеличение значения сделает индексное сканирование относительно более дорогим. Оба значения можно уменьшать и увеличивать вместе, изменяя важность стоимости дискового ввода-вывода относительно стоимости ресурсов центрального процессора, которые описываются последующими параметрами.

Произвольный доступ к механическому дисковому хранилищу обычно намного дороже чем четырехкратный последовательный доступ. Однако, использовано небольшое значение (4.0), поскольку предполагается, что большинство попыток произвольного доступа, таких как индексные чтения, относятся к данным в кэше. Значение по умолчанию может рассматриваться как модель ситуации, при которой произвольный доступ в 40 раз медленнее последовательного, но 90% случаев произвольного доступа кэшируется.

Если предположение о кэшировании 90% случаев произвольного доступа кажется не верным, значение может быть увеличено для отражения действительной стоимости произвольного доступа. Соответственно, если предполагается, что все данные помещаются в кэш, если база данных меньше чем доступная память на сервере, значение может быть уменьшено. Хранилища с небольшой стоимостью произвольного доступа по отношению к последовательному, например твердотельные носители, могут быть смоделированы лучше малыми значениями этого параметра.

Примечание. Хотя установка значения random_page_cost меньше, чем значение seq_page_cost разрешена системой, но физического смысла не имеет. Однако, имеет смысл установить для двух данных параметров равные значения, если БД полностью кэшируется в оперативной памяти, так как в этом случае не будет увеличения стоимости при непоследовательной выборке страниц. Кроме того, в интенсивно кэшируемой БД необходимо понизить оба значения относительно параметров стоимости ресурсов центрального процессора, т.к. стоимость выборки страницы, которая уже находится в оперативной памяти, намного ниже.

- cpu_tuple_cost (floating point) определяет оценку планировщика для сто-

имости обработки каждой строки при выполнении запроса. Значение по умолчанию равно 0.01.

- cpu_index_tuple_cost (floating point) определяет оценку планировщика для стоимости обработки каждой индексированной записи при индексном сканировании. Значение по умолчанию равно 0.005.

- cpu_operator_cost (floating point) определяет оценку планировщика для стоимости обработки каждого оператора или функции, выполняемых при выполнении запроса. Значение по умолчанию равно 0.0025.

- effective_cache_size (integer) определяет предположение планировщика об эффективном размере кэша на диске, доступного для единичного запроса. Значение учитывается в оценках стоимости использования индекса: высокие значения повышают вероятность использования индексного сканирования, низкие значения повышают вероятность использования последовательного сканирования. Устанавливая значение параметра, необходимо учитывать как разделяемые буферы PostgreSQL, так и часть дискового кэша ядра, которая будет использована файлами данных PostgreSQL. Кроме того, следует принимать во внимание ожидаемое число одновременных запросов к различным таблицам, так как они будут разделять доступное пространство. Параметр не влияет ни на объем разделяемой памяти, выделенной PostgreSQL, ни на дисковый кэш ядра и используется только для оценки. Значение по умолчанию 4 ГБ (4GB).

8.7.3. Оптимизация запросов на основе генетического алгоритма

Оптимизация запросов на основе генетического алгоритма (Genetic query optimizer – GEQO) является алгоритмом планирования с использованием эвристического поиска. Данный подход сокращает время планирования комплексных запросов, объединяющих много отношений, по сравнению с планами найденными с использованием обычного алгоритма полного поиска. Кроме того, поиск GEQO является рандомизированным и следовательно полученные с его помощью планы могут недетерминированно различаться.

- geqo (boolean) разрешает или запрещает оптимизацию запросов на основе генетического алгоритма. Значение по умолчании on (включена). Установка значения off не рекомендуется. Параметр geqo_threshold предоставляет возможность выборочно запретить GEQO для определенных классов запросов.

- geqo_threshold (integer) определяет использование оптимизации запросов на основе генетического алгоритма для планирования запросов с по меньшей мере таким количеством элементов FROM. Следует отметить, что конструкция FULL OUTER JOIN считается за один элемент FROM. Значение по умолчанию 12. Для простых запросов обычно рекомендуется использовать детерминированный

алгоритм полного поиска, но для запросов с большим количеством таблиц детерминированный алгоритм полного поиска будет выполняться слишком долго.

- geqo_effort (integer) определяет соотношение между временем планирования и эффективностью плана в GEQO. Значением параметра может быть целое число в интервале от 1 до 10. Значение по умолчанию 5. Увеличение значения повышает время затрачиваемое на планирование запроса, увеличивая вероятность выбора эффективного плана.

Параметр geqo_effort в действительности нигде не используется непосредственно: он применяется для вычисления значений по умолчанию всех остальных переменных, влияющих на поведение GEQO как описано далее. Использование параметра может быть заменено указанием значений последующих параметров вручную.

- geqo_pool_size (integer) определяет размер используемого GEQO пула, который является числом индивидов в генетической популяции. Минимальное значение 2. Полезные значения обычно лежат в диапазоне от 100 до 1000. Значение по умолчанию 0 определяет выбор подходящего значение на основе значения параметра geqo_effort и числа таблиц в запросе.

- geqo_generations (integer) определяет число используемых GEQO поколений, которое является числом итераций в алгоритме. Минимальное значение 1. Полезные значения обычно лежат в диапазоне от 100 до 1000. Значение по умолчанию 0 определяет выбор подходящего значение на основе значения параметра geqo_pool_size.

- geqo_selection_bias (floating point) определяет используемое GEQO смещение селекции, задавая селективное влияние внутри популяции. Значения могут лежать в диапазоне от 1.50 до 2.00. Значение по умолчанию 2.00.

- geqo_seed (floating point) задает начальное значение генератора псевдослучайных чисел, используемого GEQO для выбора случайного пути в пространстве поиска соединений. Может принимать значения от нуля (по умолчанию) до единицы. Изменение значения может влиять на качество получаемого результата.

8.7.4. Другие опции планировщика

- default_statistics_target (integer) определяет размер вектора статистики по умолчанию для столбцов таблицы, для которых он не был задан посредством ALTER TABLE SET STATISTICS. Большие значения увеличивают время, необходимое для выполнения ANALYZE, но могут улучшить качество оценок планировщика. Значение по умолчанию равно 100.

- constraint_exclusion (enum) определяет использование планировщиком запросов ограничений таблиц для оптимизации запросов. Допустимые значения пара-

метра constraint_exclusion: on (проверять ограничения для всех таблиц), off (никогда не проверять ограничения), partition (проверять ограничения только для дочерних таблиц при наследовании и подзапросов UNION ALL). Значение по умолчанию partition.

При значении параметра, разрешающем проверку для определенной таблицы, планировщик сравнивает условия запроса с ограничениями CHECK таблицы и пропускает сканирование таблиц, для которых условия противоречат ограничениям.

Пример

```
CREATE TABLE parent(key integer, ...);
CREATE TABLE child1000(check (key between 1000 and 1999))
INHERITS(parent);
CREATE TABLE child2000(check (key between 2000 and 2999))
INHERITS(parent);
...
SELECT * FROM parent WHERE key = 2400;
```

При значении параметра constraint_exclusion on, SELECT не будет сканировать таблицу child1000. При построении секционированных таблиц с использованием наследования возможно улучшение производительности.

В настоящее время параметра constraint_exclusion по умолчанию включен только для тех случаев, когда применяется секционирование таблиц. Включение параметра для всех таблиц требует дополнительных ресурсов при планировании, что очень заметно для простых запросов и практически никогда не обеспечивает выигрыш при выполнении простых запросов. Если секционированные таблицы отсутствуют, параметр рекомендуется не применять. Дополнительная информацию об исключающих ограничениях и секционировании приведена в 2.9.4.

- cursor_tuple_fraction (floating point) определяет оценку планировщика для доли строк, которые будут выбраны курсором. Значение по умолчанию 0.1. Уменьшение значения параметра склоняет планировщик к использованию для курсоров планов «быстрого старта», которые обычно возвращают первые несколько строк, в то время как получение всех строк может потребовать большого количества времени. Увеличение значения рекомендуется для увеличения в оптимизации важности критерия полного оценочное время. При максимальном значении 1.0 курсоры планируются точно так же, как обычные запросы, принимая во внимание лишь общее оценочное время без учета быстроты получения первых строк.

- from_collapse_limit (integer) определяет ограничение на количество элементов FROM в итоговом списке, при выполнении которого планировщик будет объ-

единять подзапросы в запросы более высокого уровня. Уменьшение значения уменьшает время планирования, но может ухудшить его качество. Значение по умолчанию 8.

Установка значения равным или превышающим geqo_threshold может вызвать использование планировщика GEQO, дающего не оптимальные планы.

- join_collapse_limit (integer) определяет ограничение на количество элементов FROM в итоговом списке, при выполнении которого планировщик будет переписывать явные конструкции JOIN (за исключением FULL JOIN) в списки элементов FROM. Уменьшение значения уменьшает время планирования, но могут может его качество.

Значение по умолчанию, применимое в большинстве случаев, равно значению параметра from_collapse_limit. Значение 1 означает, что не будет перегруппировки явных конструкций JOIN. Таким образом, порядок явных операций соединения, определенный в запросе, останется реальным порядком, в котором будут соединяться отношения. Планировщик запросов не всегда использует оптимальный порядок соединений, опытные пользователи могут временно установить значение параметра 1 и затем явно задать необходимый порядок соединений.

Установка значения равным или превышающим geqo_threshold может вызвать использование планировщика GEQO, дающего не оптимальные планы.

8.8. Сообщения об ошибках и протоколирование

8.8.1. Настройка параметров журнала

- log_destination (string) определяет список перечисленных через запятую мест, в которые необходимо протоколировать сообщения от сервера. СУБД PostgreSQL поддерживает несколько методов хранения сообщений от сервера, включая stderr, csvlog и syslog. По умолчанию сообщения сохраняются только в stderr. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

Если в значении параметра log_destination указан csvlog, то записи для журнала будут генерироваться в формате Comma Separated Value (CVS), то есть будут разделены запятой, что удобно при загрузке записей в программы. Дополнительная информация приведена в 8.8.4. Для обеспечения вывода в журнал формата CSV, необходимо установить значение параметра logging_collector on (включен).

Примечание. Следует отметить, что в большинстве операционных систем семейства Unix необходимо изменить конфигурацию для системного процесса syslog, чтобы включить опцию syslog для log_destination. PostgreSQL может сохра-

нять журнал в syslog под именем средств syslog от LOCAL0 до LOCAL7 (смотри параметр syslog_facility), но конфигурация syslog по умолчанию на большинстве платформ отклоняет подобные сообщения. Для настройки протоколирования в syslog необходимо в файл конфигурации демона syslog добавить строку приведенную в примере:

```
local0.*    /var/log/postgresql
```

- logging_collector (boolean) позволяет перехватывать и перенаправлять в файлы журнала сообщения, отправленные в stderr и журнал в формате CSV. Подобный подход зачастую полезнее протоколирования в syslog, так как некоторые типы сообщений могут быть не записаны в syslog (например, сообщения об ошибках динамического линковщика). Значение параметра может быть задано только при запуске сервера.
- log_directory (string) определяет, при значении параметра logging_collector on (включен), директорию, в которой будут создаваться файлы журнала. Можно указать абсолютный путь или путь относительно директории данных кластера. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера. По умолчанию используется pg_log.
- log_filename (string) определяет, при значении параметра logging_collector on (включен), имена файлов создаваемых журналов. Значение рассматривается как шаблон strftime, следовательно для указания имен, изменяющихся по времени, должны быть использованы управляющие последовательности с %. При использовании управляющих последовательностей, зависящих от часового пояса, вычисления проводятся для пояса, определенного в log_timezone. В случае когда системные шаблоны strftime не использованы непосредственно, применяются специфические для платформы (нестандартные) расширения.

При указании имени файла без управляющих последовательностей необходимо предусмотреть использование утилиты замены журнала для предотвращения переполнения протоколируемыми событиями дискового пространства. В версиях PostgreSQL, предшествовавших 8.4, при отсутствии управляющих последовательностей PostgreSQL добавлял время создания нового файла журнала, но данная возможность больше не используется.

При указании формата CSV в значении параметра log_destination, расширение .csv добавляется к временной метке в имени файла для создания имени файла в формате CSV. Если значение параметра log_filename заканчивается суффиксом

.log, то он замещается суффиксом .csv. Примером имени CSV файла может являться server_log.1093827753.csv.

Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера. По умолчанию используется postgresql-%Y-%m-%d_%H%M%S.log.

- log_file_mode (integer) определяет права доступа к файлу журнала при включенном logging_collector. Значение параметра предполагается заданным в числовом виде, принимаемым системными вызовами chmod и umask. (Для представления в восьмеричном виде, число должно начинаться с 0.)

По умолчанию для прав доступа используется значение 0600, означающее разрешение на чтение и запись только владельцу серверного процесса. Другим возможным значением может быть 0640, позволяющем членам группы владельца читать файл. Следует отметить, что для использования последнего значения необходимо задавать расположение log_directory для хранения файлов журнала вне каталога данных. Не рекомендуется делать доступным файлы журнала для всех, поскольку они могут содержать важные сведения.

Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

- log_rotation_age (integer) определяет в минутах, при значении параметра logging_collector on (включен), максимальное время жизни отдельного файла журнала. По истечении заданного числа минут создается новый файл журнала. Для запрета основанного на времени создания новых файлов необходимо установить значение 0. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

- log_rotation_size (integer) определяет, при значении параметра logging_collector on (включен), максимальный размер отдельного файла журнала. После внесения в файла журнала указанного числа килобайт, создается новый файл. Для отключения основанного на размере создания новых файлов необходимо установить значение 0. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

- log_truncate_on_rotation (boolean) определяет, при значении параметра logging_collector on (включен), что СУБД PostgreSQL будет заменять (перезаписывать) любой существующий файл журнала при совпадении имени, а не добавлять записи в него. Однако, перезапись будет осуществляться только при открытии нового файла основанного на времени, а не при запуске сервера или основанном на размере создании файла. При значении параметра off (выключен) во всех

случаях информация дописывается в существующие файлы. Например, использование этой опции с установленным для параметра log_filename значением postgresql-%H.log, приведет к созданию двадцати четырех часовых циклически перезаписываемых файлов журнала. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

Примеры:

1. Для хранения журналов за семь дней по одному файлу журнала на день, с именами server_log.Mon, server_log.Tue и так далее, и автоматической еженедельной перезаписью файлов необходимо указать для параметра log_filename значение server_log.%a, для параметра log_truncate_on_rotation значение on, а для параметра log_rotation_age значение 1440.
2. Для хранения журналов за двадцать четыре часа по одному файлу журнала на каждый час, при условии более ранней замены файла раньше, если его размер превышает 1 ГБ, необходимо указать для параметра log_filename значение server_log.%H%M, для параметра log_truncate_on_rotation значение on, для параметра log_rotation_age значение 60, а для параметра log_rotation_size значение 1000000. Включение управляющей последовательности %M в значение параметра log_filename разрешает замену файла, основанную на размере, что может привести к выбору для файла имени отличного от исходного имени файла для данного часа.

- syslog_facility (enum) определяет используемое при разрешении записи в syslog средство syslog: LOCAL0, LOCAL1, LOCAL2, LOCAL3, LOCAL4, LOCAL5, LOCAL6, LOCAL7. Значение по умолчанию LOCAL0. Дополнительная информация приведена в описании демона операционной системы syslog. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.
- syslog_ident (string) определяет при разрешении записи в syslog имя программы, используемой для идентификации сообщений PostgreSQL в журналах syslog. Значение по умолчанию postgres. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

8.8.2. Временные параметры протоколирования

- client_min_messages (enum) определяет уровень отправляемых клиенту сообщений. Возможные значения выбираются из последовательности: DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, LOG, NOTICE, WARNING, ERROR, FATAL и PANIC. Каждый

уровень включает все последующие. Чем выше уровень, тем меньше сообщений отправляется. Значение по умолчанию NOTICE. Следует отметить, что в данном случае уровень, заданный значением LOG, отличается от соответствующего уровня в последовательности возможных значений для параметра log_min_messages.

- log_min_messages (enum) определяет уровень, заносимых в журнал сервера сообщений. Возможные значения выбираются из последовательности: DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, INFO, NOTICE, WARNING, ERROR, LOG, FATAL и PANIC. Каждый уровень включает все последующие. Чем выше уровень, тем меньше сообщений заносится в журнал. Значение по умолчанию NOTICE. Следует отметить, что в данном случае уровень, заданный значением LOG, отличается от соответствующего уровня в последовательности возможных значений для параметра client_min_messages. Значение параметра могут изменять только суперпользователи.

- log_min_error_statement (enum) определяет уровень записи в журнал сервера выражений SQL, которые приводят к возникновению ошибки. Текущее выражение SQL включается в запись журнала для любого сообщения заданного или более высокого уровня. Возможные значения: DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, INFO, NOTICE, WARNING, ERROR, LOG, FATAL и PANIC. Значение по умолчанию ERROR определяет, что будут записываться все выражения SQL, приводящие к возникновению сообщений следующих типов: ошибка, сообщение в журнал, фатальная ошибка или паника. Для отключения протоколирования в журнале сервера выражений SQL, приводящих к возникновению ошибок, следует установить значение параметра PANIC. Значение параметра могут изменять только суперпользователи.

- log_min_duration_statement (integer) определяет в миллисекундах пороговое значение начиная с которого в журнал для каждого выражения записывается длительность его выполнения. Значение 0 определяет, что длительность выполнения будет записываться для всех выражений. Например, значение 250ms, определяет, что будут протоколироваться все выражения SQL, которые выполняются 250 миллисекунд или дольше. Использование параметра для отслеживания не оптимальных запросов в приложениях. Значение параметра могут изменять только суперпользователи.

Для клиентов, использующих расширенный протокол запросов, длительность выполнения этапов Parse, Bind и Execute вычисляется независимо.

Примечание. При использовании параметра совместно с параметром log_statement, текст выражений, которые будут записаны в журнал в соответствии со значением параметра log_statement, не будет повторяться в со-

общениях, записываемых в журнал в соответствии со значением параметра log_min_duration_statement. В случае когда syslog не используется, рекомендуется записывать в журнал идентификатор процесса (PID) или идентификатор сессии (ID), используя параметр log_line_prefix (смотри 8.8.3), для последующего связывания сообщения о выражении SQL с сообщением о длительности его выполнения.

В таблице 103 представлены уровни важности сообщений, используемые PostgreSQL. При протоколировании в syslog уровни важности преобразуются так, как показано в таблице.

Таблица 103 – Уровни важности сообщений

Уровень	Использование	syslog
DEBUG1..DEBUG5	Предоставляет последовательно детализируемую информацию для разработчиков	DEBUG
INFO	Предоставляет информацию, неявно запрашиваемую пользователем, например, от VACUUM VERBOSE	INFO
NOTICE	Предоставляет информацию, которая может оказаться полезной для пользователей, например, замечание о замене длинных идентификаторов	NOTICE
WARNING	Выдает предупреждения о вероятных проблемах, например, COMMIT вне блока транзакции	NOTICE
ERROR	Сообщает об ошибке, вызвавшей прерывание выполнения текущей команды	WARNING
LOG	Выдает сообщения для администраторов, например, о контрольных точках	INFO
FATAL	Сообщает об ошибке, вызвавшей прерывание текущей сессии	ERR
PANIC	Сообщает об ошибке, вызвавшей прерывание текущей сессии	CRIT

8.8.3. Настройка протоколируемых событий

- application_name (string) задает имя приложения длиной не более NAMEDATALEN (255) символов. Используется при установке соединений с сервером. Имя отображается в представлении pg_stat_activity и включается в записи CSV журнала. Также оно может быть включено в записи журнала с помощью параметра log_line_prefix. Имя может содержать только печатные ASCII символы, другие символы будут представлены знаком вопроса (?).

- debug_print_parse (boolean)

- debug_print_rewritten (boolean)

- debug_print_plan (boolean) разрешают выдачу различных отладочных сообщений. Установка для параметров значений on означает, что для каждого выпол-

няемого запроса выводится итоговое дерево разбора, результат работы модуля перезаписи запросов и план выполнения. Данные сообщения выдаются на уровне LOG, следовательно по умолчанию они будут занесены в журнал сервера, но не будут отправлены клиенту. Для изменения ситуации необходимо использовать параметры client_min_messages и/или log_min_messages. По умолчанию значения параметров off (выключен).

- debug_pretty_print (boolean) определяет использование отступов в сообщениях, созданных в соответствии со значениями параметров debug_print_parse, debug_print_rewritten или debug_print_plan. Сообщения записываются в удобном для чтения формате, но по сравнению с компактным форматом (при значении off параметра debug_print_parse) увеличивается длина сообщений. Значение по умолчанию on.

- log_checkpoints (boolean) определяет необходимость записи контрольных точек в журнал. В сообщения включается некоторая статистическая информация о каждой контрольной точке, в том числе количество записанных буферов и время, потраченное на их запись. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера. Значение параметра по умолчанию off (выключен).

- log_connections (boolean) определяет необходимость записи в журнал информации о каждой попытке соединения с сервером наряду с записями об успешной аутентификации клиента. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера. Значение параметра по умолчанию off (выключен).

Примечание. Некоторые клиентские приложения осуществляют две попытки соединения, определяя необходимость использования пароля, следовательно дублированные сообщения «connection received» (получено соединение) не обязательно указывают на наличие проблемы.

- log_disconnections (boolean) определяет необходимость записи в журнал сообщений о закрытии сессии, схожих с сообщениями определяемыми параметром log_connections и содержащими длительность сессии. Значение параметра по умолчанию off (выключен). Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

- log_duration (boolean) определяет необходимость сохранения в журнале длительности выполнения каждого выражения. Значение по умолчанию off (выключен). Значение параметра могут изменять только суперпользователи.

Для клиентов, использующих расширенный протокол запросов, длительность выпол-

нения этапов Parse, Bind и Execute записывается независимо.

Примечание. Разница между установкой значения on (включен) для параметра log_duration и присваиванием нулевого значения log_min_duration_statement заключается в том, что превышение значения log_min_duration_statement приводит к сохранению в журнале текста запроса, а использование параметра log_duration — нет. Таким образом, если значение параметра log_duration on (включен), а параметр log_min_duration_statement имеет положительное значение, в журнал записываются все длительности, но текст запроса включается только для тех выражений, которые превышают пороговое значение. Подобное поведение может использовано для сбора статистики в БД с высокой нагрузкой.

- log_error_verbosity (enum) определяет степень детализации каждого сообщения, записываемого в журнал сервера. Возможные значения выбираются из последовательности: TERSE, DEFAULT и VERBOSE. Повышение уровня расширяет набор полей для отображаемых сообщений. Значение параметра могут изменять только суперпользователи. TERSE отключает вывод DETAIL, HINT, QUERY и CONTEXT информацию. VERBOSE включает код ошибки SQLSTATE и имя файла исходного кода, имя функции и номер строки, в которой была вызвана ошибка. Значение параметра могут изменять только суперпользователи.

- log_hostname (boolean) определяет необходимость к сохранения в журнале имени узла, с которого осуществляется подключение к СУБД, наряду с IP-адресом, сохранение которого осуществляется по умолчанию. Следует отметить, что в зависимости от настройки правил разрешения имен узлов использование параметра может привести к заметному снижению производительности. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

- log_line_prefix (string) определяет строку в стиле printf, которая выводится в начале каждой записи журнала. Знаки % начинают управляющие последовательности, которые заменяются на информацию о статусе, как показано в таблице 104. Нераспознанные управляющие параметры игнорируются. Прочие символы копируются прямо в сообщение журнала. Некоторые управляющие последовательности распознаются только процессами сессий и не применяются для фоновых процессов, таких как главный процесс сервера. Информация лога может быть выровнена вправо или влево, указав числовой литерал после % и до нее. Отрицательное значение приведет к тому, что статусная информация будет дополнена справа пробелами, чтобы дать ему минимальную ширину, в то время как положительное значение отступ слева. Заполнение может быть полезно для улучшения читаемости лог-файла.

Значение параметра может быть задано только в файле `postgresql.conf` или в командной строке сервера. Значением по умолчанию является пустая строка.

Таблица 104 – Управляющие последовательности для параметра

%?	Действие	Только в сессии
%a	Имя приложения	да
%u	Имя пользователя	да
%d	Имя БД	да
%r	Имя или IP-адрес удалённого узла и удалённого порта	да
%h	Имя или IP-адрес удалённого узла	да
%p	Идентификатор процесса	нет
%t	Отметка о времени без миллисекунд	нет
%m	Отметка о времени с миллисекундами	нет
%i	Тэг команды: тип текущей команды сессии	да
%e	Код ошибки SQLSTATE	нет
%c	Идентификатор сессии: смотри далее	нет
%l	Номер записи в журнале для каждой сессии или процесса, начиная с 1	нет
%s	Время запуска процесса	нет
%v	Идентификатор виртуальной транзакции	нет
%x	Идентификатор транзакции (0, если нет назначенных)	нет
%q	Вывод не осуществляется. Указывает процессам, не обслуживающим сессии, на необходимость остановиться в данной позиции строки, игнорируется процессами сессий	нет
%%	Символ %	нет

Управляющая последовательность `%c` печатает квазиуникальный идентификатор сессии, состоящий из двух четырехбайтных чисел в шестнадцатеричной системе (нули в начало не добавляются), разделенных точкой. Первое число определяет время запуска процесса, второе определяет идентификатор процесса, так образом управляющая последовательность `%c` может быть использована для экономии места при выводе указанных значений.

Пример

```
SELECT to_hex(EXTRACT (EPOCH FROM backend_start)::integer) || '.' ||
       to_hex(periodic)
FROM pg_start_activity;
```

При установке для параметра `log_line_prefix` непустого значения рекомендуется в качестве последнего символа значения использовать пробел или знак пунктуации

для визуального разделения значения от остальной записи. Использование управляющих последовательностей может быть нецелесообразным с syslog, который добавляет свои метки времени и информацию об идентификаторе процесса.

- log_lock_waits (boolean) определяет необходимость создания в журнале сообщений о превышении времени ожидания сессией блокировки предела, установленного значением параметра deadlock_timeout. Использование параметра полезно для определения влияния ожидания блокировки на производительность. Значение по умолчанию off (выключен).

- log_statement (enum) определяет протоколируемые выражения SQL. Возможные значения: none, ddl, mod и all. Значение ddl указывает, что записываются все операторы, определяющие данные, такие как CREATE, ALTER и DROP. Значение mod указывает, что записываются все операторы ddl плюс операторы, модифицирующие данные, такие как INSERT, UPDATE, DELETE, TRUNCATE и COPY FROM. Выражения PREPARE, EXECUTE и EXPLAIN ANALYZE записываются в журнал при условии наличия в них команды подходящего типа. Для клиентов, использующих расширенный протокол запросов, протоколирование производится когда получено сообщение Execute и включены значения параметров Bind (в которых каждый вложенный знак одинарной кавычки удваивается).

Значение по умолчанию none. Значение параметра могут изменять только суперпользователи.

Примечание. Операторы, содержащие простые синтаксические ошибки, не записываются в журнал, даже если для параметра log_statement установлено значение all, поскольку сообщение записывается только после проведения простейшего разбора для определения типа выражения. В случае расширенного протокола запросов использование параметра log_statement также не обеспечивает запись протоколирование для выражений, которые дали сбой до фазы Execute (например, во время анализа или планирования). Для записи подобных сообщений необходимо указать в значении для параметра log_min_error_statement уровень ERROR (или ниже).

- log_temp_files (integer) определяет протоколирование использования временных файлов. Временные файлы могут создаваться для операций сортировки, хэширования и хранения временных результатов запросов. Запись в журнал осуществляется для каждогог временного файла перед его удалением. Значение 0 определяет, что протоколируются все временные файлы. Положительное число устанавливает порог в килобайтах, при котором протоколируются файлы с размером, равным или превышающим заданный порог. Значение по умолчанию -1 означает

запрет на запись временных файлов. Значение параметра могут изменять только суперпользователи.

- log_timezone (string) задает часовой пояс, используемый для отметок о времени в журнале. В отличие от TimeZone, значение данного параметра распространяется на весь кластер, следовательно все сессии будут выдавать согласованные отметки о времени. Значение по умолчанию GMT, но обычно переопределяется в postgresql.conf. Утилита initdb устанавливает использование часового пояса, установленного в системном окружений (смотри 5.5.3). Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

8.8.4. Использование формата CVS для протоколирования

Включение csvlog в значение параметра log_destination предоставляет удобный способ для импортирования файлов журнала в таблицу БД. Использование значения csvlog определяет, что записи журнала создаются в формате набора разделенных запятой следующих полей:

- время с точностью до миллисекунд;
- имя пользователя;
- имя БД;
- идентификатор процесса;
- узел:
  - номер порта;
  - идентификатор сессии;
  - номер записи для сессии или для процесса;
  - тип команды;
  - время начала сессии;
  - идентификатор виртуальной транзакции;
  - идентификатор реальной транзакции;
  - важность ошибки;
  - код состояния SQLSTATE;
  - сообщение об ошибке;
  - детализация сообщения об ошибке;
  - подсказку;
  - внутренний запрос, который привел к возникновению ошибки (при наличии запроса);
  - номер позиции ошибочного символа внутреннего запроса;
  - контекст ошибки;
  - запрос пользователя, который привел к возникновению ошибки (при нали-

чии запроса и необходимости его протоколирования, определяемой значением параметра log_min_error_statement);

- номер позиции ошибочного символа в запросе пользователя;
- расположение ошибки в исходном коде PostgreSQL (при значении параметра log_error_verbosity verbose);
- имя приложения application_name.

Далее приведен пример определения таблицы для хранения сообщений журнала в формате CSV:

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  PRIMARY KEY (session_id, session_line_num)
);
```

Для импортирования файла журнала в приведенную в примере таблицу необходимо использовать команду COPY FROM:

```
COPY postgres_log FROM '/full/path/to/logfile.csv' WITH csv;
```

Для упрощения автоматического импортирования файлов журнала в формате CSV, необходимо выполнить следующие шаги:

1) Укажите для параметров `log_filename` и `log_rotation_age` значения, обеспечивающие согласованную предсказуемую схему именования для файлов журнала, что позволит прогнозировать имя файла и точно знать, когда протоколирование в конкретный файл журнала завершено и файл готов к импортированию.

2) Укажите для параметра `log_rotation_size` значение 0 для запрета смены файла, на основе размера, затрудняющего предсказывание имени файла.

3) Укажите для параметра `log_truncate_on_rotation` значение `on` для предотвращения смешивания старых данных протоколирования с новыми в одном файле.

4) Приведенное ранее определение таблицы для импортирования журнала в формате CSV включает спецификацию первичного ключа. Подобный подход полезен для предотвращения случайного повторного импортирования информации. Команда `COPY` фиксирует все импортируемые за один раз данные, следовательно любая ошибка приводит к отказу процесса импортирования в целом. При импортировании части файла и последующем повтором импортировании данного файла произойдет ошибка, связанная с первичным ключом, которая приведет к отказу процесса импортирования в целом. Следовательно, импортировать файл необходимо по завершении протоколирования в данный файл и его закрытии. Данная процедура обеспечивает предотвращение случайного импортирования части не завершенной записи, приводящего к отказу при выполнении команды COPY.

8.9. Сбор статистической информации в режиме реального времени

8.9.1. Сборщик статистики о запросах и индексах

Приведенные далее параметры определяют настройки процесса сбора статистической информации для всего сервера СУБД. При включении сбора статистической информации, доступ к полученным данным может осуществлен с использованием семейства системных представлений `pg_stat` и `pg_statio`.

- `track_activities (boolean)` определяет необходимость сбора статистической информации по текущей выполняемой команде для каждой сессии, включая время начала выполнения команды. Значение параметра по умолчанию `off` (включен). Следует отметить, даже при значении параметра `on` полученная информация доступна не всем пользователям, а только суперпользователям и пользователю владельцу сессии. Следовательно, использование параметра не приводит к возникновению риска для безопасности информации. Значение параметра могут изменять только суперпользователи.

- track_activity_query_size (integer) задает число байтов в поле pg_stat_activity.current_query, зарезервированное для отслеживания текущей выполняемой команды в каждой активной сессии. Значение по умолчанию 1024 байт. Значение параметра может быть задано только при запуске сервера.
- track_counts (boolean) определяет необходимость сбора статистической информации по действиям с БД. Значение по умолчанию on (включен), так как данная информация необходима демону процесса автовакууминга. Значение параметра могут изменять только суперпользователи.
- track_io_timing (boolean) включает режим измерения времени операций ввода/вывода при работе с БД. Значение параметра по умолчанию off (включен), так как при этом производится частое обращение к ОС для получения текущего времени, что может привести к значительному увеличению нагрузки на систему. Полученные значения отображаются в pg_stat_database, в выводе команды EXPLAIN с использованием опции BUFFERS,и в pg_stat_statements. Значение параметра могут изменять только суперпользователи.
- track_functions (enum) определяет необходимость отслеживания функций (количество вызовов и использованное время). Для отслеживания только функций процедурных языков необходимо указать значение pl значение all указывается для отслеживания дополнительно функций языка SQL и языка C. Значение по умолчанию none, отключает сбор статистической информации о функциях. Значение параметра могут изменять только суперпользователи.

Примечание. Функции языка SQL являющиеся достаточно простыми, чтобы быть встроенными в вызывающий запрос не будут отслеживаться вне зависимости от значения параметра track_functions.

- update_process_title (boolean) определяет необходимость обновления заголовка процесса каждый раз, при получении сервером новой команды SQL. Заголовок процесса обычно просматривается с помощью команды ps. Значение параметра могут изменять только суперпользователи.
- stats_temp_directory (string) определяет каталог, в котором будут храниться временные статистические данные. Значением параметра может быть относительный или абсолютный путь к каталогу данных. Значение по умолчанию pg_stat_tmp. Использование параметра в файловых системах для устройств с произвольным доступом снижает требования к физическому вводу выводу и может привести у увеличению производительности. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

8.9.2. Мониторинг статистики

- log_statement_stats (boolean)
- log_parser_stats (boolean)
- log_planner_stats (boolean)
- log_executor_stats (boolean) определяют для каждого запроса запись в журнал статистической информации о производительности соответствующего модуля, предоставляя грубый инструмент профилирования. Использование параметра log_statement_stats позволяет собирать полную статистическую информацию по выражениям, прочие параметры позволяют собирать статистическую информацию по соответствующим модулям. Параметр log_statement_stats не может быть использован свместно с любым из параметров log_parser_stats (boolean), log_planner_stats (boolean), log_executor_stats (boolean). По умолчанию значения для всех параметров off. Значения параметров могут изменять только суперпользователи.

8.10. Автоматический вакууминг

Подробное описание работы демона autovacuum приведено в 13.1.6. Поведение демона autovacuum регламентируется следующими параметрами:

- autovacuum (boolean) определяет необходимость выполнения демона запуска автовакуумминга. Значение по умолчанию on (включен). Однако для функционирования автовакуумминга значение параметра track_counts должно быть также установлено в on (включен). Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

  Следует отметить, что при установленном для параметра autovacuum значении on система будет при необходимости запускать процесс автовакуумминга для предотвращения циклического переполнения счетчика идентификатора транзакций. Дополнительная информация приведена в 13.1.5.

- log_autovacuum_min_duration (integer) определяет в миллисекундах пороговое значение для длительности действий, выполняемых при автовакуумминге, при достижении которого действие протоколируется. Значение 0 определяет необходимость протоколирования всех действий автовакуумминга. Значение по умолчанию –1 отключает протоколирование в журнал действий автовакуумминга. Например, при установке значения 250 будут записаны все действия автоматического вакуумминга и анализа с длительностью равной 250 миллисекунд и больше. Использование параметра может быть полезным при отслеживании действий автовакуумминга. Значение параметра может быть задано только в файле postgresql.conf или в командной

строке сервера.

- autovacuum_max_workers (integer) определяет максимальное число одновременно работающих процессов автовакуумминга (кроме процесса запуска автовакуумминга). Значение по умолчанию 3. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

- autovacuum_naptime (integer) определяет минимальное время задержки между выполнением автовакуумминга для любой БД. В каждом цикле демон проверяет БД и по необходимости запускает для ее таблиц команды VACUUM и ANALYZE. Время задержки измеряется в секундах, значение по умолчанию равно одной минуте. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

- autovacuum_vacuum_threshold (integer) определяет минимальное число обновленных или удаленных кортежей, приводящее к запуску VACUUM на одной таблице. Значение по умолчанию 50. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера. Значение параметра может быть переопределено для отдельных таблиц изменением параметров хранения.

- autovacuum_analyze_threshold (integer) определяет минимальное число вставленных, обновленных или удаленных кортежей, необходимых для запуска команды ANALYZE для одной таблицы. Значение по умолчанию 50. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера. Значение параметра может быть переопределено для отдельных таблиц изменением параметров хранения.

- autovacuum_vacuum_scale_factor (floating point) определяет долю размера таблицы, которую нужно добавить к autovacuum_vacuum_threshold при принятии решения о запуске команды VACUUM. Значение по умолчанию 0.2 (20% размера таблицы). Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера. Значение параметра может быть переопределено для отдельных таблиц в изменением параметров хранения.

- autovacuum_analyze_scale_factor (floating point) определяет долю размера таблицы необходимо добавить к autovacuum_analyze_threshold при принятии решения о запуске команды ANALYZE. Значение по умолчанию 0.1 (10% от размера таблицы). Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера. Значение параметра может быть переопределено для отдельных таблиц изменением параметров хранения.

- autovacuum_freeze_max_age (integer) определяет максимальный возраст

(в транзакциях) для поля таблицы pg_class.relfrozenxid при достижении которого будет запущена команда VACUUM для предотвращения циклического переполнения счетчика идентификаторов транзакций внутри таблицы. Следует отметить, что система запустит процессы автовакуумминга для предотвращения циклического переполнения даже при значении off (выключен) параметра autovacuum.
Поскольку вакууминг также позволяет удалять старые файлы из каталога pg_clog, значение по умолчанию составляет 200000000 двести миллионов транзакций. Значение параметра может быть задано только при запуске сервера, но для отдельных таблиц значение может быть уменьшено изменением параметров хранения. Дополнительная информация приведена в 13.1.5.
- autovacuum_vacuum_cost_delay (integer) определяет значение стоимости задержки в миллисекундах, которое будет использовано в автоматических операциях VACUUM. Если указать значение -1, будет использовано значение vacuum_cost_delay. Значение по умолчанию 20 миллисекунд. Значение этого параметра может быть задано только в файле postgresql.conf или в командной строке сервера. Значение параметра может быть переопределено для отдельных таблиц изменением параметров хранения.
- autovacuum_vacuum_cost_limit (integer) определяет предельное значение стоимости, которое будет использоваться в автоматических операциях VACUUM. Значение по умолчанию -1 устанавливает, что будет использовано значение параметра vacuum_cost_limit. Следует отметить, что указанное значение распределяется пропорционально между выполняющимися процессами автовакуумминга, таким образом, что сумма предельных значений процессов не превышает предела заданного значением параметра autovacuum_vacuum_cost_limit. Значение параметра может быть задано только в файле postgresql.conf или в командной строке сервера. Значение параметра может быть переопределено для отдельных таблиц изменением параметров хранения.

8.11. Значения по умолчанию для соединений клиентов

8.11.1. Поведение выражений

- search_path (string) определяет порядок поиска в схемах пространства имен объекта (таблицы, типа данных, функции и прочего), ссылка на который является простым именем без указания схемы. В случае наличия в различных схемах пространства имен объектов с одинаковыми именами, используется первый найденный на основе значения параметра search_path объект. На объект, который не входит ни в одну из схем пространства имен в пути поиска, можно ссылаться только указав

содержащую его схему с помощью составного имени (с точкой).

Значением параметра search_path является список разделенных запятой имен схем. Имена несуществующих схем и схем, для которых пользователь не имеет права USAGE, игнорируются.

В случае когда элементом списка является специальное значение $user, то вместо него подставляется схема, имеющая имя, которое возвращает SESSION_USER при наличии такой схемы. Если такая схема отсутствует специальное значение $user игнорируется.

Схема системного каталога, pg_catalog, всегда включена в путь поиска вне зависимости от упоминания в значении параметра search_path. При наличии схемы системного каталога в пути поиск будет производится в указанном порядке. В случае отсутствия pg_catalog в пути, поиск в ней будет производится до поиска по прочим элементам, указанным в пути.

Аналогично, схема временных таблиц текущей сессии, pg_temp_nnn, при ее наличии, всегда включается в путь поиска. Данная схема может быть явно указана в пути посредством псевдонима pg_temp. Если данная схема не указана в пути, то поиск в ней будет проведен в первую очередь (даже перед поиском в схеме системного каталога pg_catalog). Однако, в схеме временных таблиц текущей сессии производится только поиск отношений (таблиц, представлений, последовательностей и прочих) и имен типов данных. В ней не будет производится поиск имен функций или операторов.

При создании объектов без указания конкретной целевой схемы, объекты будут помещены в первую из схем, указанных в пути поиска. При пустом значении параметра search_path будет выдано сообщение об ошибке.

Значение параметра по умолчанию "$user", public в котором вторая часть игнорируется при отсутствии схемы с именем public. Таким образом поддерживается разделяемое использование БД (когда пользователи не имеют индивидуальных схем используют разделяемую схему public), индивидуальные схемы пользователей и их сочетание. Другой порядок поиска объектов может быть определен посредством изменения значения параметра по умолчанию search_path, как глобального, так и для отдельных пользователей.

Для определения текущего действующего значения пути поиска используется функция SQL current_schemas(). Возвращаемое значение может отличаться от значения параметра search_path, поскольку значение current_schemas() определяет для запросов порядок разрешения имен объектов в пути поиска.

Дополнительная информация приведена в 2.7.

- default_tablespace (string) определяет табличное пространство по умолчанию, в котором должны создаваться объекты (таблицы и индексы), если в команде CREATE явным образом не указано табличное пространство. Значением параметра является имя табличного пространства, или пустая строка, означающая, что будет использовано табличное пространство по умолчанию для текущей БД. Если указанное значение не соответствует имени ни одного из существующих табличных пространств, PostgreSQL автоматически использует табличное пространство по умолчанию для текущей БД. Использование При указано табличное пространство не являющееся табличным пространством по умолчанию, то пользователь должен иметь права на использование в нем команды CREATE. В противном случае в создании объекта будет отказано.

Значение параметра не используется для временных таблиц (дополнительная информация в описании параметра temp_tablespaces).

Значение параметра не используется для создания баз данных. По умолчанию новая база данных наследует настройки табличных пространств от базы данных template, из которой она копируется.

Кроме того, дополнительная информация приведена в 11.6.

- temp_tablespaces (string) определяет табличные пространства для создания временных объектов (временных таблиц и индексов во временных таблицах), если в команде CREATE явным образом не указано табличное пространство. Временные файлы, используемые для сортировки больших объемов данных также будут создаваться в этих табличных пространствах.

Значением параметра является список имен табличных пространств. При указании в списке нескольких имен, PostgreSQL случайным образом выбирает имя из списка для создания каждого временного объекта. Исключением является поведение внутри транзакции. В этом случае последовательно созданные временные объекты помещаются в последовательные табличные пространства из списка. Если выбранный из списка элемент является пустой строкой, то PostgreSQL автоматически использует табличное пространство по умолчанию для текущей БД.

При интерактивном задании temp_tablespaces указание несуществующего табличного пространства приводит к возникновению ошибки, как и указание табличного пространства, для выполнения команды CREATE в котором, у пользователя отсутствуют права. Однако, при использовании предварительно установленного значения, все несуществующие табличные пространства игнорируются, как и пространства, для для выполнения команды CREATE в которых у пользователя отсутствуют права. Данное правило применимо, в частности, к значениям, заданным в файле

postgresql.conf.

Значением параметра по умолчанию является пустая строка, определяющая, что все временные объекты будут созданы в табличном пространстве по умолчанию для текущей БД.

Дополнительная информация приведена в описании параметра default_tablespace.

- check_function_bodies (boolean) указание значения параметра off (включен) отключает проверку тела функции при выполнении команды CREATE FUNCTION. Обычно значение параметра on (включен). Отключение проверки тела функции может быть полезным для предотвращения возникновения проблем, связанных с прямыми ссылками при восстановлении определения функции из дампа.

- default_transaction_isolation (enum) определяет уровень изоляции по умолчанию для каждой новой SQL-транзакции. Значение параметра может быть read uncommitted, read committed, repeatable read или serializable. Значение по умолчанию read committed.

- default_transaction_read_only (boolean) определяет статус по умолчанию «только для чтения» для каждой новой SQL-транзакция. SQL-транзакция со статусом «только для чтения» не может изменять не являющиеся временными таблицы. Значение по умолчанию off (выключен) определяет статус «чтение/запись».

- default_transaction_deferrable (boolean) определяет возможность задержки отложенных SQL транзакций в случае использования уровня изоляции serializable. В процессе исполнения такие транзакции не могут быть подвержены необходимому для обеспечения требуемого уровня изоляции контролю, что делает невозможность их прерывания в случае возникновения конкурентных обновлений. Параметр особенно применим к продолжительным транзакциям «только для чтения».

Параметр управляет статусом отложенности каждой новой транзакции. В настоящее время не применяется для транзакций, выполняющих и чтение и запись, или при уровне изоляции ниже serializable. Значение параметра по умолчанию off.

Дополнительная информация приведена в описании команды SET TRANSACTION.

- session_replication_role (enum) контролирует запуск триггеров и правил, относящихся к репликации, для текущей сессии. Изменение значения параметра требует прав суперпользователя и приведет к потере всех кэшированных планов запросов. Значение параметра может быть origin, replica и local. Значение параметра по умолчанию origin.

Дополнительная информация приведена в описании команды ALTER TABLE.

- statement_timeout (integer) определяет необходимость прерывания работы любого оператора, длящейся более указанного числа миллисекунд, начиная с момента получения сервером команды от клиента. Если значение параметра log_min_error_statement ERROR или ниже, остановленный оператор также будет записан в журнал. Значение по умолчанию 0 отключает снимает предельное ограничение.

Не рекомендуется изменять значение параметра statement_timeout в файле postgresql.conf, так как изменение повлияет на все сессии.

- lock_timeout (integer) определяет необходимость прерывания работы любого оператора, длящейся более указанного числа миллисекунд, при попытке получении блокировки на таблице, индексе, записи или ином объекте БД. Ограничение времени применяется отдельно для каждой попытки блокировке, причем как для неявных блокировок (например, LOCK TABLE или SELECT FOR UPDATE без указания NOWAIT) так и для явных попыток установки блокировок. Если значение параметра log_min_error_statement ERROR или ниже, остановленный оператор также будет записан в журнал. Значение по умолчанию 0 отключает снимает предельное ограничение.

В отличии от statement_timeout это таймаут возникает только при ожидании блокировок. Если statement_timeout отлично от нуля, бессмысленно устанавливать lock_timeout в то же или большее чем statement_timeout значение, так как в этом случае statement_timeout сработает раньше.

Не рекомендуется изменять значение параметра lock_timeout в файле postgresql.conf, так как изменение повлияет на все сессии.

- vacuum_freeze_table_age (integer) определяет значение, по достижению которого полем pg_class.relfrozenxid для таблицы, VACUUM будет осуществлять полное ее сканирование. Значение по умолчанию 150 миллионов транзакций. Несмотря на то, что пользователь может задать любое значение от нуля до 2 миллиардов, VACUUM ограничивает эффективное значение до 95% от autovacuum_freeze_max_age, так что периодический ручной запуск VACUUM имеет шанс выполниться до процедуры автовакууминга, запущенной для таблицы. Дополнительная информация приведена в 13.1.5.

- vacuum_freeze_min_age (integer) определяет возраст отключения в транзакциях, который должен использоваться командой VACUUM для решения о замене идентификаторов транзакций на FrozenXID при сканировании таблицы. Значение по умолчанию пятьдесят миллионов транзакций. Хотя, пользователи могут указать любое значение от 0 до одного миллиарда, VACUUM негласно

ограничивает эффективное значение параметра половиной значения параметра autovacuum_freeze_max_age, чтобы между принудительными операциями автовакуумингa не был неоправданно малый промежуток времени. Дополнительная информация приведена в 13.1.5.

- bytea_output (enum) устанавливает формат вывода значений типа bytea. Допустимые значения hex (по умолчанию) и escape (традиционный для PostgreSQL). Дополнительная информация приведена в 5.4. Тип bytea всегда принимает оба формата независимо от значений этого параметра.

- xmlbinary (enum) определяет правила кодирования бинарных значений в XML. Значение параметра применяется, например, при конвертировании данных типа bytea конвертируются в XML функциями xmlelement или xmlforest. Возможные значения base64 и hex определены в стандарте XML Schema. Значение по умолчанию base64. Дополнительная информация приведена в 6.14.

Актуальность выбора ограничивается только возможными запретами в клиентских приложениях. Оба метода поддерживают все возможные значения, хотя запись в шестнадцатеричной кодировке (hex) будет немного больше, чем в кодировке base64.

- xmloption (enum) указывает, что подразумевается DOCUMENT или CONTENT при конвертировании из XML в строку символов и наоборот. Дополнительная информация приведена в 5.13. Значение параметра может быть DOCUMENT или CONTENT. Значение параметра по умолчанию CONTENT.

В соответствии со стандартом SQL значение параметра изменяется следующей командой:

```
SET XML OPTION { DOCUMENT | CONTENT };
```

Данная форма записи доступна и в СУБД PostgreSQL.

8.11.2. Параметры локализации и форматирования

- DateStyle (string) определяет формат отображения даты и времени и правила интерпретирования неоднозначных входных значений даты. Исторически сложилось так, что параметр содержит два независимых компонента: спецификация формата вывода (ISO, Postgres, SQL или German) и спецификация порядка ввода вывода для года, месяца, дня (DMY, MDY или YMD). Значения компонентов могут быть заданы отдельно или совместно. Ключевые слова Euro и European являются синонимами DMY, ключевые слова US, NonEuro и NonEuropean являются синонимами для MDY. Дополнительная информация приведена в 5.5. Встроенные значения по умолчанию ISO, MDY, но initdb инициализирует конфигурационный файл, в котором значения этого параметра соответствуют локализации, указанной в

lc_time.

- IntervalStyle (enum) определяет формат отображения для интервальных значений. Значение sql_standard определяет, что выдаваемые интервальные символы будут соответствовать стандарту SQL. Значение по умолчанию postgres определяет, что отображение будет соответствовать PostgreSQL до версии 8.4, при значение ISO параметра DateStyle. Значение postgres_verbose определяет, что отображение будет соответствовать PostgreSQL до версии 8.4, при значении параметра DateStyle, определяющем не ISO-формат отображения. Значение iso_8601 соответствует отображению интервалов времени в формате с обозначениями («format with designators»), определённом в пункте 4.4.3.2 стандарта ISO 8601.

Параметр IntervalStyle определяет интерпретацию неоднозначных вводимых интервалов. Дополнительная информация приведена в 18.

- TimeZone (string) определяет часовой пояс для отображения и интерпретирования временных пометок. Значение по умолчанию unknown определяет использованного часовой пояс, установленного в системном окружении. Дополнительная информация приведена в 5.5.3.

- timezone_abbreviations (string) определяет набор аббревиатур часовых поясов, которые будут приниматься сервером для ввода даты и времени. Значение по умолчанию 'Default' определяет набор, используемый практически по всему миру, кроме того существую наборы 'Australia' и 'India' и возможность определения других наборов для особой конфигурации.

- extra_float_digits (integer) определяет количество разрядов, отображаемых для чисел с плавающей точкой, включая float4, float8 и геометрические типы данных. Значение параметра добавляется к стандартному количеству разрядов (FLT_DIG или DBL_DIG). Можно установить значение не больше 3 для включения частично значимых разрядов. Использование параметра особенно полезно для дампа (выгрузки) данных с плавающей точкой, которые должны быть восстановлены точно. Для устранения лишних разрядов устанавливается отрицательное значение параметру.

- client_encoding (string) определяет кодировку (набор символов) для клиента. По умолчанию используется кодировка БД.

- lc_messages (string) определяет язык вывода сообщений. Допустимые значения зависят от операционной системы. Дополнительная информация приведена в 12.1. Пустая строка, являющаяся значением по умолчанию определяет, что значение наследуется от окружения исполнения сервера методом, зависящим от операцион-

ной системы.

В системах не поддерживающих локализацию установка значения параметра не меняет язык вывода сообщений. При отсутствии сообщений, переведенных на используемый язык локализации, сообщения будут выводиться на английском языке. Значение параметра могут изменять только суперпользователи, поскольку параметр определяет не только язык сообщений, отправляемых клиенту, но и язык сообщений, записываемых в журнал.

- `lc_monetary (string)` определяет локализацию, используемую для форматирования неоднозначных денежных значений, например, для семейства функций `to_char`. Допустимые значения зависят от операционной системы. Дополнительная информация приведена в 12.1. Пустая строка, являющаяся значением по умолчанию определяет, что значение наследуется от окружения исполнения сервера методом, зависящим от операционной системы.

- `lc_numeric (string)` определяет локализацию, используемую для форматирования чисел, например, для семейства функций `to_char`. Допустимые значения зависят от операционной системы, подробнее см. Дополнительная информация приведена в 12.1. Пустая строка, являющаяся значением по умолчанию определяет, что значение наследуется от окружения исполнения сервера методом, зависящим от операционной системы.

- `lc_time (string)` определяет локализацию, используемую для форматирования значений даты и времени, например, для семейства функций `to_char`. Допустимые значения зависят от операционной системы, Дополнительная информация приведена в 12.1. Пустая строка, являющаяся значением по умолчанию определяет, что значение наследуется от окружения исполнения сервера методом, зависящим от операционной системы.

- `default_text_search_config (string)` определяет конфигурацию текстового поиска, используемую функциями текстового поиска, не имеющими явного аргумента, определяющего названную конфигурацию. Встроенное значение по умолчанию `pg_catalog.simple`, но `initdb` инициализирует конфигурацию с значением параметра, соответствующим заданному значению `lc_ctype`, при условии, конфигурация идентифицирована как соответствующая заданной локализации.

8.11.3. Загрузка разделяемых библиотек

Некоторые настройки используются для загрузки разделяемых библиотек на сервер, для того, чтобы загрузить дополнительную функциональность или дополнительные преимущества в производительности. Например, установка `$libdir/mylib` повлечет загрузку библиотеки `mylib.so`(или на других платформах `mylib.sl`) из стандартной директории

библиотек. Разница между настройками заключается в том, когда они применяются и какие привилении требуются для их изменения.

Библиотеки процедурных языков уже загружены, обычно используя синтакс `$libdir/plXXX`, где `XXX` может быть `pgsql`, `perl`, `tcl` или `python`.

Для каждого параметра необходимо разделять запятой их имена, если более одной библиотеки загружено. Все имена библиотек должны быть записаны в нижнем регистре, если они не заключены в двойные кавычки.

Каждая разделяемая библиотека должна быть специальным образом подготовлена для загрузки. Каждая библиотека PostgreSQL имеет «магический блок», который проверяется сервером для гарантии соответствия. По этой причине не совместимые с PostgreSQL библиотеки не будут загружены таким образом. Вам должно быть разрешено использовать средства операционной системы, такие как `LD_PRELOAD` для этого.

В общих случаях обратитесь к документации конкретного модуля для рекомендаций по загрузке.

- `local_preload_libraries (string)` определяет одну или несколько разделяемых библиотек, которые будут загружены при запуске процесса сервера, обслуживающего сессию. Для указания набора библиотек их имена перечисляются через запятую. Значение параметра не может быть изменено после начала сессии. Так как пользователь может изменять значение параметра, то могут быть указаны только библиотеки из директории `plugins`, находящейся в стандартной директории библиотек СУБД. При установке значения параметра `local_preload_libraries` директория `plugins` может быть указана явно (например, `$libdir/plugins/mylib`) или может быть использовано только имя библиотеки `mylib`.
- `session_preload_libraries (string)` определяет одну или более одной разделяемой библиотек, которые будут загружены при подключении. Только суперпользователи могут изменить эту настройку. Значение параметра окажет влияние только при старте соединения. Последующие изменения не дадут никакого эффекта. Если указанная библиотека не найдена, подключение будет завершено с ошибкой. Данная возможность позволяет проводить отладку или библиотеки для проведения измерений без использования специфической команды `LOAD` в сессии. Например, `auto_explain` может быть загружена для всех сессий пользователя, указав эту библиотеку в параметре команде `ALTER ROLE SET`. Также этот параметр может быть изменен без перезапуска сервера (но изменения применятся только при старте новой сессии), поэтому просто добавлять новые модули, даже если они будут применены ко всем сессиям. В отличие от `shared_preload_libraries` нет преимуществ производительности при загрузки библиотеки в начале сессии, а не

когда она впервые использована. Однако, есть преимущество, когда библиотека используется в пуле соединений.

Примечание. Данный параметр используется только в СУБД версии 9.6.

- shared_preload_libraries (string) определяет одну или несколько библиотек, предварительно загружаемых при запуске сервера. Для предварительной загрузки нескольких библиотек они перечисляются через запятую. Например, при значении параметра '$libdir/mylib' будет предварительно загружена библиотека mylib.so (или на некоторых платформах mylib.sl) из стандартной директории библиотек. Значение параметра может быть задано только при запуске сервера. Подобным образом могут быть предварительно загружены библиотеки процедурных языков PostgreSQL. Обычно используется формат записи '$libdir/plXXX', где вместо XXX указывается pgsql, perl, tcl или python. Предварительная загрузка разделяемой библиотеки позволяет избежать ожидания запуска библиотеки при ее первом использовании. Однако, время запуска процесса каждого нового сервера может незначительно увеличиться, даже если процесс не использует библиотеку. Таким образом, установка значения параметра рекомендуется только для тех библиотек, используемых большинством сессий. Если указанная библиотека не будет найдена, сервер не запустится. В каждой библиотеке, поддерживаемой PostgreSQL, есть «магический блок», который проверяется для обеспечения гарантий совместимости. Таким образом, не PostgreSQL-библиотеки не могут быть загружены рассмотренным способом.

8.11.4. Другие значения по умолчанию

- dynamic_library_path (string) определяет пути для поиска файлов динамически загружаемых модулей при указании имени файла, заданного в команде CREATE FUNCTION или LOAD, без директории (то есть имя файла не содержит косой черты).

Значение dynamic_library_path представляет собой список перечисленные через двоеточие абсолютных путей к директориям. Если элемент списка начинается со специальной строки $libdir, указанная при компилировании PostgreSQL директория для библиотек подставляется вместо $libdir. В названную директорию устанавливаются модули, входящие в стандартную поставку PostgreSQL. Имя директории можно установить с помощью следующей команды pg_config --pkglibdir). Например:

```
dynamic_library_path =
 '/usr/local/lib/postgresql:/home/my_project/lib:$libdir'
```

Значение параметра по умолчанию $libdir. Использование пустой строки в каче-

стве значения параметра отключает автоматический поиск по путям.

Значение параметра может быть изменено суперпользователем во время работы, но установленное подобным способом значение будет существовать только до завершения соединения с клиентом. Данный способ установки значения параметра должен использоваться только при разработке программного обеспечения. Рекомендуется задавать значение параметра в конфигурационном файле `postgresql.conf`.

- `gin_fuzzy_search_limit (integer)` определяет не жесткий верхний предел для размера набора, возвращаемого индексом `GIN`.

- `local_preload_libraries (string)` определяет одну или несколько разделяемых библиотек, которые будут загружены при запуске процесса сервера, обслуживающего сессию. Для указания набора библиотек их имена перечисляются через запятую. Значение параметра не может быть изменено после начала сессии.

Так как пользователь может изменять значение параметра, то могут быть указаны только библиотеки из директории `plugins`, находящейся в стандартной директории библиотек СУБД. При установке значения параметра `local_preload_libraries` директория `plugins` может быть указана явно (например, `$libdir/plugins/mylib`) или может быть использовано только имя библиотеки `mylib`.

В отличии от `shared_preload_libraries` загрузка библиотек при открытии сессии, а не при первом использовании не обеспечивает выигрыш в производительности. Параметр можно применять для загрузки в определенных сессиях отладочных библиотек или библиотек, измеряющих производительность, без дополнительного использования команды `LOAD`. Например, можно включить отладку для всех сессий данного пользователя, указав параметру значение при помощи команды `ALTER USER SET`.

Если указанная библиотека не найдена, соединение не будет установлено.

В каждой библиотеке, поддерживаемой PostgreSQL содержится системный блок, наличие которого проверяется для гарантирования совместимости. Данное обстоятельство обуславливает невозможность использования рассмотренного способа для загрузки не PostgreSQL библиотек.

8.12. Управление блокировками

- `deadlock_timeout (integer)` определяет в миллисекундах время ожидания блокировки до проверки наличия взаимной блокировки (deadlock). Поскольку проверка на наличие взаимной блокировки выполняется относительно медленно, то сервер производит ее не при каждом ожидании блокировки. Наличие взаимной бло-

кировки не является нормой при создании приложений поэтому значение параметра используется для ожидания блокировки в течение указанного времени пред запуском проверки на наличие взаимной блокировки. Увеличение значения сокращает время, затрачиваемое на ненужные проверки взаимной блокировки, но при этом замедляет появление отчетов об ошибках взаимной блокировки. Значение по умолчанию одна секунда является минимально возможным применимым на практике значением. Для сервера СУБД с большой нагрузкой рекомендуется увеличить значение параметра. Идеальное значение параметра должно превышать среднее время транзакции для увеличения вероятности освобождения блокировки до принятия решения о выполнении проверки на наличие взаимной блокировки.

Установка значения параметра `log_lock_waits` определяет и время ожидания перед добавлением в журнал сообщения об ожидании блокировки. Для проведения исследования задержек блокировок, возможно, следует указать более меньшее чем обычно значение параметра `deadlock_timeout`.

- `max_locks_per_transaction (integer)` определяет среднее число блокировок объектов, выделенное для каждой транзакции. Для отслеживания блокировок объектов (например, таблиц) создается разделяемая таблица блокировок, рассчитанная на количество объектов, вычисляемое по следующей формуле:

`max_locks_per_transaction * (max_connections + max_prepared_transactions)`

Следовательно одновременно может быть заблокировано не более определенного количества различных объектов. Отдельные транзакции могут блокировать большее количество объектов при условии, что блокировки всех транзакций поместились в таблицу блокировок. Значение параметра не ограничивает число строк, которые могут быть заблокированы. Опыт практического применения СУБД показал, что значение по умолчанию `64` является достаточным. Однако при наличии клиентов, использующих в одной транзакции много различных таблиц, значение параметра рекомендуется увеличить. Значение параметра может быть задано только при запуске сервера.

Увеличение значения параметра `max_locks_per_transaction` может привести к превышению требований PostgreSQL к разделяемой памяти System V ограничений конфигурации операционной системы по умолчанию. Дополнительная информация приведена по настройке параметров использования разделяемой памяти приведена в 7.4.1.

При запуске резервного сервера параметр должен быть установлен в то же или большее значение, чем на основном. В противном случае, запросы на резервном сервере исполняться не будут.

- max_pred_locks_per_transaction (integer) определяет среднее число условных блокировок объектов, выделенное для каждой транзакции. Для отслеживания условных блокировок объектов (например, таблиц) создается разделяемая таблица блокировок, рассчитанная на количество объектов, вычисляемое по следующей формуле:

max_locks_per_transaction * (max_connections + max_prepared_transactions)

Следовательно одновременно может быть заблокировано не более определенного количества различных объектов. Отдельные транзакции могут блокировать большее количество объектов при условии, что блокировки всех транзакций поместились в таблицу блокировок. Значение параметра не ограничивает число строк, которые могут быть заблокированы. Опыт практического применения СУБД показал, что значение по умолчанию 64 является достаточным для тестирования. Однако при наличии клиентов, использующих в одной транзакции много различных таблиц, значение параметра рекомендуется увеличить. Значение параметра может быть задано только при запуске сервера.

8.13. Совместимость версий и платформ

8.13.1. Предыдущие версии PostgreSQL

- array_nulls (boolean) определяет необходимость распознавания анализатором входных данных массивов не заключенных в кавычки NULL как пустых элементов массива. Значение по умолчанию on (включено) позволяет вводить массивы, содержащие пустые элементы. Однако, PostgreSQL версии до 8.2 не поддерживает пустые значения в массивах, и, следовательно, воспринимает NULL как нормальный элемент массива типа string со значением 'NULL'. Для обеспечения обратной совместимости с приложениями, разработанными с учетом поведения PostgreSQL версии до 8.2 необходимо использовать значение параметра off.

Следует отметить, что существует возможность для создания пустых элементов массива даже при значении параметра off.

- backslash_quote (enum) определяет возможность представления одинарной кавычки в виде \'. Предпочтительным способом представления, соответствующим стандарту SQL является удвоение кавычки '', но по сложившейся традиции PostgreSQL принимает также и \'. Однако, использование \' приводит к возникновению риска для безопасности информации, поскольку в некоторых клиентских кодировках существуют многобайтовые символы, в которых последний байт численно эквивалентен \ в ASCII. В случае некорректного выполнения кода на стороне клиента возникает возможность проведения атаки типа SQL-внедрение. Возможно предотвратить возникновение риска, заставив сервер не принимать запросы, в которых кавычка

экранируется символом \. Допустимые значения параметра backslash_quote: on (всегда разрешать \'), off (всегда запрещать) и safe_encoding (разрешать только если в клиентской кодировке не допускается ASCII \ в многобайтовых символах). Значение по умолчанию safe_encoding.

Следует отметить, что в соответствии стандарту строковый литерал \ означает только \. Параметр влияет только на обработку строковых литералов, не соответствующих стандарту, включая синтаксис строк управляющих последовательностей для экранирования E'...'.

- default_with_oids (boolean) определяет необходимость включения командами CREATE TABLE и CREATE TABLE AS столбца OID в новую таблицу, если не указано ни WITH OIDS, ни WITHOUT OIDS. Кроме того, параметр определяет необходимость включения столбца OID в таблицу, созданную командой SELECT INTO. Значение параметра default_with_oids по умолчанию off (выключен), в более ранних версиях значение параметра по умолчанию было on (включен).

Использование OID в таблицах пользователей считается нежелательным, поэтому в большинстве конфигураций значение параметра должно оставаться off (выключен). В приложениях, которым требуются OID для определенных таблиц, необходимо указывать WITH OIDS при создании таблиц. Значение параметра on (включен) может быть установлено для обеспечения совместимости со старыми приложениями, которые не соответствуют описанному поведению.

- escape_string_warning (boolean) определяет необходимость выдачи предупреждения при появлении обратного слэша \ в обычном строковом литерале (запись вида '...') и значении параметра standard_conforming_strings off (выключен). Значение параметра по умолчанию on (включен).

Приложения, которые используют обратный слэш для экранирования, должны быть модифицированы так, чтобы использовать формат экранирования E'...', поскольку поведение обычных строк по умолчанию будет изменено в последующих версиях для обеспечения совместимости с SQL. Параметр может быть полезен для выявления приложений, которые потеряют работоспособность.

- lo_compat_privileges (boolean) управляет использованием привилегий доступа к большим объектам. До версии PostgreSQL 9.0 большие объекты не имели собственных прав доступа и были доступны на чтение и запись для всех пользователей. Установка значения on выключает использование прав доступа к большим объектам для совместимости с предыдущими версиями. Значение по умолчанию off. Значение параметра могут изменять только суперпользователи.

Установка этого параметра не отключает все проверки доступа к большим объектам,

а только те, поведение для которых было изменено в PostgreSQL 9.0. К примеру, lo_import() и lo_export() требуют привилегий суперпользователя независимо от значения этого параметра.

- quote_all_identifiers (boolean) определяет использование заключения в кавычки идентификаторов при генерировании SQL, даже если они не являются ключевыми словами. Этот параметр действует на результат вывода команды EXPLAIN, функций типа pg_get_viewdef. Аналогичным является использование опции --quote-all-identifiers утилит pg_dump и pg_dumpall.

- sql_inheritance (boolean) определяет семантику наследования. Значение по умолчанию on, при этом в результат запроса включаются строки порожденных таблиц (как если бы был указан суффикс *). Если указано значение off, то по умолчанию порожденные таблицы не включаются при выполнении различных команд, подразумевая наличие ключевого слова ONLY. Стандарт SQL требует включения порожденных таблиц; эта опция была добавлена для совместимости с версиями до 7.1. Дополнительная информация приведена в 2.8.

Выключение sql_inheritance считается устаревшим, поскольку такое поведение чревато ошибками и в отличие от требований SQL стандарта. Обсуждение наследования в этом документе подразумевает значение этого параметра on.

- standard_conforming_strings (boolean) определяет, воспринимают ли обычные строковые литералы '...' знак обратного слэша буквально, как указано в стандарте SQL. Начиная с версии PostgreSQL 9.1, значение по умолчанию on (в предыдущих было off). Приложения могут проверять значение параметра, чтобы определить, как следует обрабатывать строковые литералы. Значение параметра on может считаться индикатором поддержки синтаксиса строки управляющей последовательности для экранирования E'...'. Синтаксис строки управляющей последовательности для экранирования E'...' должен быть использован в случае, когда приложение должно воспринимать обратный слэш как символ управляющей последовательности.

- synchronize_seqscans (boolean) разрешает синхронизировать последовательные операции чтения больших таблиц, чтобы конкурирующие операции читали одинаковый блок примерно в одно и то же время, разделяя рабочую нагрузку ввода-вывода. Если установлено значение параметра on, то, для синхронизации с действиями уже выполняющихся процессов просмотра, чтение может начаться с середины таблицы, а затем, в конце таблицы циклически, возвращается для обработки всех строк. Подобное поведение может привести к непредсказуемым изменениям в порядке строк, возвращаемых запросами, в которых не указано ORDER BY. Установка

значения параметра off гарантирует поведение PostgreSQL аналогичное версиям до 8.3, при котором следующее чтение всегда будет начинаться только с начала таблицы. Значение по умолчанию on.

8.13.2. Совместимость платформ и клиентов

- transform_null_equals (boolean) — при значении параметра on (включен), выражения вида expr = NULL (или NULL = expr) воспринимаются как expr IS NULL, то есть возвращают true, если выражение имеет пустое значение, и false, если это не так. В соответствии со стандартом SQL, выражение вида expr = NULL всегда должно возвращать пустое значение (unknown). Следовательно, значение параметра по умолчанию off.

Однако, фильтрованные формы в Microsoft Access генерируют запросы, которые используют выражение expr = NULL для проверки на пустые значения, поэтому если вы используете для доступа к БД этот интерфейс, возможно, вам понадобится включить эту опцию. Так как выражения вида expr = NULL всегда возвращают пустое значение (при использовании правильной интерпретации), они не являются очень полезными и не часто применяются в приложениях, таким образом на практике включение параметра приводит к небольшому ущербу. Но так как новых пользователей часто смущает семантика выражений, использующих пустые значения, по умолчанию эта опция отключена.

Следует отметить, что параметр влияет только на выражения вида = NULL, но не на другое сравнение операторов или другие выражения, которые являются вычислительным эквивалентом некоторых выражений, включающих оператор равенства (таких как IN). Таким образом, параметр не является универсальным средством защиты от ошибок программирования. Дополнительная информация приведена в 6.2.

8.14. Обработка ошибок

- exit_on_error (boolean) управляет прерыванием сессии клиента в случае возникновения ошибок. При значении параметра TRUE (включен), любая ошибка прерывает сессию клиента. По умолчанию задано значение FALSE, при котором только фатальные (FATAL) ошибки прерывают сессию клиента.

- restart_after_crash (boolean) задает автоматически перезапуск PostgreSQL после процесса обработки. Значение параметра по умолчанию TRUE (включен). Такое значение является нормой, обеспечивая максимальную доступность сервера. Однако, в некоторых случаях, таких как использование PostgreSQL в кластерных решениях, может быть полезным запрещение перезапуска,

что бы механизмы кластеризации могли работать соответствующим образом.

8.15. Предустановленные параметры

Следующие параметры доступны только для чтения, и могут быть определены при компиляции PostgreSQL или при его установке. Поэтому, они были исключены из файла `postgresql.conf`. Эти параметры определяют различные аспекты поведения СУБД PostgreSQL, в которых могут заинтересованы определенные приложения, в особенности внешние административные приложения.

- `block_size (integer)` сообщает размер блока на диске. При сборке сервера этот размер определяется значением переменной `BLCKSZ`. Значение по умолчанию составляет `8192` байта. Параметр `block_size` влияет на значение некоторых конфигурационных переменных (таких, как `shared_buffers`). Дополнительная информация приведена в 8.4.

- `data_checksums (boolean)` сообщает, используются ли контрольные суммы для этого кластера (см. 19.1).

Примечание. Данный параметр используется только в СУБД версии 9.6.

- `integer_datetimes (boolean)` сообщает, была ли произведена сборка PostgreSQL с поддержкой 64-битных целочисленных времени и даты. Параметр может быть отключен, посредством использования при сборке PostgreSQL опции `--disable-integer-datetimes`. Значение по умолчанию `on`.

- `lc_collate (string)` сообщает локализацию, в которой производится сортировка текстовых данных. Дополнительная информация приведена в 12.1. Значение определяется при инициализации кластера БД.

- `lc_ctype (string)` сообщает локализацию, которая определяет классификацию символов. Дополнительная информация приведена в 12.1. Значение определяется при инициализации кластера БД. Обычно, оно совпадает со значением `lc_collate`, но для некоторых приложений может быть установлено другое значение.

- `max_function_args (integer)` сообщает максимальное число аргументов функции. Оно задается при сборке сервера значением переменной `FUNC_MAX_ARGS`. Значение по умолчанию `100`.

- `max_identifier_length (integer)` сообщает максимальную длину идентификатора, определяемую при сборке, как значение, на единицу меньшее значения переменной `NAMEDATALEN`. По умолчанию переменная `NAMEDATALEN` имеет значение `256`, следовательно, значение по умолчанию для `max_identifier_length` составляет `255` байта.

- max_index_keys (integer) сообщает максимальное число ключей индекса. Оно определяется при сборке сервера значением переменной INDEX_MAX_KEYS. Значение по умолчанию 32.

- segment_size (integer) сообщает число блоков (страниц), которые могут храниться в файловом сегменте. Определяется при сборке сервера значением переменной RELSEG_SIZE. Максимальный размер файла сегмента в байтах соответствует segment_size, умноженному на block_size. По умолчанию 1 ГБ.

- server_encoding (string) сообщает кодировку (набор символов) БД, задаваемую при создании БД. Обычно клиентам необходимо использовать значение параметра client_encoding.

- server_version (string) сообщает номер версии сервера. Он определяется при сборке сервера значением переменной PG_VERSION.

- server_version_num (integer) сообщает целочисленный номер версии сервера. Он определяется при сборке сервера значением переменной PG_VERSION_NUM.

- wal_block_size (integer) сообщает размер блока WAL на диске. Он определяется при сборке сервера значением переменной XLOG_BLCKSZ. Значение по умолчанию 8192 байта.

- wal_segment_size (integer) сообщает число блоков (страниц) в файле сегмента WAL. Полный объем файла сегмента WAL в байтах равен значению параметра wal_segment_size, умноженному на значение параметра wal_block_size. Значение по умолчанию 16 МБ.

8.16. Дополнительные опции

Данная возможность была предоставлена для добавления параметров, которые не являются стандартными для СУБД PostgreSQL, в дополнительных модулях (таких как процедурные языки). Она позволяет конфигурировать дополнительные модули стандартными методами.

Дополнительные опции имеют составное имя: имя дополнительного модуля (расширения), затем точка и имя параметра, согласно синтаксису имен SQL, например plpgsql.variable_conflict.

Поскольку дополнительные опции должны быть установлены до загрузки соответствующего дополнительного модуля, PostgreSQL принимает любые параметры с составными именами. Подобные переменные рассматриваются как метки и не действуют до загрузки определившего их модуля. При загрузке такого модуля, он добавляет определение своих переменных, конвертируя метки в соответствующие значения согласно этим определениям,

и выдавая предупреждения для всех не распознанных меток, начинающихся с названия модуля.

8.17. Параметры для разработчиков

Следующие параметры предназначены для работы с исходными текстами СУБД PostgreSQL и в некоторых случаях для помощи при восстановлении серьезно поврежденных БД. Необходимость в использовании данных параметров в рабочей конфигурации СУБД отсутствует, поэтому параметры были исключены из примера файла `postgresql.conf`. Следует отметить, что многие параметры требуют включения специальных флагов при компилировании исходных текстов.

- `allow_system_table_mods (boolean)` разрешает модификацию структуры системы таблиц. Используется в `initdb`. Значение параметра может быть задано только при запуске сервера.
- `debug_assertions (boolean)` включает проверку различных операторов контроля (логических выражений, которые предполагаются истинными). Использование параметра полезно при отладке, для преодоления неизвестных проблем или сбоев, так как она помогает выявить ошибки программирования. Для использования параметра, при сборке PostgreSQL посредством конфигурационной опции `--enable-cassert` должен быть определен макрос `USE_ASSERT_CHECKING`. Следует отметить, что параметр `debug_assertions` по умолчанию имеет значение `on`, если PostgreSQL был собран с включенной проверкой операторов контроля.
- `ignore_system_indexes (boolean)` параметр позволяет игнорировать системные индексы при чтении системных таблиц (но при этом обновлять индексы при изменении таблиц). Параметр полезен при восстановлении поврежденных системных индексов. Значение параметра не может быть изменено после начала сессии.
- `post_auth_delay (integer)` определяет задержку в секундах при запуске нового процесса сервера после завершения процедуры аутентификации. Параметр предоставляет возможность подсоединить отладчик к процессу сервера. Значение параметра не может быть изменено после начала сессии.
- `pre_auth_delay (integer)` определяет задержку в секундах сразу после запуска нового процесса сервера перед проведением процедуры аутентификации. Параметр предоставляет возможность подсоединить отладчик к процессу сервера для отслеживания возможных ошибок при аутентификации. Значение параметра может быть задано только в файле `postgresql.conf` или в командной строке сервера.
- `trace_notify (boolean)` генерирует большое количество отладочной информа-

ции для команд LISTEN и NOTIFY. Для перенаправления данной информации в журнал клиента или сервера, уровень client_min_messages или log_min_messages, соответственно, должен быть DEBUG1 или ниже.

- trace_recovery_messages (enum) разрешает вывод в журнал отладочной информации, относящейся к восстановлению. Параметр разрешает пользователю переопределять нормальные установки параметра log_min_messages, но только для определенных сообщений. Это введено для использования при отладке сервера горячей замены. Доступные значения DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1 и LOG. По умолчанию установлено значение LOG, не изменяющее штатное поведение. Другие значения обеспечивают вывод относящейся к восстановлению отладочной информации с заданным приоритетом и выше, чем на уровне LOG. Значение этого параметра может быть задано только в файле postgresql.conf или в командной строке сервера.

- trace_sort (boolean) включение параметра разрешает выдачу информации об использовании ресурсов при выполнении операций сортировки. Параметр доступен, только если при компилировании PostgreSQL был определен макрос TRACE_SORT. В настоящее время TRACE_SORT определен по умолчанию.

- trace_locks (boolean) включение параметра разрешает выдачу информации об использовании блокировок. Информация включает в себя тип блокированной операции, тип блокировки и уникальный идентификатор заблокированного или разблокированного объекта. Кроме того, включается информация о битовых масках тех типов блокировок, которые уже применялись к этому объекту, и тех, которые ожидают применения к нему. Для каждого типа блокировки также добавляется число примененных блокировок, блокировок, ожидающих применения, и общее число. Далее приведен пример результирующего файла журнала.

```
%  LOG: LockAcquire: new: lock(0xb7acd844) id(24688,24696,0,0,0,1)
    grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
    wait(0) type(AccessShareLock)
%  LOG: GrantLock: lock(0xb7acd844) id(24688,24696,0,0,0,1)
    grantMask(2) req(1,0,0,0,0,0,0)=1 grant(1,0,0,0,0,0,0)=1
    wait(0) type(AccessShareLock)
%  LOG: UnGrantLock: updated: lock(0xb7acd844) id(24688,24696,0,0,0,1)
    grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
    wait(0) type(AccessShareLock)
%  LOG: CleanUpLock: deleting: lock(0xb7acd844) id(24688,24696,0,0,0,1)
    grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
    wait(0) type(INVALID)
```

Подробная информация о структуре, которая используется для сохранения в дамп, содержится в src/include/storage/lock.h.

Параметр доступен только если при компилировании PostgreSQL был определен макрос LOCK_DEBUG.

- trace_lwlocks (boolean) включение параметра разрешает выдачу информации об использовании упрощенных блокировок. Упрощенные блокировки предназначены в первую очередь для взаимного исключения доступа к данным в разделяемой памяти. Параметр доступен только если при компилировании PostgreSQL был определен макрос LOCK_DEBUG.

- trace_userlocks (boolean) включение параметра разрешает выдачу информации об использовании пользовательских блокировок. Выходные данные такие же, как для trace_locks, но только для пользовательских блокировок. В PostgreSQL начиная с версии 8.2 пользовательские блокировки отсутствуют. В настоящее время использование параметра ни на что не влияет.

Параметр доступен только если при компилировании PostgreSQL был определен макрос LOCK_DEBUG.

- trace_lock_oidmin (integer) параметр определяет, что для таблиц с идентификатором объекта ниже указанного в значении параметра OID отключается отслеживание блокировок для таблиц. Параметр можно использовать для предотвращения вывода информации по системным таблицам.

Параметр доступен только если при компилировании PostgreSQL был определен макрос LOCK_DEBUG.

- trace_lock_table (integer) параметр определяет безусловное отслеживание блокировок для таблицы с указанным в значении параметра идентификатором объекта OID.

Параметр доступен только если при компилировании PostgreSQL был определен макрос LOCK_DEBUG.

- debug_deadlocks (boolean) при включении параметра сохраняется вся информация о текущих блокировках на момент возникновения таймаута взаимной блокировки.

Параметр доступен только если при компилировании PostgreSQL был определен макрос LOCK_DEBUG.

- log_btree_build_stats (boolean) при включении параметра статистика использования системных ресурсов (память и процессор) при различных btree-операциях будет заноситься в журнал. Параметр доступен только если при компилировании PostgreSQL был определен макрос BTREE_BUILD_STATS.

- wal_debug (boolean) при включении параметра выдается отладочная информация, касающаяся WAL.

Параметр доступен только если при компилировании PostgreSQL был определен макрос WAL_DEBUG.

- zero_damaged_pages (boolean) включение параметра определяет, что при обнаружении поврежденного заголовка страницы СУБД PostgreSQL вместо сообщения об ошибке, прервавшей выполнение текущей команды, будет выдавать предупреждение, обнулять поврежденную страницу и продолжать выполнение. Подобное поведение приведет к потере данных, а именно, все строки поврежденной страницы. Однако включение параметра позволит вам обойти ошибку и получить строки из любых неповрежденных страниц, которые могут быть в таблице. Использование параметра полезно при восстановлении данных после повреждения, вызванного ошибкой аппаратного или программного обеспечения. В общем случае не следует включать параметр при наличии надежды на восстановление данных с поврежденных страниц таблицы. Значение параметра по умолчанию off, и может быть изменено только суперпользователем.

8.18. Короткие параметры

Для удобства использования для некоторых параметров доступны однобуквенные ключи опций командной строки. В таблице приведено их описание. Некоторые из опций существуют по историческим причинам, и их присутствие в списке однобуквенных ключей не означает, что их необходимо обязательно использовать.

Таблица 105 – Однобуквенные ключи опций командной строки

Короткий ключ	Эквивалент
-A x	debug_assertions = x
-B x	shared_buffers = x
-d x	log_min_messages = DEBUGx
-e	datestyle = euro
-fb, -fh, -fi, -fm, -fn, -fo, -fs, -ft	enable_bitmapscan = off, enable_hashjoin = off, enable_indexscan = off, enable_mergejoin = off, enable_nestloop = off, enable_indexonlyscan = off, enable_seqscan = off, enable_tidscan = off
-F	fsync = off
-h x	listen_addresses = x
-i	listen_addresses = '*'
-k x	unix_socket_directory = x
-l	ssl = on

Окончание таблицы 105

Короткий ключ	Эквивалент
-N x	max_connections = x
-O	allow_system_table_mods = on
-p x	port = x
-P	ignore_system_indexes = on
-s	log_statement_stats = on
-S x	work_mem = x
-tpa, -tpl, -te	log_parser_stats = on, log_planner_stats = on, log_executor_stats = on
-W x	post_auth_delay = x

9. АУТЕНТИФИКАЦИЯ КЛИЕНТА

При попытке соединения с сервером СУБД клиентское приложение указывает пользователя СУБД PostgreSQL, от имени которого осуществляется подключение, аналогично тому, как пользователь выполняет вход в ОС. В пределах окружения SQL активное имя пользователя СУБД определяет права на объекты БД (см. 10). Следовательно, важно ограничивать доступ пользователя на подключение к базам данных.

Примечание. В соответствии с 10 СУБД PostgreSQL осуществляет управление правами в терминах ролей. Далее будем использовать термин «пользователь СУБД» (или пользователь БД), подразумевая роль с привилегией `LOGIN`.

Аутентификацией называется процесс проверки сервером СУБД подлинности клиента и последующего определения наличия для клиентского приложения (для пользователя, который запускает клиентское приложение) права на соединение с БД от имени указанного пользователя БД.

PostgreSQL предлагает несколько различных методов аутентификации клиента. Метод, используемый для аутентификации конкретного клиентского соединения, может быть выбран на основе адреса узла сети клиента, БД и пользователя.

Имена пользователей СУБД PostgreSQL логически отделены от имен пользователей ОС, в которой запущен сервер. Если все пользователи сервера СУБД имеют учетные записи в ОС на серверной машине, целесообразно использовать имена пользователей БД, соответствующие именам пользователей ОС. Однако, сервер, принимающий удаленные соединения, может иметь много пользователей БД, для которых учетные записи в локальной ОС отсутствуют. В подобных случаях, сервер СУБД не осуществляет привязку имен пользователей БД к именам пользователей ОС.

ВНИМАНИЕ! СУБД PostgreSQL была доработана для обеспечения соответствия требованиям по защите информации от несанкционированного доступа. При реализации данных требований была установлена необходимость обеспечения соответствия пользователей СУБД учетным записям в ОС. Таким образом, при настройке аутентификации в СУБД следует использовать только методы аутентификации, в которых осуществляется подобное сопоставление. Для других пользователей осуществляется доступ только к незащищенной информации.

ВНИМАНИЕ! При использовании защищенного сервера СУБД из состава ОС в режиме мандатного управления доступом не допускается отключать аутентификацию субъектов доступа установкой в конфигурационном файле кластера `pg_hba.conf` режима `trust` (без аутентификации).

9.1. Файл конфигурации pg_hba.conf

Аутентификация клиентов настраивается в конфигурационном файле, который обычно называется pg_hba.conf и находится в каталоге данных кластера БД. Host-based authentication (HBA) – аутентификация на основе адресов узлов сети. По умолчанию файл pg_hba.conf устанавливается при инициализации каталога данных с помощью initdb. Однако, данный конфигурационный файл для аутентификации может быть размещен в другом месте (см. конфигурационный параметр hba_file).

Файл pg_hba.conf представляет собой набор записей, каждая из которых находится в отдельной строке. Пустые строки игнорируются, так же как и любой текст после символа #, обозначающего комментарии. Запись не может занимать более одной строки. Запись состоит из нескольких полей, разделенных пробелами и/или символами табуляции. Поле может содержать символ пробела при условии, что все значение поля заключено в кавычки. Заключение в кавычки любого ключевого слова в полях database, user, или address (например, all или replication) лишает их специального значения, и они рассматриваются как база данных, пользователь или компьютер с таким именем.

Каждая запись определяет тип соединения, диапазон IP-адресов клиента (когда это имеет значение для типа соединения), имя БД, имя пользователя и метод аутентификации, который следует использовать для соединений, соответствующих этим параметрам. При аутентификации используется первая запись с подходящим типом соединения, адресом клиента, запрашиваемой БД и именем пользователя. При этом возможность перехода через запись или возврата отсутствует: в случае когда выбрана какая-либо запись, и аутентификация не выполнена, прочие записи не рассматриваются. При отсутствии подходящий для аутентификации записи доступ запрещен.

Запись может иметь один из следующих семи форматов:

```
local      database  user  auth-method  [auth-options]
host       database  user  address  auth-method  [auth-options]
hostssl    database  user  address  auth-method  [auth-options]
hostnossl  database  user  address  auth-method  [auth-options]
host       database  user  IP-address  IP-mask  auth-method  [auth-options]
hostssl    database  user  IP-address  IP-mask  auth-method  [auth-options]
hostnossl  database  user  IP-address  IP-mask  auth-method  [auth-options]
```

Поля имеют следующие значения:

Запись local используется при попытке соединения через доменные сокеты Unix. Если нет записи такого типа, доступ через доменные сокеты Unix запрещен.

Запись host соответствует попытке установить соединение по TCP/IP. Записи host могут применяться как для соединений, использующих SSL, так и для не использующих

SSL.

Примечание. Удаленные соединения по TCP/IP возможны, только если при запуске сервера было указано соответствующее значение для конфигурационного параметра `listen_addresses`, т.к. по умолчанию сервер ожидает TCP/IP-соединения только на локальном адресе обратной связи `localhost`.

Запись `hostssl` соответствует попыткам соединения по TCP/IP с обязательным использованием SSL. Для применения данной опций необходимо, чтобы сервер был собран с поддержкой SSL. Кроме того, при запуске сервера необходимо разрешить использование SSL с помощью конфигурационного параметра `ssl` (см. 7.9).

Запись `hostnossl` реализует логику противоположную записи `hostssl` и соответствует соединениям по TCP/IP, не использующим SSL.

Поле `database` определяет, какие имена БД соответствуют этой записи. Значение `all` соответствует всем БД. Значение `sameuser` указывает, что запись подходит, если запрашиваемая БД имеет то же имя, что и запрашивающий пользователь. Значение `samerole` указывает, что запрашивающий пользователь должен быть членом роли с тем же именем, что и запрашиваемая БД (значение `samegroup` является устаревшим, но принимается как вариант написания `samerole`). Суперпользователь не рассматривается членом роли при использовании `samerole`, если он явным образом не определен членом заданной роли напрямую или косвенно. Значение `replication` указывает, что запись соответствует установке соединения для репликации (при установке соединения для репликации конкретная база данных не указывается). Кроме того, в значении может быть указано имя определенной БД PostgreSQL. Может быть указано несколько имен БД, перечисленных через запятую. Может быть указан отдельный файл, содержащий имена БД, посредством предшествующего имени файла символа `@`.

Поле `user` определяет имена пользователей БД, соответствующих данной записи. Значение `all` определяет, что запись действует для всех пользователей СУБД. В других случаях указывается имя определенного пользователя БД или имя группы, предваряемое символом +. Следует отметить, что в PostgreSQL не существует реального отличия между пользователями и группами. Символ + в действительности определяет соответствие записи любым ролям, которые прямо или опосредованно являются членами данной роли, в то время как имя без символа + означает соответствие записи только данной конкретной роли. Суперпользователь не рассматривается членом указанной роли, если он явным образом не определен членом заданной роли напрямую или косвенно. Может быть указано несколько имен пользователей, перечисленных через запятую. Может быть указан отдельный файл, содержащий имена пользователей, посредством предшествующего имени файла символа `@`.

Поле address определяет адреса клиентских машин, соответствующие данной записи. Поле может содержать имя машины, диапазон IP-адресов или одно из описанных далее ключевых слов.

ВНИМАНИЕ! В настоящее время отсутствует поддержка протокола IPv6. Задаваемые в конфигурационном файле pg_hba.conf адреса должны соответствовать протоколу IPv4.

IP-адрес задается в стандартной нотации, с использованием разделенных точками десятичных чисел, и длины маски CIDR. Длина маски показывает число старших битов, которые должны совпадать с соответствующими битами IP-адреса клиента. Биты в IP-адресе правее битов, определенных маской, должны быть нулевыми. Не допускается наличие любых пробелов между IP-адресом, символом / и длиной маски CIDR. Далее приведены типичные примеры значений поля CIDR-address.

Примеры:

1. 172.20.143.89/32 — значение поля для конкретного узла сети;
2. 172.20.143.0/24 — значение поля для сети класса C;
3. 10.6.0.0/16 — значение поля для сети класса A;
4. 0.0.0.0/0 — значение поля для всех адресов IPv4;
5. ::/0 — значение поля для всех адресов IPv6.

Для задания конкретного узла сети необходимо указывать в маске CIDR значение 32 для формата IPv4 или 128 для формата IPv6. Пропуск в сетевых адресах завершающих нулей не допускается.

IP-адрес, заданный в формате IPv4, будет подходить и для соединений с соответствующим адресом в формате IPv6. Например, значение 127.0.0.1 соответствует адресу в формате IPv6 ::ffff:127.0.0.1. Значение, заданное в формате IPv6, будет подходить только для соединений с адресами в формате IPv6, даже если представленный адрес содержится в диапазоне IPv4-в-IPv6. Следует отметить, что значения поля в формате IPv6 будут отброшены в случае, когда системная библиотека C не поддерживает формат адресов IPv6.

Также может быть использовано ключевое слово all для соответствия любых IP-адресов, samehost для соответствия адресов, совпадающих с адресом сервера, а samenet для соответствия адресов, находящихся в одной подсети с сервером.

При задании имени машины (любое значение, не являющееся IP-адресом или ключевым словом, считается именем) это имя сравнивается с результатом обратного разыменования IP-адреса клиента (например, с помощью DNS, если он используется). Сравнение имени машины не зависит от регистра символов. В случае совпадения производится прямое

разыменование (например, с помощью DNS) имени для проверки соответствия какого-либо из полученных IP-адресов адресу клиента. Если обе проверки выполняются, то запись считается подходящей. (Имя машины, заданное в pg_hba.conf, должно быть одним из имен, возвращаемых системой разыменования имен для IP-адреса клиента, в противном случае запись не будет считаться подходящей. Некоторые базы данных имен машин позволяют задать соответствие нескольких имен одному IP-адресу, но ОС возвращает только одно имя для заданного IP-адреса.)

Имя машины, начинающееся с точки (.) соответствует суффиксу реального имени. Таким образом, .example.com соответствует foo.example.com (а не только example.com).

При задании имен машин в pg_hba.conf необходимо, чтобы разыменование выполнялось быстро. Одним из решений может быть настройка локального кэша разыменования, такого как nscd. При необходимости может быть включен конфигурационный параметр log_hostname для вывода в журнал имени машины клиента вместо его IP-адреса.

Данное поле используется только в записях host, hostssl и hostnossl.

Поля IP-address и IP-mask используются как альтернатива нотации address. Вместо указания длины маски в отдельном столбце IP-mask задается действительная маска. Например, значение 255.0.0.0 поля IP-mask соответствует маске CIDR длиной 8 в формате IPv4, а значение 255.255.255.255 поля IP-mask соответствует маске CIDR длиной 32.

Данные поля используются только в записях host, hostssl и hostnossl.

Поле auth-method определяет метод аутентификации, используемый соединениями, соответствующими данной записи. Далее приведены возможные значения поля. Дополнительная информация приведена в 9.3.

- значение trust безусловно разрешает соединение. Данный метод позволяет любому, кто может соединиться с сервером СУБД PostgreSQL, входить под любой учетной записью СУБД PostgreSQL без пароля. Дополнительная информация приведена в 9.3.1;
- значение reject безусловно отвергает попытки соединений. Использование значения полезно при необходимости отфильтровать отдельные узлы сети из группы;
- значение md5 требует от клиента поддержки аутентификации с преобразованием пароля по алгоритму MD5. Дополнительная информация приведена в 9.3.2;
- значение password требует от клиента поддержки аутентификации с передачей пароля в открытом виде. Так как в данном случае пароль передается по сети в явном виде, использование метода не рекомендуется в недоверенных сетях. Дополнительная информация приведена в 9.3.2;

- значение gss требует использования для аутентификации пользователей GSSAPI. Значение доступно только для TCP/IP-соединений. Дополнительная информация приведена в 9.3.3;

- значение krb5 требует использования для аутентификации пользователей Kerberos V5. Доступно только для TCP/IP-соединений. Дополнительная информация приведена в 9.3.4.

- значение ident требует получения для клиента имени пользователя ОС путем обращения к ident серверу клиента и проверки его соответствия имени, предъявляемому клиентом. Доступно только для TCP/IP-соединений. Для локальных соединений вместо ident должен применятся метод peer. Дополнительная информация приведена в 9.3.5;

- значение peer требует получения для клиента имени пользователя ОС и проверки его соответствия имени, предъявляемому клиентом. Доступно только для локальных соединений. Дополнительная информация приведена в 9.3.6;

- значение ldap требует использования для аутентификации сервера LDAP. Дополнительная информация приведена в 9.3.7;

- значение radius требует использования для аутентификации сервера RADIUS. Дополнительная информация приведена в 9.3.8;

- значение cert требует использования для аутентификации клиентских сертификатов SSL. Дополнительная информация приведена в 9.3.9;

- значение pam требует использования для аутентификации сервиса подключаемых модулей аутентификации (Pluggable Authentication Modules – PAM), предоставляемого ОС. Дополнительная информация приведена в 9.3.10.

Поля auth-options. После поля auth-method может быть указана одна или несколько опций метода аутентификации в форме пар name=value (имя=значение).

Файлы, включенные посредством символа @ воспринимаются как списки имен, разделенных запятыми или пробелами. Комментарии вводятся #, аналогично файлу pg_hba.conf и разрешены вложенные конструкции с использованием символа @. Если имя файла, следующее за символом @, не является абсолютным путем, то оно определяется относительно директории, содержащей файл, в котором находится ссылка на данное имя файла.

Порядок записей файла pg_hba.conf имеет значение, вследствие последовательной проверки записей при каждой попытке соединения. В общем случае, первые записи должны содержать наиболее жесткие ограничения параметров соединения и наименее безопасные методы аутентификации, в последующих записях должны содержать менее жесткие ограничения для параметров соединения и более безопасные методы аутенти-

фикации. Например, можно использовать метод аутентификации trust для локальных TCP/IP-соединений, но требовать пароль для удаленных TCP/IP-соединений. В данном случае запись, определяющая метод аутентификации trust для соединений с 127.0.0.1, должна предшествовать записи, требующая аутентификации с паролем для широкого диапазона допустимых IP-адресов клиентов.

Файл pg_hba.conf считывается при запуске и при получении основным процессом сервера сигнала SIGHUP. После редактирования данного файла во время работы сервера, для повторного считывания файла необходимо отправить серверу сигнал, используя команду pg_ctl reload или kill -HUP.

Для подключения к определенной БД, пользователь должен не только пройти проверку в соответствии с записями pg_hba.conf, но и иметь привилегию CONNECT для данной БД. Таким образом, установку ограничений на подключение пользователей к БД проще осуществлять, добавляя или отбирая привилегию CONNECT, а не создавая правила в записях pg_hba.conf.

Подробная информация о различных методах аутентификации содержится в 9.3.

Пример

Примеры записей файла pg_hba.conf:

```
# Разрешить любому пользователю локальной системы подключаться к любой
# БД под любым именем пользователя БД, используя доменные сокеты Unix
# (по умолчанию для локальных соединений).
#
# TYPE  DATABASE     USER         ADDRESS              METHOD
local   all          all                               trust

# То же, используя локальные TCP/IP-соединения обратная петля.
#
# TYPE  DATABASE     USER         ADDRESS              METHOD
host    all          all          127.0.0.1/32         trust

# То же, но со столбцом IP-маски.
#
# TYPE  DATABASE     USER         IP-ADDRESS   IP-MASK          METHOD
host    all          all          127.0.0.1    255.255.255.255  trust

# То же для IPv6.
#
# TYPE  DATABASE     USER         ADDRESS              METHOD
```

```
host    all          all          ::1/128              trust

# То же, используя имя машины (обычно одинаково для IPv4 и IPv6).
#
# TYPE  DATABASE     USER         ADDRESS              METHOD
host    all          all          localhost            trust

# Разрешить любому пользователю с любого узла с IP-адресом 192.168.93.x
# подключаться к БД "postgres" от имени пользователя, которое выдаст для
# данного соединения сервис ident (как правило, имя пользователя Unix).
#
# TYPE  DATABASE     USER         ADDRESS              METHOD
host    postgres     all          192.168.93.0/24      ident

# Разрешить пользователю с узла сети 192.168.12.10 подключаться
# к БД "postgres", указав правильный пароль.
#
# TYPE  DATABASE     USER         ADDRESS              METHOD
host    postgres     all          192.168.12.10/32     md5

# Разрешить пользователю с узла домена example.com подключаться
# к БД "postgres", указав правильный пароль.
#
# TYPE  DATABASE     USER         ADDRESS              METHOD
host    all          all          .example.com         md5

# В отстутствие предыдущих строк типа "host", следующие две строки запрещают
# все соединения с 192.168.54.1 (поскольку запись будет проверена в первую
# очередь), но разрешают соединения используя Kerberos 5 с любого другого
# адреса. Нулевая маска означает, что ни один бит IP-адреса не
# рассматривается, следовательно подходит любой узел.
#
# TYPE  DATABASE     USER         ADDRESS              METHOD
host    all          all          192.168.54.1/32      reject
host    all          all          0.0.0.0/0            gss

# Разрешить пользователям с узлов 192.168.x.x подключаться к любой БД, если
# они проходят проверку ident. Если, например, сервис ident сообщает, что
```

```
# имя пользователя "bryanh", и он запрашивает соединение как пользователь
# PostgreSQL "guest1", то соединение разрешается, если в pg_ident.conf есть
# запись для карты "omicron", разрешающая "bryanh" соединяться как "guest1".
#
# TYPE  DATABASE     USER         ADDRESS              METHOD
host    all          all          192.168.0.0/16       ident map=omicron

# Если для локальных соединений записаны следующие три строки, они разрешают
# локальным пользователям подключаться только к их собственным БД (БД с тем
# же именем, что и имя пользователя БД) за исключением администраторов и
# членов роли "support", которые могут подключаться к любым БД. В файле
# $PGDATA/admins содержится список имен администраторов.
# Пароли требуются во всех случаях.
#
# TYPE  DATABASE     USER         ADDRESS              METHOD
local   sameuser     all                               md5
local   all          @admins                           md5
local   all          +support                          md5

# Две записи предыдущего примера можно объединить в одну:
local   all          @admins,+support                  md5

# Столбец DATABASE также может содержать списки и имена файлов:
# TYPE  DATABASE          USER   ADDRESS              METHOD
local   db1,db2,@demodbs  all                         md5
```

9.2. Карты имен пользователей

При использовании внешней системы аутентификации, подобно `Ident` или `GSSAPI`, имя пользователя ОС, который инициирует соединение, может не совпадать с именем пользователя БД, которое необходимо использовать для соединения. В подобном случае может быть применена карта имен пользователей ОС для отображения имен пользователей ОС в имена пользователей БД. Для использования карты имен пользователей необходимо указать `map=map-name` в поле опций файла `pg_hba.conf`. Опция поддерживается всеми методами аутентификации, которые получают внешние имена пользователей. Поскольку может быть необходимо использовать различные отображения для различных соединений, имя используемой карты задается в параметре `map-name` в файле `pg_hba.conf` для указания карты, применяемой для каждого конкретного соединения.

Карты отображений имен пользователей определены в файле карт, который по умол-

чанию называется pg_ident.conf и находится в каталоге данных кластера. Существует возможность поместить файл карт в любое другое место (см. описание конфигурационного параметра ident_file 8.2). Файл карт содержит записи следующего вида:

```
map-name system-username database-username
```

Комментарии и пробелы обрабатываются аналогично файлу pg_hba.conf. Поле map-name задает произвольное имя, которое будет использовано для ссылки на карту в файле pg_hba.conf. В двух других полях указывается имя пользователя ОС и соответствующее имя пользователя БД. Одно имя, заданное в поле map-name, может быть неоднократно использовано для указания множества отображений имен пользователей в одной карте.

Ограничения относительно количества пользователей БД соответствующих данному пользователю ОС и наоборот отсутствуют. Следовательно, запись в карте должна задаваться в значении, что данному пользователю ОС разрешено подключение как такому-то пользователю БД, а не в значении, что они эквивалентны. Подключение будет разрешено если существует некоторая запись в карте, определяющая, что имя пользователя, полученное от внешней системы аутентификации, соответствует имени пользователя БД, указанному в запросе на подключение.

Если поле system-username начинается с символа (/), то оставшаяся часть содержимого поля воспринимается как регулярное выражение. Дополнительная информация о синтаксисе регулярных выражений PostgreSQL приведена в 6.7.3.1. Регулярное выражение может включать одно заключенное в скобки подвыражение, на которые можно ссылаться в поле database-name, используя \1. Данная возможность позволяет выполнять отображение множества имен пользователей в одной строке, что особенно полезно для простых синтаксических подстановок. Далее приведен пример использования регулярных выражений в карте имен пользователей.

Пример

```
mymap   /^(.*)@mydomain.com       \1
mymap   /^(.*)@otherdomain.com    guest
```

Записи в данном примере будут исключать доменную часть имени для пользователей, у которых системное имя пользователя завершается @mydomain.com и разрешать всем пользователям, у которых системное имя завершается @otherdomain.com входить от имени учетной записи guest.

Следует помнить, что по умолчанию регулярное выражение может осуществлять сопоставление для части строки. Целесообразно использовать ^ и $, как показано в приведенном выше примере, для принудительного выполнения сопоставления для системного имени пользователя в целом.

Файл pg_ident.conf считывается при запуске и при получении сервером сигнала SIGHUP. По завершении редактирования данного файла во время работы системы необходимо отправить серверу сигнал посредством команды pg_ctl reload или kill -HUP для повторного считывания файла.

Далее приведен пример файла pg_ident.conf, который может быть использован совместно с файлом pg_hba.conf, приведенным в примере 9.1. В соответствии с записями файлов pg_ident.conf и pg_hba.conf, любой пользователь, выполнивший вход в систему в IP-сети 192.168 с именем, отличным от bryanh, ann или robert, доступ не получит. Системный пользователь robert сможет получить доступ при подключении с именем пользователя PostgreSQL bob, а не robert или какой-либо еще. Системный пользователь ann может подключиться только как ann. Системный пользователь bryanh сможет подключиться как bryanh или как guest1.

Пример

Файл pg_ident.conf

```
# MAPNAME     SYSTEM-USERNAME    PG-USERNAME

omicron       bryanh             bryanh
omicron       ann                ann
# на этих машинах bob соответствует имени пользователя robert
omicron       robert             bob
# bryanh также может подключаться как guest1
omicron       bryanh             guest1
```

9.3. Методы аутентификации

Рассмотрим различные методы аутентификации.

9.3.1. Доверенная аутентификация trust

При указании метода аутентификации trust PostgreSQL предполагает, что любому, кто может подключиться к серверу, разрешен доступ к БД от имени любого пользователя СУБД (в том числе и с именами суперпользователей). Конечно, при этом действуют ограничения, заданные в полях database и user. Данный метод следует использовать только при наличии адекватной защиты подключений к серверу на уровне ОС.

Метод аутентификации trust подходит и очень удобен для локальных соединений на однопользовательской рабочей станции. Как правило, метод не пригоден для многопользовательских машин. Однако, метод trust можно применять на многопользовательской системе, установив на сервере ограничения на доступ к файлу доменного сокета Unix, используя механизм прав ФС. Для настройки ограничений необходимо уста-

новить значение конфигурационного параметра unix_socket_permissions (и, возможно, unix_socket_group), как описано в 8.3 или установить значение конфигурационного параметра unix_socket_directory (в версии СУБД 9.6 unix_socket_directories), поместив файл сокета в защищенную директорию.

Настройка прав ФС помогает при подключении с использованием Unix-сокета. Локальные TCP/IP-соединения не ограничиваются правами ФС. Следовательно, при использовании прав ФС для обеспечения локальной безопасности необходимо исключить из файла pg_hba.conf строку host ... 127.0.0.1 ... или выбрать для нее другой метод аутентификации вместо trust.

Метод аутентификации trust подходит для TCP/IP-соединений только при наличии доверия каждому пользователю в каждой системе, которые могут использоваться для подключения к серверу, согласно записям pg_hba.conf, в которых указан метод trust. Рекомендуется не использовать метод trust для TCP/IP-соединений с адресами отличными от localhost (127.0.0.1).

9.3.2. Аутентификация по паролю

Методами аутентификации с паролем являются md5 и password. Данные методы работают одинаково, за исключением того, что в одном случае пароль отправляется по сети в виде хэш-значения, полученного по алгоритму MD5, а в другом случае в виде обычного текста.

Для противодействия перехвату паролей при передаче по сети предпочтительным является использование метода md5. Следует по возможности избегать применения метода password. Однако, метод md5 не может быть использован с параметром db_user_namespace (см. 8.3.2). Если соединение защищено с помощью SSL-шифрования, передача пароля может быть безопасной (хотя при использовании SSL лучшим решением будет использование аутентификации с помощью SSL-сертификатов).

Пароли к БД PostgreSQL отличаются от паролей пользователей ОС. Пароли для каждого пользователя БД хранятся в системном каталоге в pg_authid. Можно управлять паролями с помощью команд SQL CREATE USER и ALTER USER, например, CREATE USER foo WITH PASSWORD 'secret'. По умолчанию, если пароль для пользователя не задан, то хранится пустой пароль, и пользователь не сможет пройти аутентификацию.

9.3.3. Аутентификация с помощью GSSAPI

GSSAPI является промышленно стандартизированным протоколом для безопасной аутентификации, определенным в RFC 2743. PostgreSQL поддерживает GSSAPI с аутентификацией Kerberos, в соответствии с RFC 1964. GSSAPI для систем, которые его

поддерживают, предоставляет возможность автоматической аутентификации (однократное предъявление пароля). Аутентификация с использованием данного протокола является безопасной, но данные в рамках соединения с БД без использования SSL пересылаются в открытом виде.

При использовании GSSAPI с Kerberos, применяется стандартный принципал в формате `servicename/hostname@realm`. Сервер PostgreSQL будет принимать любой принципал, который указан в файле ключей (keytab) сервера, но необходимо удостовериться, чтобы указаны корректные параметры соединения, используя параметр `krbsrvname`. Настройка по умолчанию может быть изменена с `postgres` при указании скрипту `./configure --with-krb-srvnam=whatever`. В большинстве случаев данный параметр не требует изменения. Некоторые реализации Kerberos, такие как Microsoft Active Directory, требуют, чтобы имя сервиса было написано в верхнем регистре (`POSTGRES`).

`hostname` – полное имя хоста на сервервной машине. Область принципала сервиса (realm) является предпочтительной областью на серверной машине.

Клиентские принципалы должны иметь имя пользователя базы данных в качестве первого компонента принципала, например, `pgusername@realm`. В качестве альтернативы, вы можете импользовать соответствие пользователей между первым компонентом принципала и именем пользователя базы данных. По умолчанию область (realm) клиента не проверяется PostgreSQL. Если вы используете межобластную аутентификацию и необходимо проверять область, используйте параметр `krb_realm` или включите `include_realm` и используйте соответствие пользователей для проверки области.

Удостоверьтесь, что файл ключей сервера доступен только на чтение (и предпочтительно, только на чтение) аккаунтом сервера PostgreSQL(см. 7.1). Путь файла ключей определяется с помощью `krb_server_keyfile` конфигурационного файла (см. 8.3.2). Значением по умолчанию является `/usr/local/pgsql/etc/krb5.keytab` (или иная директория, указанная как `sysconfdir` при сборке). По причине безопасности рекомендуется использовать отдельный файл ключей для сервера PostgreSQL, даже если установлены права на чтение системного файла ключей.

Файл ключей генерируется с помощью програмного обеспечения Kerberos. Для подробностей обращайтесь к его документации. Следующий пример для совместимой с MIT реализацией Kerberos 5:

Пример

```
kadmin% ank -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

При подключении к базе данных убедитесь, что у вас есть билет для принципала,

совпадающего с именем пользователя базы данных. Например, пользователю базы данных fred принципалу fred@EXAMPLE.COM будет разрешено в подключении. Для того, чтобы разрешить принципал fred/users.example.com@EXAMPLE.COM, используйте карту имен пользователей (см. 9.2).

Для GSSAPI доступны следующие опции:

- include_realm определяет при значении параметра 1 необходимость включения имени области (realm) из имени принципала в имя пользователя ОС, которое формируется при сопоставлении имен с использованием карт имен (см. 9.2). Данная возможность полезна при работе с пользователями ОС, относящимися к разным областям (realm);

- map разрешает сопоставление между именами пользователей ОС и БД. Дополнительная информация приведена в 9.2. Для принципала Kerberos username/hostbased@EXAMPLE.COM именем пользователя для сопоставления является username/hostbased при выключенном include_realm, и username/hostbased@EXAMPLE.COM при включенном include_realm;

- krb_realm задает область, которой должно соответствовать имя принципала. При установке значения параметра подключение будет разрешено только для пользователей с указанной областью (realm). Если значение параметра не задано, то могут подключаться пользователи с любой областью (realm), для которых выполнено сопоставление имен.

9.3.4. Аутентификация Kerberos

ВНИМАНИЕ! Использование аутентификации Kerberos в чистом виде настоятельно не рекомендуется и допустимо только для поддержки обратной совместимости. Во вновь установленных и обновленных конфигурациях СУБД рекомендуется использовать промышленно стандартизованный протокол аутентификации GSSAPI (см. 9.3.3). В СУБД версии 9.6 метод аутентификации krb5 больше не поддерживается.

Kerberos является промышленно стандартизованной системой безопасной аутентификации, пригодной для организации распределенных вычислений в сетях общего доступа. Kerberos предоставляет механизм безопасной аутентификации, но в отличии от SSL не обеспечивает безопасной передачи запросов и данных по сети.

PostgreSQL поддерживает Kerberos версии 5. Поддержка Kerberos включается при сборке PostgreSQL.

PostgreSQL функционирует как обычный сервис Kerberos. Имя принципала сервиса задается в формате servicename/hostname@realm.

Значение servicename может быть задано на стороне сервера с использованием конфигурационного параметра krb_srvname, а на стороне клиента с использованием

параметра соединения krbsrvname. В большей части системных окружений значение параметра не требует изменения. Однако, при необходимости поддерживать несколько серверов PostgreSQL в одной системе потребуется внесение изменений. Некоторые реализации Kerberos также могут потребовать изменения имени сервиса, например Microsoft Active Directory требует указания имени сервиса в верхнем регистре (POSTGRES).

Значение hostname определяет полное имя узла сети, на котором функционирует сервер СУБД. Значение realm задает область принципала сервера, предпочтительно совпадающую с областью системы, в которой функционирует сервер.

Принципалы клиентов должны в качестве первого элемента принципала использовать имя пользователя БД PostgreSQL, например pgusername@realm. Может быть использована карта имен пользователей для сопоставления первому элементу имени принципала имени пользователя БД. По умолчанию PostgreSQL не проверяет область клиента. Если включена перекрестная аутентификация между областями и необходимо проверять область, следует использовать параметр krb_realm в pg_hba.conf или включить параметр include_realm для проверки области при сопоставлении имен пользователей.

Файл ключей keytab сервера должен быть доступен на чтение (предпочтительно только на чтение) для учетной записи сервера PostgreSQL. Дополнительная информация приведена в 7.1. Расположение файла ключей определяется значением конфигурационного параметра krb_server_keyfile. /usr/local/pgsql/etc/krb5.keytab (или директория, определенная при сборке в переменной sysconfdir).

Файл ключей keytab генерируется программными средствами Kerberos. Дополнительная информация приведена в документации на Kerberos. Приведенный далее пример применим для MIT-совместимых реализаций Kerberos 5.

Пример

```
kadmin% ank -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

При подключении к БД необходимо наличие сеансового билета для принципала, соответствующего запрашиваемому имени пользователя БД. Например, для имени пользователя БД fred при аутентификации на сервере БД может быть использован принципал fred@EXAMPLE.COM или fred/users.example.com@EXAMPLE.COM.

При использовании на WEB-сервере Apache модулей mod_auth_kerb и mod_perl для параметра тип аутентификации AuthType может быть установлено значение KerberosV5SaveCredentials с выполнением через модуль mod_perl скриптов. Данная возможность позволяет устанавливать безопасное соединение с БД через WEB-сервер, не требуя дополнительного ввода пароля.

Для Kerberos поддерживаются следующие опции:

- `map` разрешает сопоставление между именами пользователей ОС и БД. Дополнительная информация приведена в 9.2.

- `include_realm` определяет при значении параметра 1 необходимость включения имени области (realm) из имени принципала в имя пользователя ОС, которое формируется при сопоставлении имен с использованием карт имен (см. 9.2). Данная возможность полезна при работе с пользователями ОС, относящимися к разным областям (realm);

- `krb_realm` задает область, которой должно соответствовать имя принципала. При установке значения параметра подключение будет разрешено только для пользователей с указанной областью (realm). Если значение параметра не задано, то могут подключаться пользователи с любой областью (realm), для которых выполнено сопоставление имен;

- `krb_server_hostname` задает часть имени принципала сервиса, определяющую имя узла сети. Которая в комбинации с `krb_srvname` используется для получения полного принципала сервиса `krb_srvname/krb_server_hostname@REALM`. Если значение параметра не задано, используется имя узла сети сервера.

9.3.5. Аутентификация Ident

Метод аутентификации `ident` заключается в получении клиентского имени пользователя ОС от ident-сервера и последующем использовании его как разрешенного имени пользователя БД или сопоставлении ему имени пользователя БД посредством карты имен пользователей. Метод доступен только для TCP/IP-соединений.

Примечание. При указании `ident` для не TCP/IP соединений вместо него используется метод `peer` (см. 9.3.6).

Для метода `ident` поддерживается опция `map`, которая разрешает сопоставление имен пользователей ОС и БД. Дополнительная информация приведена в 9.2.

Метод использует протокол Identification Protocol, описанный в RFC 1413. Практически любая ОС семейства Unix включает сервер идентификации, который по умолчанию ожидает соединения по протоколу TCP, слушая порт 113. Основной задачей сервера идентификации является выдача ответов на запросы типа «Какой пользователь инициировал соединение с вашего порта `X` и подключается к моему порту `Y`?». Поскольку PostgreSQL известны оба значения портов X и Y, когда физическое соединение установлено, серверу идентификации на узле сети, с которого устанавливается клиентское соединение, может быть выдан запрос и, следовательно, определено имя пользователя ОС для данного соединения.

Недостаток этой процедуры заключается в зависимости от целостности системы кли-

ента. В случае когда клиентская система не является доверенной или скомпрометирована, нарушитель может запустить любую программу, прослушивающую порт 113, и вернуть имя пользователя по своему усмотрению. Следовательно, данный метод аутентификации может быть использован только в закрытых сетях, в которых каждая клиентская система находится под контролем, а системный администратор и администратор СУБД работают в тесном взаимодействии. Таким образом, необходимо обеспечить доверенность системы, в которой функционирует сервер идентификации. Следует отметить, что в RFC 1413 содержится следующее предупреждение:

«The Identification Protocol is not intended as an authorization or access control protocol»

Некоторые серверы идентификации имеют нестандартную опцию для преобразования имени пользователя с помощью ключа, известного только администратору системы, в которой работает сервер идентификации. При работе с PostgreSQL данную опцию невозможно использовать, т.к. PostgreSQL не имеет механизма обратного преобразования возвращенной строки для получения действительного имени пользователя.

9.3.6. Аутентификация Peer

Метод аутентификации `peer` заключается в получении клиентского имени пользователя ОС от ядра ОС и последующем использовании его как разрешенного имени пользователя БД или сопоставлении ему имени пользователя БД посредством карты имен пользователей. Метод доступен только для локальных соединений.

Для метода `peer` поддерживается опция `map`, которая разрешает сопоставление имен пользователей ОС и БД. Дополнительная информация приведена в 9.2.

Метод аутентификации `peer` доступен только в ОС, предоставляющих функцию `getpeereid()`, параметр сокета `SO_PEERCRED` или подобный механизм. В настоящее время это Linux, FreeBSD, NetBSD, OpenBSD, BSD/OS и Solaris.

9.3.7. Аутентификация LDAP

Данный метод аутентификации работает подобно методу `password`, за исключением того, что LDAP используется как метод проверки пароля. LDAP используется только для подтверждения соответствия имени пользователя и пароля. Следовательно, для аутентификации пользователя с применением LDAP необходимо наличие его учетной записи в БД.

Метод аутентификации LDAP может действовать в двух режимах. В первом, называемым простым связыванием, сервер выполняет попытку обращения к LDAP-серверу с использованием уникального имени (distinguished name DN), состоящего из следующих частей: `prefix`, `username` и `suffix`. Обычно, значение `prefix` используется для указания `cn=` или `DOMAIN\` в окружении Active Directory, а суффикс используется для задания

оставшейся части уникального имени в окружении не Active Directory.

Во втором режиме, называемом связыванием с поиском, сервер сначала связывается с каталогом LDAP от имени выделенного пользователя с именем и паролем, заданными опциями `ldapbinddn` и `ldapbindpasswd`, и производит поиск пользователя, выполняющего попытку установки соединения с СУБД. Если не заданы имя и пароль такого пользователя, обращение к каталогу LDAP выполняется анонимно. Поиск выполняется в поддереве `ldapbasedn` по атрибуту `ldapsearchattribute`. В случае успешного обнаружения пользователя, сервер отключается и пытается обратиться к каталогу LDAP от имени этого пользователя с помощью пароля, предоставленного клиентом, для проверки его правильности. Такой режим соответствует подходу, используемому при аутентификации с помощью LDAP в других программных продуктах, таких как модуль Apache `mod_authnz_ldap` или `pam_ldap`. Описанный метод более гибок в плане расположения объектов, описывающих пользователей, в каталоге LDAP, но требует от сервера установки двух отдельных соединений с LDAP-сервером.

Для LDAP поддерживаются следующие опции:

- `ldapserver` определяет имя или IP-адрес сервера LDAP, к которому осуществляется обращение. Через пробел может быть указано несколько серверов;
- `ldapport` определяет номер порта сервера LDAP, к которому осуществляется обращение. Если номер порта не определен, используется значение порта по умолчанию из библиотеки LDAP;
- `ldaptls` определяет, при значении 1, необходимость защиты соединения между PostgreSQL и сервером LDAP с использованием TLS. Следует отметить, что таким образом защищается только сетевой трафик между PostgreSQL и сервером LDAP, соединение с клиентом без применения SSL остается не защищенным.

Следующие опции используются только в простом режиме:

- `ldapprefix` определяет `prefix`, предшествующий имени пользователя при создании уникального имени;
- `ldapsuffix` определяет `suffix`, следующий за именем пользователя при создании уникального имени.

Следующие опции используются только в режиме связывания с поиском:

- `ldapbasedn` определяет DN элемента, от которого начинается поиск пользователя;
- `ldapbinddn` определяет DN пользователя, от имени которого производится поиск.
- `ldapbindpasswd` определяет пароль пользователя, от имени которого производится поиск;
- `ldapsearchattribute` определяет имя атрибута, со значением которого сравнивается имя пользователя при поиске;

- `ldapurl` определяет URL-строку соединения с сервером LDAP в соответствии с RFC 4516. Является альтернативным способом задания некоторых других LDAP-опций в более компактной и стандартной форме. Используется следующий формат строки:

  `ldap://host[:port]/basedn[?[attribute][?[scope]]]`

  `scope` должно быть одним из значений `base`, `one` или `sub`. Обычно используется последнее. Может быть использован только один атрибут. Другие опции LDAP, такие как фильтры и расширения, не поддерживаются.

  Для не анонимных соединений опции `ldapbinddn` и `ldapbindpasswd` всегда должны быть заданы.

  Для использования безопасных LDAP-соединений в дополнение к `ldapurl` должна использоваться опция `ldaptls`. Схема работы с `ldaps` (непосредственное SSL-соединение) не поддерживается.

Смешанное использование опций простого режима и режима связывания с поиском является ошибкой.

Пример конфигурации LDAP для простого режима:

```
host ... ldap ldapserver=ldap.example.net ldapprefix="cn=" \
  ldapsuffix=", dc=example, dc=net"
```

При установке соединения с СУБД от имени пользователя СУБД `someuser` PostgreSQL выполняет попытку обращения к LDAP-серверу с использованием DN `cn=someuser, dc=example, dc=net` и пароля, предоставленного клиентом. Если соединение успешно устанавливается, то доступ к базе данных разрешается.

Пример конфигурации LDAP для режима связывания с поиском:

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example, dc=net" \
  ldapsearchattribute=uid
```

При установке соединения с СУБД от имени пользователя СУБД `someuser`, PostgreSQL выполняет попытку обращения к LDAP-серверу анонимно (опция `ldapbinddn` не указана), выполняет поиск записи с (`uid=someuser`) в указанном базовом DN. Если такая запись найдена, выполняется попытка связи с LDAP-сервером от имени найденного пользователя и пароля, предоставленного клиентом. Если установка второго соединения проходит успешно, доступ к базе данных разрешается.

Пример той же конфигурации LDAP для режима связывания с поиском, записанной в виде URL:

```
host ... ldap lapurl="ldap://ldap.example.net/dc=example,dc=net?uid?sub"
```

Поскольку LDAP использует запятые и пробелы для отделения различных частей уникального имени, необходимо окружать двойными кавычками значения параметров при конкурировании LDAP.

9.3.8. Аутентификация RADIUS

Данный метод аутентификации работает подобно методу `password`, за исключением того, что RADIUS используется как метод проверки пароля. RADIUS используется только для подтверждения соответствия имени пользователя и пароля. Следовательно, для аутентификации пользователя с применением RADIUS необходимо наличие его учетной записи в БД.

При использовании аутентификации RADIUS указанному серверу RADIUS посылается сообщение запроса доступа (Access Request message). Запрос имеет тип «только аутентификация» (Authenticate Only) и содержит параметры для имени пользователя, пароля (зашифрованного) и идентификатор NAS. Запрос шифруется с использованием разделяемого с сервером секрета. Сервер RADIUS отвечает либо об успешной аутентификации (Access Accept), либо об отказе (Access Reject). Учетные возможности RADIUS не поддерживаются.

Для RADIUS поддерживаются следующие опции:

- `radiusserver` определяет имя или IP-адрес сервера RADIUS, к которому осуществляется обращение. Параметр является обязательным;
- `radiussecret` определяет общую секретную фразу для защищенного взаимодействии с RADIUS-сервером. Фраза должна иметь одинаковое значение на PostgreSQL- и RADIUS-серверах. Рекомендуется использовать не менее 16-ти символов. Параметр является обязательным.

  Примечание. Вектор шифрования будет считаться криптографически надежным, только если PostgreSQL собран с поддержкой OpenSSL. В противном случае передача данных серверу RADIUS считается только скрытой, но не безопасной, и при необходимости должны использоваться другие меры обеспечения безопасности.
- `radiusport` определяет номер порта сервера RADIUS, к которому осуществляется обращение. Если номер порта не определен, используется значение порта по умолчанию, равное 1812;
- `radiusidentifier` определяет строку, используемую в качестве NAS идентификатора в RADIUS-запросах. Параметр может быть использован как второй параметр, идентифицирующий к примеру пользователя базы данных, выполняющего попытку аутентификации, на основании чего может быть выбрана политика RADIUS-сервера. Без указания идентификатора по умолчанию используется `postgresql`.

9.3.9. Аутентификация с использованием сертификатов

В данном методе аутентификации применяются клиентские сертификаты SSL. Следовательно, он доступен только для SSL-соединений. При использовании метода сервер будет требовать от клиента предоставления действительного сертификата. Запрос на ввод пароля клиенту отправляться не будет. Атрибут сертификата `cn` сравнивается с указанным

в запросе именем пользователя БД и при их совпадении пользователю разрешается вход. Карты имен пользователей могут быть использованы для разрешения входа с значением cn, отличным от имени пользователя БД. Дополнительная информация приведена в 9.2.

9.3.10. Аутентификация PAM

Данный метод аутентификации работает подобно методу password, за исключением того, что PAM (Pluggable Authentication Modules — подключаемый модуль аутентификации) используется как механизм аутентификации. Имя сервиса PAM по умолчанию postgresql. PAM используется только для подтверждения соответствия имени пользователя и пароля. Следовательно, для аутентификации пользователя с применением PAM необходимо наличие его учетной записи в БД.

Для PAM поддерживается опция pamservice, которая определяет имя сервиса PAM.

Примечание. При настройке PAM для чтения /etc/shadow аутентификация не пройдет, поскольку сервер PostgreSQL запускается не от имени суперпользователя root. При настройке PAM для использования LDAP или других методов аутентификации такой проблемы нет.

9.4. Ошибки при аутентификации

Ошибки при аутентификации или проблемы, связанные с ними, обычно выражаются в появлении сообщений об ошибках, подобных представленным ниже.

Получение следующего сообщения наиболее вероятно в случае, когда соединение с сервером установлено успешно, но сервер отказывается продолжать диалог. Данное сообщение предполагает, что сервер отклонил запрос на подключение, поскольку не нашел подходящей записи в конфигурационном файле pg_hba.conf.

```
FATAL:  no pg_hba.conf entry for host "123.123.123.123",
user "andym", database "testdb"
```

Сообщения, подобные следующему, означают, что соединение с сервером установлено, сервер готов продолжать диалог, но после прохождения процедуры аутентификации по методу, определенному в файле pg_hba.conf. Необходимо в зависимости от содержимого сообщения, проверить правильность вводимого пароля, программное обеспечение Kerberos или ident.

```
FATAL:  Password authentication failed for user "andym"
```

Сообщения, подобные следующему, означают, что не удалось найти указанное имя пользователя.

```
FATAL:  user "andym" does not exist
```

Сообщения, подобные следующему, означают, что БД, к которой выполняется подключение, не существует. Следует отметить, что если имя БД не указано, то используется

значение по умолчанию.

```
FATAL:  database "testdb" does not exist
```

Журнал сервера может содержать больше информации об ошибках при аутентификации, чем выдаваемые клиенту сообщения. При возникновении сомнений в причине ошибки при аутентификации необходимо проверить журнал сервера.

10. РОЛИ И ПРИВИЛЕГИИ В СУБД

В СУБД PostgreSQL для управления правами на доступ к БД используется концепция ролей. Под ролью понимается пользователь или группа пользователей БД. Роли могут являться владельцами объектов БД (например, таблиц) и могут назначать привилегии на управление объектами для других ролей, имеющих доступ к данными объектам.

Концепция ролей объединяет концепции «пользователи» и «группы». В версиях PostgreSQL до 8.1. пользователи и группы являлись отдельными субъектами доступа, но в настоящее время существуют только роли. Любая роль может действовать как пользователь, как группа или, как и то и другое.

Далее описывается создание ролей, управление ролями и система привилегий.

ВНИМАНИЕ! СУБД PostgreSQL была доработана для обеспечения соответствия требованиям по защите информации от несанкционированного доступа. При реализации данных требований была установлена необходимость обеспечения соответствия пользователей СУБД учетным записям в ОС. Также введены некоторые ограничения на использование ролей в СУБД (см **??**).

10.1. Роли СУБД

Роли СУБД концептуально совершенно отличаются от пользователей ОС. На практике обеспечение соответствия между ними может быть удобным, но не является необходимым. Роли СУБД являются глобальными для всего кластера БД (а не для отдельной БД). Для создания роли используется команда SQL `CREATE ROLE`:

```
CREATE ROLE name;
```

Значение `name` должно удовлетворять правилам для идентификаторов SQL: или не использовать специальные символы или заключаться в двойные кавычки. На практике команда обычно используется с дополнительными опциями, такими как `LOGIN`. Для удаления созданной роли используется аналогичная команда SQL `DROP ROLE`:

```
DROP ROLE name;
```

Для удобства реализованы программы `createuser` и `dropuser`, которые вызываются из оболочки командной строки:

```
createuser name
dropuser name
```

Для получения перечня существующих ролей необходимо опросить системный каталог `pg_roles`:

Пример

```
SELECT rolname FROM pg_roles;
```

Метакоманда `\du` программы `psql` также используется для перечисления существу-

ющих ролей.

Для начальной загрузки СУБД во вновь инициализированной системе всегда существует одна предопределенная роль. Данная роль всегда является «суперпользователем», и по умолчанию (если не изменено при выполнении `initdb`) имеет то же имя, что и пользователь ОС, инициализировавший кластер БД. По традиции роль называется `postgres`. Для создания новых ролей необходимо подключиться от имени данной начальной роли.

Каждое подключение к серверу СУБД осуществляется от имени определенной роли, которая определяет первичные права доступа для команд, используемых в контексте подключения. Имя роли, используемое при подключении к БД, определяется клиентом в запросе на соединение способом, зависящим от приложения. Например, программа `psql` для задания роли использует опцию командной строки `-U`. Многие приложения по умолчанию берут имя текущего пользователя ОС (включая `createuser` и `psql`). Следовательно, зачастую удобно обеспечить соответствие между именами ролей и именами пользователей ОС.

Набор ролей СУБД, от имени которых клиент может осуществить подключение, определяется значениями параметров аутентификации клиента, как описано в 9. Таким образом, наличие ограничения, определяющего, что имя роли клиента для подключения должно совпадать с именем пользователя ОС, не является необходимым, так же как и имя пользователя в системе не обязательно должно совпадать с реальным именем пользователя. Поскольку роль определяет набор прав доступа для подключенного клиента, необходимо внимательно выполнять соответствующую настройку в многопользовательском окружении.

10.2. Атрибуты ролей СУБД

Роль БД может иметь следующий набор атрибутов, определяющих ее права и взаимодействующих с системой аутентификации клиента:

1) *Привилегия установки соединения* определяет, что роль имеет право `LOGIN` (вход в СУБД) и имя роли может использоваться для подключения к БД. Роль с атрибутом `LOGIN` может рассматриваться как пользователь БД. Для создания роли с правом входа в СУБД используются следующие команды:

```
CREATE ROLE name LOGIN;
CREATE USER name;
```

Команда `CREATE USER` является эквивалентом команды `CREATE ROLE`, за исключением того, что `CREATE USER` использует атрибут `LOGIN` по умолчанию, а `CREATE ROLE` нет.

2) *Статус суперпользователя* определяет, что роль является суперпользователем. Суперпользователь СУБД обходит все проверки прав на доступ. Целесооб-

разно основную работу проводить от имени роли, не являющейся суперпользователем. Для создания нового суперпользователя СУБД используется команда `CREATE ROLE name SUPERUSER` от имени роли, которая уже является суперпользователем.

3) *Создание баз данных* явно определяет разрешение на создание БД (исключая суперпользователей, поскольку они обходят все проверки прав на доступ). Для создания подобной роли используется команда `CREATE ROLE name CREATEDB`.

4) *Создание ролей* явно определяет разрешение на создание других ролей (кроме суперпользователей, т.к. они обходят все проверки прав на доступ). Для создания подобной роли используется команда `CREATE ROLE name CREATEROLE`. Роль с правом на создание ролей `CREATEROLE` может также изменять и удалять другие роли, предоставлять и отменять членство в ролях. Однако, для создания, изменения, удаления или изменения членства ролей с атрибутом «суперпользователь» необходим статус суперпользователя, права `CREATEROLE` недостаточно.

5) *Инициация репликации* явно определяет разрешение на инициацию потоковой репликации (исключая суперпользователей, поскольку они обходят все проверки прав на доступ). Используемая для репликации роль должна иметь право `LOGIN`. Для создания подобной роли используется команда `CREATE ROLE name REPLICATION LOGIN`.

6) *Наличие пароля* определяет наличие пароля для роли. Атрибут значим, только если метод аутентификации клиента требует, чтобы пользователь вводил пароль при установке подключении к БД. Пароли используются в методах аутентификации `password` и `md5`. Пароли СУБД отличаются от паролей ОС. Пароль можно задать при создании роли с помощью команды `CREATE ROLE name PASSWORD 'string'`.

Атрибуты роли после создания можно менять командой `ALTER ROLE`. Подробная информация находится в справочнике по командам `CREATE ROLE` и `ALTER ROLE`.

На практике целесообразно создать роль с привилегиями `CREATEDB` и `CREATEROLE`, но не являющуюся суперпользователем, а затем использовать данную роль для выполнения операций по управлению СУБД и ролями. Подобный подход позволяет избежать рисков, возникающих при решении от имени суперпользователя задач, которые в действительности не требуют данной привилегии.

Для роли можно определить свои значения по умолчанию многих рабочих конфигурационных параметров, рассмотренных в 8. Например, если по какой-либо причине необходимо запретить индексное сканирование (что на практике не является удачным решением), в любом подключении можно использовать команду:

```
ALTER ROLE myname SET enable_indexscan TO off;
```

Команда сохранит новое значение параметра (но не установит его немедленно). В последующих подключениях для этой роли будет считаться, что `SET enable_indexscan TO off` было выполнено непосредственно перед запуском сессии. Существует возможность изменить значение параметра во время сессии, т.к. данным способом задается только значение по умолчанию. Для изменения значения по умолчанию, установленное для данной роли, используется команда `ALTER ROLE rolename RESET varname`. Следует отметить, что настраивать специальные значения по умолчанию для ролей без привилегии `LOGIN` бесполезно, поскольку они никогда не будут использоваться.

10.3. Членство в ролях СУБД

Зачастую удобно объединять пользователей в группы для облегчения управления правами. При таком подходе права назначаются и отнимаются для группы в целом. В СУБД PostgreSQL подход реализуется посредством создания роли, представляющей группу с последующим назначением членства в группе отдельным пользовательским ролям.

Для настройки групповой роли необходимо создать ее командной:

```
CREATE ROLE name;
```

В общем случае, роль, используемая в качестве группы, не имеет атрибута `LOGIN`, но при необходимости атрибут может быть добавлен.

После создания групповой роли можно добавлять и удалять ее членов с командами `GRANT` и `REVOKE`, соответственно:

```
GRANT group_role TO role1, ... ;
REVOKE group_role FROM role1, ... ;
```

Можно предоставить членство в групповой роли другим групповым ролям, поскольку не существует реальных отличий между групповыми и не групповыми ролями. СУБД не позволит установить циклическое членство в группах. Кроме того, запрещено предоставлять членство в групповой роли для специального имени `PUBLIC`.

Члены роли могут использовать права группы двумя способами. Во-первых, каждый член группы может явно выполнить команду `SET ROLE` и временно «стать» групповой ролью. В таком состоянии сессия БД имеет доступ к правам групповой роли, а не первоначальной роли, от имени которой был произведен вход, и любые созданные объекты БД рассматриваются как принадлежащие групповой роли, а не первоначальной. Во-вторых, члены группы, имеющие атрибут `INHERIT`, автоматически используют права ролей, членами которых они являются. Например, предположим, что выполнены команды:

```
CREATE ROLE joe LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel NOINHERIT;
```

```
GRANT admin TO joe;
GRANT wheel TO admin;
```

Непосредственно после подключения от имени роли joe, сессия БД будет использовать права, предоставленные непосредственно joe плюс все права, предоставленные роли admin, поскольку joe «наследует» все права admin. Однако, права, предоставленные wheel, недоступны, поскольку joe хотя опосредованно и является членом wheel, но членство предоставлено через роль admin, которая имеет атрибут NOINHERIT. После выполнения команды:

```
SET ROLE admin;
```

сессия будет пользоваться только правами, предоставленными admin, а не joe. После выполнения команды:

```
SET ROLE wheel;
```

сессия будет пользоваться только правами, предоставленными wheel, а не admin или joe. Первоначальное установка прав может быть возвращена с помощью одной из команд:

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

Примечание. Команда SET ROLE всегда разрешает выбор любой роли, членом которой, прямо или опосредованно, является роль, от имени которой был произведен вход. Таким образом, в приведенном примере нет необходимости становиться ролью admin для того, чтобы стать ролью wheel.

Примечание. В стандарте SQL есть четкое различие между пользователями и ролями, и пользователи автоматически не наследуют права в отличие от ролей. Такое поведение в СУБД PostgreSQL может быть обеспечено установкой для ролей, используемым как роли SQL, атрибута INHERIT, а ролям, используемым как пользователи SQL, атрибут NOINHERIT. Однако, по умолчанию в PostgreSQL для всех ролей устанавливается атрибут INHERIT для обеспечения обратной совместимости с версиями до 8.1, в которых пользователи всегда пользовались правами, предоставленными группам, членам которых они являются.

Атрибуты ролей LOGIN, SUPERUSER, CREATEDB и CREATEROLE могут считаться особыми привилегиями, которые никогда не наследуются, как обычные права на доступ к объектам БД. Для получения возможности использовать названные особые привилегии необходимо действительно выполнить команду SET ROLE став ролью, которая имеет соответствующий атрибут. Далее предположим, что привилегии CREATEDB и CREATEROLE были предоставлены роли admin. Тогда при подключении от имени роли joe указанные особые привилегии не будут предоставлены немедленно, но могут быть получены после

выполнения `SET ROLE admin`.

Для удаления групповой роли используется команда `DROP ROLE`:

```
DROP ROLE name;
```

При этом, любое членство в данной групповой роли автоматически отменяется, но другого влияния на роли, являющееся членами данной групповой роли не оказывается. Следует отметить однако, что все объекты, владельцем которых является данная групповая роль, сначала должны быть удалены или связанны с другим владельцам, и любые права, предоставленные групповой роли, должны быть отняты.

10.4. Функции и триггеры

Функции и триггеры (функции, которые автоматически выполняются при определенных условиях) позволяют пользователям вставлять в серверную часть код, который другие пользователи могут выполнить непреднамеренно. Таким образом, оба механизма разрешают пользователям относительно легко запустить троянские программы других пользователей. Единственной реальной защитой является жесткий контроль за возможностью определения функций.

Функции выполняются в серверном процессе с правами ОС для демона сервера СУБД. Если используемый для создания функции язык программирования позволяет беспрепятственный доступ к памяти, то существует возможность изменить внутреннюю структуры данных сервера. Таким образом, кроме всего прочего, функции могут обойти системный контроль доступа. Функции, написанные на языках, позволяющих осуществлять подобный доступ к памяти, считаются недоверенными и в СУБД PostgreSQL только суперпользователям разрешено создавать функции, написанные на этих языках.

11. УПРАВЛЕНИЕ БД

Каждый работающий сервер PostgreSQL управляет одной или несколькими БД. Таким образом, БД находятся на верхнем уровне иерархии для организации объектов SQL (объектов БД). Далее описаны свойства БД, способы их создания, удаления и управления ими.

11.1. Обзор управления БД

Базой данных является именованный набор объектов SQL (объектов БД). Как правило, каждый объект БД (таблицы, функции и прочие) принадлежит одной и только одной БД. Но существует несколько системных каталогов, например, `pg_database`, которые принадлежат всему кластеру и доступны из каждой БД в пределах кластера. Более точно БД является совокупностью схем, содержащих таблицы, функции и прочие объекты. Таким образом, полная иерархия следующая: сервер, БД, схема, таблица (или объект другого типа, например, функция).

При подключении к серверу СУБД клиент должен указать в запросе на соединение имя БД, к которой осуществляется подключение. Подключение в одной сессии более чем к одной БД невозможно. Но приложение не ограничено в числе подключений, которые осуществляются к той же или к другим БД. Базы данных физически разделены, и контроль доступа к ним осуществляется на уровне подключения. Следовательно, если один сервер PostgreSQL работает с проектами или пользователями, которые должны быть разделены, то рекомендуется поместить их в отдельных БД. Если проекты или пользователи взаимосвязаны и должны иметь возможность использовать ресурсы друг друга, то их следует поместить в одну БД, но, возможно, в отдельных схемах. Схема является чисто логической структурой, доступ к которой определяется правами, настроенными в системе. Дополнительная информация об управлении схемами приведена в 2.7.

БД создаются командой `CREATE DATABASE` (см. 11.2), а удаляются командой `DROP DATABASE` (см. 11.5). Для получения перечня существующих БД необходимо посмотреть системный каталог `pg_database`, например:

```
SELECT datname FROM pg_database;
```

Кроме того, можно использовать метакоманду `\l` программы `psql` или опцию командной строки `-l`.

Примечание. В стандарте SQL базы данных называются «каталогами», но практической разницы нет.

11.2. Создание БД

Для создания БД необходимо наличие запущенного и работающего сервера PostgreSQL (см. 7.3).

БД создаются следующей командой SQL:

```
CREATE DATABASE name;
```

где name должно удовлетворять обычным правилам для идентификаторов SQL. Текущая роль автоматически становится владельцем новой БД. Именно владелец БД имеет право удалить ее впоследствии (что приведет к удалению всех объекты внутри БД, даже если они имеют других владельцев).

Создание БД является операцией, ограничиваемой правами доступа. Дополнительная информация о предоставлении необходимых прав приведена в 10.2.

Поскольку для выполнения команды CREATE DATABASE необходимо наличие подключения к серверу СУБД, остается вопрос о создании первой БД. Первая БД всегда создается командой initdb при инициализации области хранения данных — кластера БД (см. 7.2). Данная БД называется postgres. Таким образом, для создания первой «обычной» БД можно подключиться к БД postgres.

Вторая БД template1 также создается командой initdb. При создании в кластере новой БД, как правило, клонируется template1. Это означает, что все изменения, внесенные в template1 распространятся на все создаваемые впоследствии БД. Следовательно, рекомендуется не использовать template1 для реальной работы, но при разумном использовании данная возможность может быть удобной. Дополнительная информация приведена в 11.3.

Для удобства существует программа createdb, которая может быть выполнена из оболочки командной строки для создания новых БД:

```
createdb dbname
```

Данная программа не делает ничего необычного. Она подключается к БД postgres и выполняет команду CREATE DATABASE. Детальная информация о вызове createdb содержится в справочной странице. Запуск createdb без аргументов создаст БД с именем текущего пользователя.

Примечание. В 9 приведена информация об ограничении права доступа к какой-либо БД.

Иногда необходимо создать БД для другого пользователя. Его роль должна стать владельцем новой БД, чтобы иметь возможность конфигурировать и управлять ею самостоятельно. Этого можно достичь с помощью одной из перечисленных ниже команд:

```
CREATE DATABASE dbname OWNER rolename;
```

из SQL-окружения, или:

```
createdb -O rolename dbname
```

из командной строки.

Для создания БД для другого пользователя (т.е. для роли, членом которой вы не являетесь) необходимо быть суперпользователем.

11.3. Шаблоны БД

Команда `CREATE DATABASE` фактически копирует существующую БД. По умолчанию она копирует стандартную системную БД `template1`. Таким образом, эта БД является «шаблоном», из которого создаются все остальные БД. При добавлении объектов в `template1` данные объекты будут скопированы во все БД, созданные впоследствии. Подобное поведение позволяет обеспечить локализацию изменений стандартного набора объектов БД. Например, при добавлении в `template1` процедурного языка PL/Perl, он автоматически будет добавляться ко всем создаваемым БД пользователей.

Существует вторая стандартная системная БД `template0`. Данная БД содержит исходные объекты `template1`, таким образом, в ней содержатся только стандартные объекты, предопределенные в данной версии PostgreSQL. После выполнения `initdb` БД `template0` не изменяется. Следовательно, указав в команде `CREATE DATABASE` копировать `template0` вместо `template1`, можно создать «чистую» пользовательскую БД, не содержащую изменения, сделанные в `template1`. Наличие подобной возможности особенно удобно при восстановлении `pg_dump` дампа: скрипт дампа должен восстанавливаться в чистой БД для обеспечения гарантии воссоздания правильного содержимого заархивированной БД, не содержащего конфликтов между объектами, которые могли быть добавлены в `template1` позднее.

Другой причиной для копирования `template0` вместо `template1` является возможность задания настроек локализации при ее копировании, тогда как копия `template1` должна иметь ее настройки. Это связано с тем, что `template1` может содержать данные, специфичные для выбранной кодировки или настроек локализации, а `template0` — нет.

Для создания БД копированием `template0` используются команды:

```
CREATE DATABASE dbname TEMPLATE template0;
```

из SQL-окружения, или:

```
createdb -T template0 dbname
```

из командной строки.

Существует возможность создавать дополнительные шаблонные БД, и при выполнении команды `CREATE DATABASE` в качестве шаблона использовать любую БД кластера. Однако, важно понимать, что подобные действия не являются полноценной реализацией действия по копированию базы данных. Принципиальное ограничение заключается в невозможности подключения ни одной другой сессии к исходной БД во время копирования.

Команда CREATE DATABASE не будет выполнена, если при ее запуске есть хоть одно подключение к исходной БД, если же подключений нет, все новые подключения блокируются до завершения выполнения CREATE DATABASE.

В pg_database для каждой БД присутствуют два полезных флага: столбцы datistemplate и datallowconn. Столбец datistemplate можно сделать индикатором, что БД предназначена для использования в качестве шаблона в CREATE DATABASE. Если флаг установлен, БД сможет клонировать любой пользователь, имеющий привилегию CREATEDB. В противном случае, клонировать БД сможет только суперпользователь или ее владелец. Если значение datallowconn false, то к БД запрещено устанавливать новые подключения (но существующие сессии не закрываются простой установкой значения флага false). Для БД template0 обычно установлено datallowconn = false, что предотвращает ее изменение. БД template0 и template1 должны всегда быть отмечены флагом datistemplate = true.

Примечание. Следует отметить, что ни template1, ни template0 не имеют какого-то особого статуса, за исключением того, что имя по умолчанию template1 является исходной БД для CREATE DATABASE. Например, можно беспрепятственно удалить template1 и воссоздать ее из template0. Подобная операция рекомендуется в случае, когда template1 по неосторожности была «замусорена». Для удаления template1 необходимо установить флаг pg_database.datistemplate = false.

Примечание. При инициализации кластера создается БД postgres. По умолчанию пользователи и приложения подключаются к данной БД, которая является просто копией template1 и может быть удалена и вновь создана при необходимости.

11.4. Конфигурирование БД

Сервер PostgreSQL предоставляет значительное количество конфигурационных параметров (см. 8). Существует возможность установить специфические для БД значения для многих из данных параметров.

Например, при необходимости отключить оптимизатор GEQO для определенной БД, обычно его отключают для всех БД или при каждом подключении клиента проверяют, что выполнена команда SET geqo TO off. Для задания настройки по умолчанию для БД необходимо выполнить команду:

```
ALTER DATABASE mydb SET geqo TO off;
```

Выполнение команды сохранит настройку (но не применит немедленно). При последующих подключениях к данной БД поведение будет соответствовать выполнению команды SET geqo TO off непосредственно перед началом сессии. Следует отметить, что пользователи могут изменить в своих сессиях эту настройку, которая только является значением по умолчанию. Для отмены внесенного изменения используется команда:

```
ALTER DATABASE dbname RESET varname;
```

11.5. Удаление БД

БД удаляются командой:

```
DROP DATABASE name;
```

Только владелец БД или суперпользователь может удалить БД. При удалении БД удаляются все содержащиеся в ней объекты. Откат удаления БД невозможен.

Выполнение команды `DROP DATABASE` невозможно из соединения с самой удаляемой БД. Однако, можно подключиться к любой другой БД, включая `template1`. БД `template1` является единственным средством для удаления последней пользовательской БД в кластере.

Для удобства предоставляется программа оболочки командой строки `dropdb` для удаления БД:

```
dropdb dbname
```

В отличие от `createdb` при отсутствии имени БД программа по умолчанию не удаляет БД с именем текущего пользователя.

11.6. Табличные пространства

Табличные пространства в PostgreSQL предоставляют администраторам СУБД возможность определять в ФС места для хранения файлов, представляющих объекты БД. На созданное табличное пространство можно ссылаться по имени при создании объектов БД.

Используя табличные пространства администратор может управлять размещением СУБД PostgreSQL на диске, что полезно по меньшей мере по двум причинам. Во-первых, если в разделе или томе, на котором был инициализирован кластер заканчивается свободное место и он не может быть расширен, на другом разделе может быть создано табличное пространство, которое можно использовать до переконфигурации системы.

ВНИМАНИЕ! Несмотря на то, что табличные пространства находятся за пределами основного каталога PostgreSQL данных, они являются неотъемлемой частью кластера базы данных и не может рассматриваться в качестве автономного набора файлов данных. Они зависят от метаданных, содержащиеся в системном каталог, и, следовательно, не могут быть присоединены к другому кластеру базы данных или к резервной копии по отдельности. Точно так же, если вы потеряете часть файлов табличного пространства (при удалении файлов, сбоя диска, и т.д.), кластер базы данных может стать нечитаемым или не в состоянии быть запущенным. Размещение табличного на временной файловой системы (как виртуальный диск) повышает надежность всего кластера.

Во-вторых, табличные пространства позволяют администратору использовать знания схемы применения объектов БД для оптимизации производительности. Например,

интенсивно используемый индекс может быть размещен на быстром диске высокой доступности, таком как дорогостоящий твердотельный диск. В то же время, таблица для хранения архивных данных, которая редко используются или не критична для производительности, может храниться на менее дорогой и более медленной дисковой системе.

Для определения табличного пространства используется команда `CREATE TABLESPACE`:

```
CREATE TABLESPACE fastspace LOCATION '/ssd1/postgresql/data';
```

Местом размещения должна быть существующая пустая директория, принадлежащая системному пользователю PostgreSQL. Все объекты, создаваемые впоследствии в табличном пространстве, будут храниться в файлах внутри этой директории. Место хранения не должно быть расположено на съемном или промежуточном носителе, т.к. кластер может не работать, если табличное отсутствует или потеряно.

Примечание. Обычно отсутствует смысл в создании более одного табличного пространства в одной логической ФС, поскольку невозможно контролировать расположение отдельных файлов в пределах логической ФС. Однако, PostgreSQL не накладывает подобных ограничений и фактически явно не осведомлен о границах ФС в ОС. PostgreSQL просто хранит файлы в указанных директориях.

Создание табличного пространства должно выполняться суперпользователем СУБД, но после этого можно разрешить обычным пользователям использовать данное табличное пространство, предоставив на него право `CREATE`.

Таблицы, индексы и целые БД могут быть назначены определенному табличному пространству. Для чего пользователь, имеющий право `CREATE` на табличное пространство, должен передать имя табличного пространства как параметр соответствующей команды. Например, создание таблицы в табличном пространстве `space1`:

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

Альтернативой является использование параметра `default_tablespace`:

```
SET default_tablespace = space1;
CREATE TABLE foo(i int);
```

При установке для параметра `default_tablespace` в качестве значения непустой строки значения, обеспечивается неявное задание `TABLESPACE` для команд `CREATE TABLE` и `CREATE INDEX`, которые не содержат явного задания табличного пространства.

Существует также параметр `temp_tablespaces`, который определяет расположение временных таблиц и индексов, а также временных файлов, используемых в таких целях как сортировка больших объемов данных. Значением параметра может быть список имен табличных пространств, а не только одно имя, и, следовательно нагрузка, связанная с временными объектами может быть распределена по нескольким табличным пространствам.

Случайный член списка выбирается при каждом создании временного объекта.

Табличное пространство, ассоциированное с БД, используется для хранения системных каталогов данной БД. Кроме того, оно является табличным пространством по умолчанию для таблиц, индексов и временных файлов, создаваемых в этой БД, если не задано значение TABLESPACE или другой выбор не определен значением default_tablespace или temp_tablespaces. Если БД создается без указания табличного пространства для нее, используется табличное пространство БД шаблона.

Два табличных пространства автоматически создаются initdb. Табличное пространство pg_global используется для разделяемых системных каталогов. Табличное пространство pg_default является табличным пространством по умолчанию для БД template1 и template0. И, следовательно, является табличным пространством по умолчанию для всех других БД также, если не переопределено значением TABLESPACE в команде CREATE DATABASE.

После создания табличное пространство может быть использовано из любой БД при наличии у запрашивающего пользователя достаточных прав, что означает, что табличное пространство не может быть удалено, пока все объекты во всех БД, использующих данное табличное пространство, не будут удалены.

Для удаления пустого табличного пространства используется команда DROP TABLESPACE.

Для определения перечня существующих табличных пространств необходимо проверить системный каталог pg_tablespace, например:

```
SELECT spcname FROM pg_tablespace;
```

Метакоманда \db программы psql также применима для перечисления существующих табличных пространств.

PostgreSQL применяет символические ссылки для упрощения работы с табличными пространствами, что означает, что табличные пространства могут быть использованы только в системах, поддерживающих символические ссылки.

Директория $PGDATA/pg_tblspc содержит символические ссылки, которые указывают на все не встроенные табличные пространства, определенные в кластере. Хотя это не рекомендуется, но возможно устанавливать расположение табличных пространств вручную, переопределяя данные ссылки с учетом следующих двух требований: не выполнять данную операцию во время работы сервера, после перезапуска сервера необходимо обновить каталог pg_tablespace, для указания новых местоположений, в противном случае pg_dump будет показывать старое расположение табличных пространств.

12. ЛОКАЛИЗАЦИЯ

Далее описаны доступные возможности по локализации с точки зрения администратора. PostgreSQL поддерживает два подхода к локализации:

- использование возможностей ОС по локализации для обеспечения специфических порядков сортировки, форматирования записи чисел, переведенных сообщений и других аспектов;
- представление наборов символов для поддержки хранения текста на всех видах языков, и предоставление возможности перевода наборов символов между клиентом и сервером.

12.1. Поддержка локализации

Поддержка локализации имеет отношение к региональным предпочтениям приложений, касающимся алфавитов, сортировки, форматирования чисел и прочего. В PostgreSQL используются стандартные средства ISO C и POSIX, поставляемые с ОС.

12.1.1. Обзор поддержки локализации

Поддержка локализации автоматически инициализируется при создании кластера БД командой `initdb`. Команда `initdb` инициализирует кластер БД, используя по умолчанию настройки локализации окружения исполнения, следовательно, если для ОС настроена локализация, необходимая в кластере БД, то дополнительная настройка не требуется. Если необходимо использовать другую локализацию (или неизвестна локализация, указанная в настройках системы), то можно указать `initdb` явно, какую локализацию следует использовать, с помощью опции `--locale`.

Пример

```
initdb --locale=sv_SE
```

В примере для Unix-систем устанавливается локализация для шведского языка (Swedish — sv), используемого в Швеции (SE). Другие примеры: `en_US` — английский в США или `fr_CA` — французский в Канаде. Если для локализации может быть использовано более одной таблицы символов, то запись будет выглядеть следующим образом: `language_territory.codeset`. Например, `fr_BE.UTF-8` представляет французский язык (fr) вариант для Бельгии (BE) с набором символов `UTF-8`.

В большей части Unix-систем список доступных локализаций может быть получен с помощью команды `locale -a`.

Иногда полезно смешивать правила нескольких локализаций, например, правила сортировки английского языка, но сообщения на испанском языке. Для поддержки данной возможности существует набор категорий локализации, которые управляют определенными

аспектами правил локализации (см. таблицу 106).

Таблица 106 – Категории локализации

Параметр	Описание
LC_COLLATE	Порядок сортировки строки
LC_CTYPE	Классификация символов
LC_MESSAGES	Язык сообщений
LC_MONETARY	Формат записи денежных единиц
LC_NUMERIC	Формат записи чисел
LC_TIME	Формат записи даты и времени

Имена категорий переводятся в имена опций initdb для переопределения настроек локализации конкретной категории. Например, для настройки французской локализации в Канаде, но с использованием правил записи денежных единиц США, применяется следующая команда:

```
initdb --locale=fr_CA --lc-monetary=en_US
```

Если необходимо обеспечить поведение системы, как системы без поддержки локализации, то используется специальная локализация C или POSIX.

Сущность некоторых категорий такова, что их значения должны быть зафиксированы при создании БД. Можно использовать различные настройки для разных БД, но после создания БД их изменение невозможно. К таким категориям относятся LC_COLLATE и LC_CTYPE, которые влияют на порядок сортировки индексов, и, следовательно должны быть зафиксированы во избежание повреждения индексов текстовых столбцов (эти ограничения могут быть смягчены использованием способов сортировки (collations), как описано в 12.2). Значения по умолчанию для данных категорий определяются во время работы initdb и используются при создании новых БД, если не указано иное в команде CREATE DATABASE.

Другие категории локализации могут быть изменены в любое время посредством установки значений конфигурационных параметров сервера СУБД, которые имеют такие же имена как и категории локализации (см. 8.11.2). Значения, определенные при выполнении команды initdb, в действительности только записываются в конфигурационный файл postgresql.conf, как значения по умолчанию, используемые при запуске сервера. При удалении данных установок из postgresql.conf, сервер наследует настройки из окружения исполнения.

Следует отметить, что поведение локализации сервера определяется значениями переменных окружения, полученными сервером, а не окружением какого-либо клиента. Таким образом, необходимо установить корректные настройки локализации до запуска сервера. В случае настройки различных локализаций у клиента и у сервера, возможно

появление сообщений на разных языках, в зависимости от источника сообщений.

Примечание. Наследование локализации от окружения исполнения, для большинства ОС означает следующее. Для данной категории локализации, например для сортировки, проверяются переменные окружения в указанном порядке до тех пор, пока не найдена первая, для которой определено значение: LC_ALL, LC_COLLATE (или переменная, соответствующая категории), LANG. Если не задана ни одна из переменных окружения, используется значение локализации по умолчанию C.

Примечание. Некоторые библиотеки локализации сообщений проверяют также переменную окружения LANGUAGE, которая переопределяет все прочие настройки локализации для указания языка сообщений. При возникновении сомнений необходимо обратиться к документации на ОС, в части касающейся gettext.

Для разрешения перевода сообщений на предпочитаемый пользователем язык, необходимо включить NLS во время сборки (configure –enable-nls). Прочая поддержка локализации включается автоматически.

12.1.2. Поведение локализации

Настройки локализации влияют на следующие характеристики SQL:

- порядок сортировки в запросах, использующих ORDER BY или стандартные операторы сравнения над текстовыми данными;
- функции upper, lower и initcap;
- операторы сравнения по шаблону (LIKE, SIMILAR TO и регулярные выражения POSIX); локализация влияет как на сравнения в зависимости от регистров символов, так и на классификацию символов классами регулярных выражений;
- семейство функций to_char;
- возможность использовать индексы в выражениях LIKE.

Недостаток использования в PostgreSQL локализаций, отличных от C или POSIX, заключается во влиянии на производительность: снижению скорости обработки символов и невозможности использовать обычные индексы в выражениях LIKE. По указанной причине локализацию следует использовать только если это действительно необходимо.

В качестве обходного пути для разрешения использования в PostgreSQL индексов для не C локализаций существует несколько классов операторов, которые позволяют создавать индексы, выполняющие строгое посимвольное сравнение, игнорируя правила сравнения локализации.

12.1.3. Проблемы локализации

Если поддержка локализации не работает в соответствии с приведенными выше пояснениями, необходимо проверить правильность конфигурирования локализации в ОС.

Для проверки установленных в ОС локализаций можно использовать команду `locale -a`.

Необходимо убедиться, что PostgreSQL действительно использует требуемую локализацию. Значения категорий `LC_COLLATE` и `LC_CTYPE` определяются при создании БД и не могут быть изменены. Другие категории локализации, включая `LC_MESSAGES` и `LC_MONETARY`, изначально определяются окружением, в котором запущен сервер, но могут быть изменены во время работы. Действующие настройки локализации можно проверить с помощью команды `SHOW`.

Каталог `src/test/locale` в дистрибутиве исходных кодов содержит набор тестов поддержки локализаций в PostgreSQL.

Клиентские приложения, которые обрабатывают ошибки сервера, разбирая текст сообщения об ошибке, очевидно столкнутся с проблемами, при получении сообщений сервера на другом языке. Авторам подобных приложений рекомендуется использовать схему кодирования ошибок вместо разбора текста сообщений.

12.2. Поддержка способов сортировки

Поддержка способов сортировки дает возможность задать порядок сортировки и поведение при классификации символов отдельно для каждого столбца или операции. Это позволяет снизить ограничения, связанные с невозможностью изменения параметров `LC_COLLATE` и `LC_CTYPE` после создания базы данных.

12.2.1. Концепция

Каждое выражение с сортируемыми типами данных имеет способ сортировки. Встроенными сортируемыми типами данных являются `text`, `varchar` и `char`. Пользовательские типы данных также могут быть помечены как сортируемые, кроме того, домен, созданный из сортируемого типа данных, считается сортируемым. Если выражение является ссылкой на столбец, то способом сортировки выражения считается способ сортировки, заданный для этого столбца. Если выражение является константой, способом сортировки выражения считается способ сортировки по умолчанию для типа данных константы. Способ сортировки более сложных выражений определяется по способам сортировки их аргументов.

Способом сортировки выражения может быть значение `"default"`, обозначающее использование настроек локализации, определенных для БД. Способ сортировки выражения может быть неопределенным. В этом случае, операции сортировки или операции, которые используют способ сортировки, вызовут ошибку.

При выполнении сортировки или классификации символов СУБД использует способ сортировки входного выражения. Это происходит, например, при выполнении выражения `ORDER BY` или функций и операторов с именами типа <. В качестве способа сортировки, применяемого для `ORDER BY` выбирается способ сортировки ключа сортировки. Способ

сортировки, применяемый при вызове функций или операторов, определяется по их аргументам, как описано выше. Помимо операторов сравнения, способы сортировки используются функциями конвертирования между верхним и нижним регистром символов, такими как `lower`, `upper` и `initcap`; операторами сопоставления с шаблоном; `to_char` и похожими функциями.

При вызове функции или оператора для выполнения запрошенной операции используется способ сортировки, определяемый по способам сортировки аргументов. Если результатом выполнения функции или оператора является сортируемый тип данных, его способ сортировки используется во время анализа в качестве заданного способа сортировки выражения функции или оператора, если окружающее их выражение требует информацию об их способе сортировки.

Определение способа сортировки выражения может быть неявным и явным. Отличие заключается в процедуре определения сочетания нескольких способов сортировки, когда они встречаются в одном выражении. Явное определение способа сортировки возникает при использовании предложения `COLLATE`; во всех остальных случаях используется неявное определение. Если требуется объединение нескольких способов сортировки, например для вызова функции, используются следующие правила:

1) Если какие-нибудь входные выражения имеют явное определение способа сортировки, то все явно полученные способы сортировки среди входных выражений должны быть одинаковыми, иначе возникает ошибка. В случая наличия какого-нибудь явно полученного способа сортировки, он принимается в качестве результата комбинации способов сортировки.

2) Иначе, все входные выражения должны иметь одинаковые неявно полученные способы сортировки или способ сортировки по умолчанию. Если присутствует какой-нибудь способ сортировки отличный от способа сортировки по умолчанию, он принимается в качестве результата комбинации способов сортировки. Иначе, результатом является способ сортировки по умолчанию.

3) В случае наличия конфликтующих способов сортировки, отличных от способа сортировки по умолчанию, их комбинация считается неопределенной. Это не является ошибкой, пока не будет вызвана функция, требующая информацию о способе сортировки. В этом случае во время выполнения будет вызвана ошибка.

Например, рассмотрим следующее определение таблицы:

```
CREATE TABLE test1 (
    a text COLLATE "de_DE",
    b text COLLATE "es_ES",
    ...
```

```
);
```

Тогда в запросе:

```
SELECT a < 'foo' FROM test1;
```

сравнение < выполняется в соответствии с правилами DE, поскольку выражение содержит комбинацию неявно полученного способа сортировки со способом сортировки по умолчанию. Но в запросе:

```
SELECT a < ('foo' COLLATE "fr_FR") FROM test1;
```

сравнение выполняется с использование правил fr_FR, т.к. явное определение способа сортировки перекрывает неявное. Более того, в запросе:

```
SELECT a < b FROM test1;
```

анализатор не может определить, какой из способов сортировки применять, поскольку столбцы a и b имеют конфликтующие неявные определения способа сортировки. В связи с тем, что оператор < требует информацию о способе сортировки, будет вызвана ошибка. Ошибка может быть устранена путем явного задания способа сортировки любому из входных выражений следующим образом:

```
SELECT a < b COLLATE "de_DE" FROM test1;
```

или эквивалентным:

```
SELECT a COLLATE "de_DE" < b FROM test1;
```

С другой стороны, структурно похожее выражение:

```
SELECT a || b FROM test1;
```

не вызовет ошибку, поскольку оператор || и его результат не используют способы сортировки.

Способ сортировки, сопоставленный с комбинацией входных выражений функции или оператора, принимается к результату их выполнения, если этот результат относится к сортируемому типу данных, например в запросе:

```
SELECT * FROM test1 ORDER BY a || 'foo';
```

сортировка будет выполнена в соответствии с правилами DE, но следующий запрос:

```
SELECT * FROM test1 ORDER BY a || b;
```

вызовет ошибку, потому что, несмотря на то, что оператор || не требует информации о способе сортировки, предложение ORDER BY ее требует. Как и раньше, конфликт может быть разрешен явным заданием способа сортировки:

```
SELECT * FROM test1 ORDER BY a || b COLLATE "fr_FR";
```

12.2.2. Управление способами сортировки

Способ сортировки является объектом SQL-схемы, который отображает SQL-имя в настройки локализации ОС. Обычно оно отображается в комбинацию LC_COLLATE и LC_CTYPE. Также способ сортировки связан с набором символов (см. 12.3). Для различных

наборов символов может существовать способ сортировки с одним и тем же именем.

На всех платформах доступны способы сортировки default, C и POSIX. Набор дополнительных способов сортировки зависит от ОС. Способ сортировки по умолчанию использует значения LC_COLLATE и LC_CTYPE, заданные для БД при ее создании. Способы сортировки C и POSIX задают «традиционное C» поведение, при котором символами считаются только символы ASCII от "A" до "Z", и сортировка выполняется строго по значению их кодов.

Если ОС обеспечивает поддержку использования нескольких локализаций в одной программе (newlocale и подобные функции), при инициализации кластера БД initdb заполняет системный каталог pg_collation способами сортировки, основанными на всех вариантах локализации, доступных в ОС на то время. Например, ОС может обеспечивать поддержку локализации de_DE.utf8. initdb создаст способ сортировки de_DE.utf8 для кодировки UTF8, имеющий LC_COLLATE и LC_CTYPE, установленные в значение de_DE.utf8. Дополнительно создается способ сортировки без тега .utf8. Таким образом, создается возможность использования способа сортировки de_DE, который менее громоздкий для записи, и делает запись менее зависимой от кодировки символов. Тем не менее, изначальный набор способов сортировки зависит от платформы.

При необходимости использования способа сортировки от другими значениями LC_COLLATE и LC_CTYPE, может быть создан новый способ сортировки командой CREATE COLLATION. Также эта команда может быть применена для создания нового способа сортировки на основе существующего, что может быть удобно для возможности использования в приложениях имен способов сортировки, не зависящих от ОС.

В пределах обычной БД интерес представляют только способы сортировки, использующие кодировку самой БД. Другие записи в pg_collation игнорируются. Так что, укороченное имя способа сортировки de_DE может рассматриваться уникальным в пределах данной БД, даже если оно не является уникальным глобально. Рекомендуется использовать укороченные имена способов сортировки, т.к. в случае смены кодировки БД, не потребуется вносить изменения. Следует отметить, что способы сортировки default, C и POSIX могут быть использованы независимо от кодировки БД.

PostgreSQL считает различные способы сортировки несовместимыми, даже если они имеют идентичные свойства. Например:

```
SELECT a COLLATE "C" < b COLLATE "POSIX" FROM test1;
```

вызовет ошибку, несмотря на то, что C и POSIX обладают идентичным поведением. В связи с этим, одновременное использование полных и укороченных имен способов сортировки не рекомендуется.

12.3. Поддержка наборов символов

Поддержка наборов символов в PostgreSQL позволяет хранить текст в различных кодировках, включая однобайтовые наборы символов, такие как серия ISO 8859, и многобайтовые наборы символов, такие как EUC (Extended Unix Code), UTF-8 и внутренний код Mule. Все поддерживаемые наборы символов могут быть прозрачно использованы клиентами, но несколько из них не поддерживаются сервером (т.е. как серверные кодировки). Набор символов по умолчанию выбирается при инициализации кластера БД PostgreSQL командой initdb. Он может быть переопределен при создании БД, следовательно, существует возможность иметь БД с различной кодировкой.

Однако, существует важное ограничение, что выбранный набор символов БД должен быть совместим с настройками категорий локализации LC_CTYPE и LC_COLLATE для БД. Для локализации C или POSIX допустим любой набор символов, но для других локализаций правильно работать будет только один набор символов.

12.3.1. Поддерживаемые наборы символов

Таблица 107 содержит наборы символов, доступные в PostgreSQL.

Таблица 107 – Наборы символов PostgreSQL

Наименование	Описание	Язык	Сервер	Байтов на символ	Псевдонимы
BIG5	Большая пятерка	Традиционный китайский	Нет	2	WIN950, Windows950
EUC_CN	Расширенный UNIX Code-CN	Упрощенный китайский	Да	3	
EUC_JP	Расширенный UNIX Code-JP	Японский	Да	3	
EUC_JIS_2004	Расширенный UNIX Code-JP, JIS X 0213	Японский	Да	3	
EUC_KR	Расширенный UNIX Code-KR	Корейский	Да	3	
EUC_TW	Расширенный UNIX Code-TW	Традиционный китайский, тайваньский	Да	3	
GB18030	Национальный стандарт	Китайский	Нет	2	
GBK	Расширенный национальный стандарт	Упрощенный китайский	Нет	2	WIN936, Windows936
ISO_8859_5	ISO-8859-5, ECMA 113	Латиница/кириллица	Да	1	
ISO_8859_6	ISO-8859-6, ECMA 114	Латиница/арабский	Да	1	
ISO_8859_7	ISO-8859-7, ECMA 118	Латиница/греческий	Да	1	

Окончание таблицы 107

Наименование	Описание	Язык	Сервер	Байтов на символ	Псевдонимы
ISO_8859_8	ISO-8859-8, ECMA 121	Латиница/иврит	Да	1	
JOHAB	JOHAB	Корейский (Hangul)	Нет	3	
KOI8R	KOI8-R	Кириллица (Россия)	Да	1	KOI8
KOI8U	KOI8-U	Кириллица (Украина)	Да	1	
LATIN1	ISO-8859-1, ECMA 94	Западная Европа	Да	1	ISO88591
LATIN2	ISO-8859-2, ECMA 94	Центральная Европа	Да	1	ISO88592
LATIN3	ISO-8859-3, ECMA 94	Южная Европа	Да	1	ISO88593
LATIN4	ISO-8859-4, ECMA 94	Северная Европа	Да	1	ISO88594
LATIN5	ISO-8859-9, ECMA 128	Турецкий	Да	1	ISO88599
LATIN6	ISO-8859-10, ECMA 144	Скандинавия	Да	1	ISO885910
LATIN7	ISO-8859-13	Прибалтика	Да	1	ISO885913
LATIN8	ISO-8859-14	Кельтский	Да	1	ISO885914
LATIN9	ISO-8859-15	LATIN1 с знаком евро и диакритическими знаками	Да	1	ISO885915
LATIN10	ISO-8859-16, ARSO SR 14111	Румынский	Да	1	ISO885916
MULE_INTERNAL	Внутренний код Mule	Многоязычный Emacs	Да	4	
SJIS	Shift JIS	Японский	Нет	2	Mskanji, ShiftJIS, WIN932, Windows932
SHIFT_JIS_2004	Shift JIS, JIS X 0213	Японский	Нет	2	
SQL_ASCII	не определено	Любой	Да	1	
UHC	Унифицированный код Hangul	Корейский	Нет	2	WIN949, Windows949
UTF8	Unicode, 8-bit	Все	Да	4	Unicode
WIN866	Windows CP866	Кириллица	Да	1	ALT
WIN874	Windows CP874	Тайский	Да	1	
WIN1250	Windows CP1250	Центральная Европа	Да	1	
WIN1251	Windows CP1251	Кириллица	Да	1	WIN
WIN1252	Windows CP1252	Западная Европа	Да	1	
WIN1253	Windows CP1253	Греческий	Да	1	
WIN1254	Windows CP1254	Турецкий	Да	1	
WIN1255	Windows CP1255	Иврит	Да	1	
WIN1256	Windows CP1256	Арабский	Да	1	
WIN1257	Windows CP1257	Прибалтика	Да	1	
WIN1258	Windows CP1258	Вьетнамский	Да	1	ABC, TCVN, TCVN5712, VSCII

Не все интерфейсы прикладного программирования поддерживают все наборы символов. Например, драйвер JDBC для PostgreSQL не поддерживает MULE_INTERNAL, LATIN6, LATIN8 и LATIN10.

Поведение с SQL_ASCII значительно отличается от поведения с остальными наборами символов. При установке в качестве серверного набора символов SQL_ASCII, сервер интерпретирует значения 0-127 в соответствии со стандартом ASCII, а значения 128-255 рассматриваются как неинтерпретируемые символы. При SQL_ASCII не производится никакого преобразования кодировок. Таким образом, данная настройка является не столько декларацией использования специфической кодировки, сколько заявлением об игнорировании кодировки. В большинстве случаев, если обрабатываются данные не ASCII, не рекомендуется выбирать SQL_ASCII, поскольку PostgreSQL не сможет помочь в конвертировании или проверке символов не ASCII.

12.3.2. Выбор набора символов

Набор символов по умолчанию для кластера PostgreSQL определяется с помощью initdb.

Пример

```
initdb -E EUC_JP
```

В приведенном примере в качестве значения по умолчанию задается набор символов (кодировка) EUC_JP (расширенный Unix Code для японского языка). Можно использовать опцию --encoding вместо -E. Если ни одна из названных опций не задана, initdb пытается определить подходящую для использования кодировку на основе заданной локализации или локализации по умолчанию.

При создании БД можно указать кодировку, отличную от значения по умолчанию, при условии, что эта кодировка совместима с выбранной локализацией:

```
createdb -E EUC_KR -T template0 --lc-collate=ko_KR.euckr \
  --lc-ctype=ko_KR.euckr korean
```

Таким образом, будет создана БД с именем korean, для которой используются набор символов EUC_KR и локализация ko_KR. Другим способом является использование команды SQL:

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR' COLLATE='ko_KR.euckr' \
  CTYPE='ko_KR.euckr' TEMPLATE=template0;
```

Кодировка БД хранится в системном каталоге pg_database. Кодировку можно узнать, используя опцию -l или метакоманду \l программы psql.

Пример

```
$ psql -l
```

```
                                    List of databases
   Name    |  Owner   | Encoding  |  Collation  |    Ctype    |          Acc \
ess Privileges
-----------+----------+-----------+-------------+-------------+------------- \
-----------------------
 clocaledb | hlinnaka | SQL_ASCII | C           | C           |
 englishdb | hlinnaka | UTF8      | en_GB.UTF8  | en_GB.UTF8  |
 japanese  | hlinnaka | UTF8      | ja_JP.UTF8  | ja_JP.UTF8  |
 korean    | hlinnaka | EUC_KR    | ko_KR.euckr | ko_KR.euckr |
 postgres  | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
 template0 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/hlinnaka, \
hlinnaka=CTc/hlinnaka}
 template1 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/hlinnaka, \
hlinnaka=CTc/hlinnaka}
(7 rows)
```

В большинстве современных ОС PostgreSQL может определить, какой набор символов кодовая таблица подразумевается значением `LC_CTYPE`, и контролировать, использование для БД только соответствующей кодировки. В старых системах ответственность за использование набора символов соответствующего выбранной локализации возлагалась на пользователя. Ошибки в данной области могут привести к аномальному поведению при выполнении операций, зависящих от локализации, таких, как сортировка.

PostgreSQL позволяет суперпользователям создавать БД с кодировкой `SQL_ASCII`, даже если значение `LC_CTYPE` отлично от `C` или `POSIX`. Как было замечено выше, `SQL_ASCII` не требует, чтобы данные, хранимые в БД, имели какую-либо определенную кодировку, следовательно, такой выбор может создает риск аномального поведения, связанного с локализацией. Не рекомендуется использовать такую комбинацию настроек.

12.3.3. Автоматическое преобразование наборов символов между сервером и клиентом

PostgreSQL поддерживает автоматическое преобразование наборов символов между сервером и клиентом для определенных сочетаний наборов символов. Информация для преобразования хранится в системном каталоге `pg_conversion`. PostgreSQL поставляется с предопределенными преобразованиями, показанными в таблице 108. Можно создать новое преобразование, используя команду SQL `CREATE CONVERSION`.

Таблица 108 – Преобразование наборов символов между клиентом и сервером

Наборов символов сервера	Доступные наборы символов клиента
BIG5	Не поддерживается в качестве серверной кодировки.
EUC_CN	EUC_CN, MULE_INTERNAL, UTF8.
EUC_JP	EUC_JP, MULE_INTERNAL, SJIS, UTF8.
EUC_KR	EUC_KR, MULE_INTERNAL, UTF8.
EUC_TW	EUC_TW, BIG5, MULE_INTERNAL, UTF8.
GB18030	Не поддерживается в качестве серверной кодировки.
GBK	не поддерживается в качестве серверной кодировки.
ISO_8859_5	ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN866, WIN1251.
ISO_8859_6	ISO_8859_6, UTF8.
ISO_8859_7	ISO_8859_7, UTF8.
ISO_8859_8	ISO_8859_8, UTF8.
JOHAB	JOHAB, UTF8.
KOI8R	KOI8R, ISO_8859_5, MULE_INTERNAL, UTF8, WIN866, WIN1251.
KOI8U	KOI8U, UTF8.
LATIN1	LATIN1, MULE_INTERNAL, UTF8.
LATIN2	LATIN2, MULE_INTERNAL, UTF8, WIN1250.
LATIN3	LATIN3, MULE_INTERNAL, UTF8.
LATIN4	LATIN4, MULE_INTERNAL, UTF8.
LATIN5	LATIN5, UTF8.
LATIN6	LATIN6, UTF8.
LATIN7	LATIN7, UTF8.
LATIN8	LATIN8, UTF8.
LATIN9	LATIN9, UTF8.
LATIN10	LATIN10, UTF8.
MULE_INTERNAL	MULE_INTERNAL, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8, LATIN1 to LATIN4, SJIS, WIN866, WIN1250, WIN1251 .
SJIS	Не поддерживается в качестве серверной кодировки.
SQL_ASCII	Любые (преобразование не выполняется).
UHC	Не поддерживается в качестве серверной кодировки.
UTF8	Все поддерживаемые кодировки.
WIN866	WIN866, ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN1251
WIN874	WIN874, UTF8.
WIN1250	WIN1250, LATIN2, MULE_INTERNAL, UTF8.
WIN1251	WIN1251, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN866.

Окончание таблицы 108

Наборов символов сервера	Доступные наборы символов клиента
WIN1252	WIN1252, UTF8.
WIN1253	WIN1253, UTF8.
WIN1254	WIN1254, UTF8.
WIN1255	WIN1255, UTF8.
WIN1256	WIN1256, UTF8.
WIN1257	WIN1257, UTF8.
WIN1258	WIN1258, UTF8.

Для включения автоматического преобразования наборов символов необходимо указать PostgreSQL набор символов (кодировку), который будет использовать клиент. Существует несколько способов выполнения данной операции:

1) Метакомандой `\encoding` в `psql`. Метакоманда `\encoding` позволяет изменять кодировку клиента во время работы. Например, для изменения кодировки на SJIS необходимо ввести:

```
\encoding SJIS
```

2) Функцией из библиотеки `libpq` для управления кодировкой клиента.

3) Командой SQL `SET client_encoding TO`:

```
SET CLIENT_ENCODING TO 'value';
```

Кроме того, можно использовать стандартный синтаксис SQL `SET NAMES`:

```
SET NAMES 'value';
```

Для запроса текущей кодировки клиента:

```
SHOW client_encoding;
```

Для возврата значения по умолчанию:

```
RESET client_encoding;
```

4) Переменной окружения `PGCLIENTENCODING`. Если переменная окружения `PGCLIENTENCODING` определена в клиентском окружении, то кодировка клиента автоматически выбирается при установке соединения с сервером. Указанное в переменной значение впоследствии может быть изменено любым из вышеперечисленных способов.

5) Конфигурационной переменной `client_encoding`. Если задано значение переменной `client_encoding`, то кодировка клиента автоматически выбирается при установке соединения с сервером. Указанное в переменной значение впоследствии может быть изменено любым из вышеперечисленных способов.

Если преобразование конкретного символа невозможно, например, выбраны наборы символов `EUC_JP` для сервера и `LATIN1` для клиента, но некоторые японские символы не

имеют представления в LATIN1, то будет выдано сообщение об ошибке.

Если определен набор символов клиента SQL_ASCII, то преобразование кодировок отключается независимо от выбранного набора символов сервера. Аналогично для сервера, не рекомендуется использовать SQL_ASCII, если не все обрабатываемые данные кодируются в ASCII.

13. ПЛАНОВЫЕ ЗАДАЧИ ОБСЛУЖИВАНИЯ СУБД

PostgreSQL, подобно любой другой СУБД, требует регулярного выполнения определенных задач для достижения оптимальной производительности.

Очевидным примером задачи обслуживания является регулярное создание резервных копий данных. Отсутствие актуальной резервной копии делает невозможным восстановление поврежденных данных после отказа (например, выхода из строя диска), пожара, случайного уничтожением критической таблицы. Подробное описание механизмов резервного копирования и восстановления, доступных в PostgreSQL, приведено в 14.

Другой важной задачей обслуживания является периодический вакууминг БД, описанный в 13.1.1. С данной задачей тесно связано обновление статистики, используемой планировщиком запросов, рассматриваемое в 13.1.3.

Кроме того, периодического внимания требует управление файлом журнала, описанное в 13.3.

Существует скрипт `check_postgres.pl` для мониторинга состояния СУБД и выдачи сообщений о нештатных ситуациях. Скрипт `check_postgres.pl` интегрирован с Nagios и MRTG, но может быть использован и самостоятельно.

PostgreSQL по сравнению с некоторыми другими СУБД не требует большого объема технического обслуживания.

13.1. Плановый вкууминг

БД PostgreSQL требуют периодического обслуживания, называемого вакууминг. В многих конфигурациях достаточно разрешить проведение вакуумминга демону `autovacuum` (см. 13.1.6). Для достижения наилучшего результата необходимо настроить параметры автовакуумминга. Некоторые администраторы СУБД предпочитают дополнять или заменять действия данного демона командами `VACUUM`, управляемыми вручную, которые обычно выполняются в соответствии со скриптами `cron` или иного планировщика задач.

13.1.1. Общее описание вакуумминга

Команда `VACUUM` СУБД PostgreSQL должна регулярно обрабатывать каждую таблицу по следующим причинам:

1) Для освобождения или повторного использования дискового пространства, занимаемого обновленными или удаленными строками.

2) Для обновления статистической информации, используемой планировщиком запросов PostgreSQL.

3) Для обновления карты видимости, ускоряющей работу механизма доступа `index-only`.

4) Для защиты от потери старых данных вследствие циклического переполнения счетчика транзакций.

Каждая из указанных причин требует проведения операций VACUUM с различной частотой и в различных пределах. Более подробное описание приведено далее.

Существует два варианта команды VACUUM: стандартная команда VACUUM и VACUUM FULL. Команда VACUUM FULL может освободить больше дискового пространства, но и работает значительно медленнее. Кроме того, стандартная команда VACUUM может выполняться параллельно с рабочими операциями БД. (Команды SELECT, INSERT, UPDATE и DELETE продолжают нормально функционировать, хотя модификация определения таблицы такими командами как ALTER TABLE, будет невозможна при проведении операции вакуумминга для таблицы.) Команда VACUUM FULL требует эксклюзивной блокировки таблицы, с которой работает, следовательно, параллельное выполнение других команд невозможно. В общем случае, администраторы должны стараться использовать стандартную команду VACUUM избегая VACUUM FULL.

Команда VACUUM порождает существенное количество трафика ввода/вывода, что может привести к снижению производительности для других активных сессий. В 8.4.4 описаны значения конфигурационных параметров, которые могут быть установлены для уменьшения влияния фонового вакуумминга на производительность.

13.1.2. Освобождение дискового пространства

В PostgreSQL выполнение команды UPDATE или DELETE для строки не производит немедленное удаление старой версии строки. Данный подход необходим для получения преимуществ многоверсионного управления конкурентным доступом: версия строки не должна удаляться, пока она потенциально видима в других транзакциях. Но в конечном итоге устаревшая или удаленная версия строки более не представляет интереса для транзакций. Занимаемое такой строкой пространство должно быть освобождено для повторного использования новыми строками во избежание роста необходимого дискового пространства. Команда VACUUM и выполняет описанное освобождение.

Стандартная форма VACUUM удаляет ненужные версии строки в таблицах и индексах и отмечает пространство, доступное для повторного использования. Однако, данное пространство не возвращается ОС за исключением особого случая, когда одна или несколько страниц в конце таблицы становятся полностью свободными и может быть легко выполнена эксклюзивная блокировка. В противоположность, VACUUM FULL активно уплотняет таблицы путем создания новой версии таблицы без пропусков. Это минимизирует размер таблицы, но может потребовать значительных временных затрат. Кроме того, до завершения операции требуется дополнительное место на диске для создания копии таблицы.

Обычной целью проведения планового вакуумминга является выполнение стандарт-

ных команд VACUUM с частотой, позволяющей избежать выполнения VACUUM FULL. Демон автовакуумминга пытается работать подобным образом, и в действительности никогда не выполняет VACUUM FULL. В данном подходе главное не сохранение минимального размера таблиц, а поддержка устойчивого состояния использования дискового пространства, при котором каждая таблица занимает дисковое пространство, эквивалентное ее минимальному размеру плюс дополнительное место, необходимое для нее между проведением операций вакуумминга. Хотя команда VACUUM FULL может быть использована для уменьшения размера таблицы до минимума и возврата освобожденного места ОС, данная операция не имеет особого смысла, поскольку размер таблицы снова увеличится в будущем. Таким образом, для обслуживания часто обновляемых таблиц умеренно частое использование стандартной команды VACUUM предпочтительнее редкого выполнения VACUUM FULL.

Некоторые администраторы предпочитают самостоятельно настраивать расписание для вакуумтинга, выполняя, например, все операции ночью, когда загрузка СУБД минимальна. Недостаток выполнения вакуумминга в соответствии с фиксированным расписанием заключается в возможности резкого увеличения количества операций по обновлению таблицы, и значительному увеличению ее размера, приводящему к необходимости запуска VACUUM FULL для высвобождения дискового пространства. Использование демона автовакуумминга облегчает решение данной проблемы, поскольку демон планирует проведение вакуумминга динамически в зависимости от числа операций обновления. Таким образом, не рекомендуется полностью отключать демон автовакуумминга при возможности непредсказуемого резкого повышения нагрузки. В качестве компромиссного решения предлагается настройка параметров демона для реакции только на нестандартное увеличение активности по обновлению с целью предотвращения выхода ситуации из-под контроля, ожидая выполнения основной работы плановыми командами VACUUM при нормальной нагрузке.

Если автовакуум не используется, наилучшим подходом является планирование ежедневного выполнения VACUUM для всей БД в период минимальной нагрузки, дополняемое при необходимости более частыми операциями вакуумминга для часто обновляемых таблиц. Некоторые конфигурации с экстремально высокой частотой обновлений выполняют вакуумминг таблиц каждые несколько минут. При наличии нескольких БД в кластере необходимо использовать VACUUM для каждой из них (применение программы vacuumdb может быть полезным).

Следует отметить, что применение ни одной из описанных форм VACUUM не будет достаточным для таблицы, содержащей большое количество старых версий строк, возникающих в результате массового выполнения операция обновления и удаления. При наличии подобной таблицы и необходимости освободить чрезмерно занимаемое дисковое пространство наилучшим подходом является использование команды CLUSTER или

одного из переписывающих таблицу вариантов команды ALTER TABLE. Данные команды полностью переписывают новую копию таблицы и строят для нее новые индексы. Подобно VACUUM FULL, они требуют эксклюзивной блокировки. Кроме того, они временно используют дополнительное дисковое пространство, поскольку старые версии таблицы и индексов не могут быть удалены до завершения создания новых.

При наличии таблицы, содержимое которой периодически полностью удаляется, следует рассмотреть применение команды TRUNCATE вместо использования DELETE с последующим VACUUM. Команда TRUNCATE удаляет все содержимое таблицы немедленно и не требует последующего запуска VACUUM или VACUUM FULL. Недостатком данного метода является нарушение строгой семантики многоверсионного управления конкурентным доступом.

13.1.3. Обновление статистической информации планировщика

Планировщик запросов PostgreSQL полагается на статистическую информацию о содержимом таблиц при создании хороших планов запросов. Данная статистическая информация собирается командой ANALYZE, которая вызывается самостоятельно или как опциональный этап при выполнении VACUUM. Важно, чтобы статистическая информация была достаточно точной, в противном случае выбор неудачных планов может провести к снижению производительности СУБД.

При включенном демоне автовакууминга автоматически выдаются команды ANALYZE при значительном изменении содержимого таблицы. Однако, администраторы могут предпочесть настройку расписания для операций ANALYZE вручную, в особенности когда известно, что обновление таблицы не затрагивает статистическую информацию по представляющим интерес столбцам. Расписание операций ANALYZE, составляемое демоном, является только функцией от числа добавленных или обновленных столбцов. Демон не имеет сведений приводили ли эти операции к значимому изменению статистической информации.

Как и при проведении вакуумирования для освобождения места, частые обновления статистической информации более важны для часто обновляемых таблиц чем для редко обновляемых таблиц. Однако, даже для часто обновляемых может не быть необходимости в обновлении статистической информации при незначительных изменениях в статистическом распределении данных. Простой метод эмпирического определения необходимости изменения статистической информации заключается в анализе количества изменений минимальных и максимальных значений в столбцах таблицы. Например, столбец timestamp, содержащий значения времени обновления строк, будет иметь постоянно возрастающее максимальное значение при каждом обновлении или добавлении строк, следовательно такой столбец будет вероятно нуждаться в более частом обновлении статистической информации, чем столбец, содержащий URL (Uniform Resource Locator) для страниц, доступных

на сайте. Столбец URL также может обновляться часто, но статистическое распределение его значений вероятно будет меняться относительно медленно.

Возможно выполнять команду ANALYZE для определенных таблиц и даже для отдельных столбцов для обеспечения более частого обновления определенной требуемой приложением статистической информации. На практике, однако, наилучшим решением зачастую является анализ всей БД, потому что данная операция является быстрой. Команда ANALYZE использует статистически случайный выбор строк таблицы, а не чтение каждой отдельной строки.

Несмотря на то, что настройка по столбцам частоты выполнения команды ANALYZE может быть не очень продуктивной, настройка уровня детализации по столбцам статистической информации, собираемой командой ANALYZE может оказаться полезной. Для столбцов, часто используемых в выражениях WHERE и содержащих данные с большим разбросом значений, могут потребоваться гистограммы данных с более высокой степенью детализации, чем для других столбцов. Настройка может быть выполнена командой ALTER TABLE SET STATISTICS или изменением используемого по умолчанию для всей СУБД значения конфигурационного параметра default_statistics_target.

По умолчанию существует ограниченная информация об избирательности функций. В то же время, при создании вычисляемого на основе функции индекса о функции может быть собрана полезная статистика, что может значительно улучшить планы использования подобного индекса.

Демон автовакууминга не выполняет ANALYZE для внешних таблиц, т.к. не имеет возможности определить полезность этого. При необходимости использования статистики для внешних таблиц для оптимального планирования следует использовать ручное управление запуском ANALYZE для таких таблиц с помощью подходящего планировщика.

13.1.4. Обновление карт видимости

Ваккууминг поддерживает карты видимости для каждой таблицы для хранения информации о страницах, содержащих только версии записей, видимые активными транзакциями (и будущими транзакциями, до момента модификации страницы). Существует два применения этого.

Во-первых, при следующем вызове операции вакууминга такие страницы пропускаются, т.к. не содержат версий записей для чистки.

Во-вторых, это позволяет PostgreSQL получать результаты некоторых запросов только с помощью индекса, без обращения к самой таблице. Поскольку индексы PostgreSQL не содержат информации о видимости версий записей, при обычном использовании индекса выбираются все версии записей из буфера таблицы с последующей проверкой их видимости для текущей транзакции. В противоположность этому, механизм доступа index-only прове-

ряет сначала карту видимости. Если известно, что все записи на странице видимы, выборка полной записи из буфера таблицы и ее проверка могут быть пропущены. Это особенно заметно на больших объемах данных, когда карта видимости предотвращает излишние обращения к дисковой подсистеме. Карта видимости требует значительно меньше памяти, чем буфер таблицы, и легко может быть закэширована даже для таблиц большого размера.

13.1.5. Предотвращение циклического переполнения счетчика идентификаторов транзакций

Семантика многоверсионного управления конкурентным доступом (MVCC) транзакций в PostgreSQL зависит от возможности сравнения идентификаторов транзакций (XID): версия строки вставки с `XID`, большим, чем `XID` текущей транзакции, «находится в будущем» и не должна быть видна для текущей транзакции. Но поскольку идентификаторы транзакций имеют ограниченный размер (в настоящее время 32 бита), кластер, проработавший достаточно долго (было более 4 миллиардов транзакций), может столкнуться с проблемой циклического переполнения счетчика транзакций: счетчик `XID` проходит нулевое значение, и все транзакции, которые только что были в прошлом, оказываются в будущем, что означает недоступность результатов выполнения транзакций и катастрофическую потерю данных. В действительности данные все еще существуют, но они недоступны. Для предотвращения возникновения подобной ситуации необходимо проводить вакууминг для каждой таблицы в каждой БД по меньшей мере через каждые 2 миллиарда транзакций.

Причина того, что периодический вакууминг решает проблему в том, что PostgreSQL сохраняет специальный XID `FrozenXID`. Этот XID не подчиняется обычным правилам сравнения XID и всегда считается старше любого обычного XID. Обычные XID сравниваются с помощью арифметики по модулю 232. Это означает, что для каждого нормального XID существуют два миллиарда XID, которые "старше"и два миллиарда, которые "новее"; еще один способ сказать, что это что обычное пространство XID пространство зациклено. Таким образом, как только версия строки была создана с определенным обычным XID, версия строки может появиться "в прошлом"для последующих двух миллиардов транзакций, независимо от того, о каком нормальном XID мы говорим. Если версия строки все еще существует после более чем два миллиарда транзакций, она неожиданно оказывается в будущем. Для того чтобы это предотвратить, эти старые версии строк должны быть переназначены на XID `FrozenXID` прежде чем они достигнут двух миллиардов следующих старых транзакций. Если хотя бы один раз они будут маркированы этим специальным XID, они будут появляться "в прошлом"для всех обычных транзакций и поэтому такие версии строк будут существовать, пока не будут удалены независимо от того, как долго длятся транзакции. Это переназначение старых XID обрабатывается в `VACUUM`.

Параметр `vacuum_freeze_min_age` контролирует насколько «старыми» должны

быть значения XID для того, чтобы быть заменены на FrozenXID. Увеличение этого параметра поможет избежать от ненужной работы, если строки, которые были заморожены недавно, были изменены, но уменьшение значения этого параметра увеличивает количество операций, которое должно пройти перед вакуумингом таблицы снова.

В обычном случае VACUUM пропускает страницы, которые не содержат никаких ненужных версий строк, но такие страницы все еще могут содержать версии строк со «старыми» XID. Для гарантирования замены всех «старых» XID на FrozenXID необходимо выполнять сканирование всей таблицы. Параметр vacuum_freeze_table_age определяет момент выполнения данного действия командой VACUUM: полная очистка таблицы производится если таблица не подвергалась полному сканированию vacuum_freeze_table_age - vacuum_freeze_min_age транзакций. Установка значения 0 параметра vacuum_freeze_table_age определяет, что VACUUM всегда сканирует все записи без учета карт видимости.

Максимальное время, в течение которого для таблицы может не проводиться вакууминг, составляет 2 миллиарда транзакций минус значение параметра vacuum_freeze_min_age, использованное при выполнении командой VACUUM последнего полного сканирования таблицы. Если операция вакуумминга для таблицы не будет проводиться дольше этого времени, возможна потеря данных. Для предотвращения потери данных автовакууминг вызывается для каждой таблицы, которая может содержать незамороженные строки, чей XID старше, чем возраст, определенный конфигурационным параметром autovacuum_freeze_max_age (Данная операция выполняется, даже если автовакууминг отключен).

Это означает, что, если для таблицы не проводится вакууминг другими способами, автовакууминг для таблицы вызывается приблизительно каждые autovacuum_freeze_max_age минус vacuum_freeze_min_age транзакций. Для таблиц, для которых регулярно проводится вакууминг в целях экономии дискового пространства, это неважно. Однако, для статических таблиц (включая таблицы, в которые данные добавляются, но не изменяются и не удаляются) нет необходимости производить вакууминг для высвобождения дискового пространства, и, следовательно, может быть полезно попытаться максимизировать интервал между вынужденными автовакуумингами для больших статических таблиц. Очевидно, что максимизация интервала может быть выполнена увеличением значения autovacuum_freeze_max_age или уменьшением значения vacuum_freeze_min_age.

Эффективным максимальным значением параметра vacuum_freeze_table_age является 0,95*autovacuum_freeze_max_age; установка большего значения приведет к округлению до максимального значения (значения autovacuum_freeze_max_age). Установка значения больше чем autovacuum_freeze_max_age не имеет смысла, поскольку в

целях предотвращения циклического переполнения счетчика транзакций автовакууминг в этой точке будет запускаться в любом случае, а множитель 0,95 оставляет некоторую возможность для выполнения команды VACUUM вручную. Наилучшим правилом является установка значения параметра vacuum_freeze_table_age несколько ниже значения autovacuum_freeze_max_age, оставляя достаточный интервал для выполнения VACUUM по расписанию или запуска в данном интервале автовакууминга вследствие выполнения операций удаления или обновления. Установка значения слишком близкого к autovacuum_freeze_max_age приведет к запускам автовакууминга с целью предотвращения циклического переполнения буфера, даже если для таблицы недавно был выполнен вакууминг с целью возврата дискового пространства, уменьшение значения может привести к более частым запускам полного сканирования таблицы.

Единственным недостатком увеличения значения autovacuum_freeze_max_age (и значения vacuum_freeze_table_age вместе с ним) является увеличение дискового пространства, занимаемого каталогом кластера БД pg_clog, поскольку она должна содержать статус завершения всех транзакций в интервале, определенном значением autovacuum_freeze_max_age. Для хранения статуса завершения используется два бита на транзакцию, следовательно при максимально допустимом значении параметра autovacuum_freeze_max_age около двух миллиардов, размер pg_clog может возрасти до 0,5 ГБ. Если данный объем дискового пространства незначителен по сравнению с общим размером БД, то рекомендуется устанавливать максимально допустимое значение параметра autovacuum_freeze_max_age. В противном случае, устанавливаемое значение зависит от ограничения на размер pg_clog. Значение по умолчанию, 200 миллионов транзакций, соответствует 50 МБ для pg_clog.

Еще одним недостатоком уменьшения vacuum_freeze_min_age является ситуация, в которой к VACUUMсделает бесполезную работу: замораживание версии строки — пустая трата времени, если строка изменена чуть позже (заставив ее приобрести новый XID). Поэтому значение параметра должно быть достаточно большим, чтобы строки маркировались FrozenXID, как только они перестанут изменяться.

Для того,чтобы отслеживать возраст старейших XIDs в базе данных, статистика VACUUM хранится XID в системных таблицах pg_class и pg_database. В частности, столбец relfrozenxid таблицы pg_class содержит XID, который использовался в последнем полном вакууминге этой таблицы. Все обычные XIDs, которые старше этого XID, гарантированно заменяются на FrozenXID в таблице. Аналогично столбец datfrozenxid таблицы pg_database базы данных является нижняя граница обычных XID, которые есть в этой базе данных — это просто минимальное из значений relfrozenxid таблицы в базе данных. Для получения этой информации выполните следующие запросы:

```
SELECT c.oid::regclass as table_name,
       greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as age
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm');
SELECT datname, age(datfrozenxid) FROM pg_database;
```

Столбец age является показателем числа транзакций между отсекающим XID и XID текущей транзакцией.

Обычно VACUUM сканирует только страницы, которые были модифицированы со времени последнего вакуумирования, но relfrozenxid может быть использован только при полном сканировании таблицы. Таблица полностью сканируется, когда relfrozenxid старше, чем vacuum_freeze_table_age транзакций при использовании команды VACUUM FREEZE, или, когда все страницы требуют проведения вакуумирования для удаления строк, которые более не используются. После завершения полного сканирования таблицы при выполнении VACUUM значение age(relfrozenxid) должно немного больше, чем использованное значение vacuum_freeze_min_age (больше на число транзакций, которые стартовали после запуска VACUUM). Если при выполнении VACUUM полное сканирование для таблицы не проводилось, и значение age(relfrozenxid) достигло autovacuum_freeze_max_age, то вскоре для данной таблицы будет принудительно выполнен автовакуумирование.

Если по какой-либо причине с помощью операции автовакуумирования не удалось удалить старые XID из таблицы, то по достижении самым старым XID БД значения десяти миллионов транзакций от начальной точки цикла, система начнет выдавать предупреждения подобные следующим:

```
WARNING:  database "mydb" must be vacuumed within 177009986 transactions
HINT:  To avoid a database shutdown, execute a database-wide VACUUM in "mydb".
```

В соответствии с данным предупреждением, проблема должна быть решена посредством запуска VACUUM вручную, но следует отметить, что команда VACUUM должна выполняться от имени суперпользователя. В противном случае произойдет сбой при обработке системных каталогов, и, следовательно не сможет измениться значение datfrozenxid для БД. При игнорировании предупреждений система откажется выполнять новые транзакции, когда до циклического переполнения счетчика останется менее одного миллиона транзакций:

```
ERROR:  database is not accepting commands to avoid wraparound
  data loss in database "mydb"
HINT:  Stop the postmaster and vacuum that database in single-user mode.
```

Наличие безопасного допуска в один миллион транзакций необходимо для предоставления администратору возможности восстановить работу без потери данных, выполнив

вручную необходимые команды VACUUM. Однако, поскольку система не будет выполнять команды после перехода в режим безопасного останова, то единственной возможность является остановка сервера и использование для выполнения VACUUM в однопользовательском режиме. Однопользовательский режим не будет вызвать переход в режим безопасного останова.

13.1.6. Демон autovacuum

В PostgreSQL существует опциональное, но настоятельно рекомендуемое для использования, средство, называемое автовакууминг, которое предназначено для автоматизации выполнения команд VACUUM и ANALYZE. При включении автовакууминга проверяются таблицы с большим количеством вставленных, обновленных или удаленных записей. При проверке используются средства сбора статистической информации, следовательно, автовакууминг не может быть использован, если не установлено значение true параметра track_counts. В конфигурации по умолчанию автовакууминг включен, и связанные с ним параметры настроены соответствующим образом.

Демон автовакууминга в действительности состоит из нескольких процессов. Существует постоянный процесс демона, называемый autovacuum launcher (запускатель автовакууминга) и отвечающий за запуск процессов autovacuum worker (рабочих процессов автовакууминга) для всех БД. Запускатель осуществляет распределение работы в времени, пытаясь запускать один рабочий процесс для каждой БД каждые autovacuum_naptime секунд. Для каждой БД запускается отдельный процесс, при этом максимальное число одновременно запущенных процессов составляет autovacuum_max_workers. В случае необходимости обработать более, чем autovacuum_max_workers БД, следующая БД будет обрабатываться через минимально возможный временной интервал по завершении первого рабочего процесса. Каждый рабочий процесс проверяет каждую таблицу в пределах обрабатываемой БД и при необходимости выполняет VACUUM и/или ANALYZE. Для мониторинга активности автовакууминга может быть использован log_autovacuum_min_duration.

Если за короткий временной интервал несколько больших таблиц оказываются имеющими право на проведение вакууминга, то все рабочие процессы автовакууминга могут оказаться занятыми выполнением вакууминга для данных таблиц в течение длительного периода времени. Это может привести к возникновению ситуации, когда для других таблиц и БД вакууминг проводиться не будет до освобождения какого-либо рабочего процесса. Ограничения на число рабочих процессов для одной БД не устанавливается, однако рабочие процессы пытаются избежать повторения работы, которая уже выполнена другими рабочими процессами. Следует отметить, что число запущенных рабочих процессов не зависит от ограничений, установленных значениями max_connections и superuser_reserved_connections.

Для таблиц, у которых значение relfrozenxid старше чем autovacuum_freeze_max_age транзакций, всегда выполняется вакууминг (это также применимо для таблиц, у которых максимальный возраст до замены идентификаторов был модифицирован с помощью параметров хранения. Кроме того, вакууминг проводится при превышении числом устаревших после предыдущего выполнения VACUUM записей порога вакуумминга. Порог вакуумминга определяется следующим образом:

```
порог вакуумминга = базовый порог вакуумминга +
  масштабирующий множитель вакуумминга * число записей
```

Базовый порог вакуумминга определяется значением параметра autovacuum_vacuum_threshold, масштабирующий множитель определяется значением параметра autovacuum_vacuum_scale_factor, а число записей значением pg_class.reltuples. Число устаревших записей может быть получено от сборщика статистической информации и является приблизительным значением, обновляемым при каждом выполнении операции UPDATE и DELETE. Число записей является приблизительным вследствие возможной потери некоторой информации при большой загрузке. Если значение relfrozenxid для таблицы старше чем vacuum_freeze_table_age транзакций, то проводится полное сканирование таблицы для замены идентификаторов старых записей и сдвига значения relfrozenxid, в противном случае сканируются только страницы модифицированные с момента последнего вакуумминга.

Для анализа используется схожее условие, при котором порог определяется следующим образом:

```
порог анализа = базовый порог анализа +
масштабирующий множитель анализа * число записей
```

Полученное значение сравнивается с общим числом записей, вставленных или обновленных с момента последнего выполнения ANALYZE.

Временные таблицы недоступны для автовакуумминга. Однако, операции вакуумминга и анализа могут быть выполнены в течении сессии комадами SQL.

По умолчанию пороги и масштабирующие множители берутся из postgresql.conf, но существует возможность их потабличного переопределения посредством изменения параметров хранения. Если значения изменены посредством параметров хранения, то используются эти измененные значения, в противном случае используются глобальные значения. Дополнительная информация приведена в 8.10.

Помимо значений базовых порогов и масштабирующих множителей существует шесть параметров, которые могут быть заданы для каждой таблицы посредством параметров хранения. Для первого параметра autovacuum_enabled может быть задано значение false для выдачи демону автовакуумминга указания пропустить таблицу. В

данном случае таблица будет затрагиваться автовакуумингом только с целью предотвращения циклического переполнения счетчика транзакций. Следующие два параметра, autovacuum_vacuum_cost_delay и autovacuum_vacuum_cost_limit, используются для задания специфичных для таблицы значений, определяющих задержку операции вакуумирования на основе оценки стоимости. Параметры autovacuum_freeze_min_age, autovacuum_freeze_max_age и autovacuum_freeze_table_age используются для настройки значений vacuum_freeze_min_age, autovacuum_freeze_max_age и vacuum_freeze_table_age, соответственно.

При выполнении нескольких рабочих процессов, предел стоимости сбалансирован между выполняющимися рабочими процессами таким образом, чтобы общее влияние на систему было одинаковым независимо от числа реально выполняющихся рабочих процессов. Тем не менее, обработка любых рабочих таблиц, у которых параметры autovacuum_vacuum_cost_delay или autovacuum_vacuum_cost_limit были установлены не рассматривается в алгоритме балансировки

13.2. Плановое переиндексирование

В определенных случаях полезно осуществлять периодическое перестроение индексов посредством выполнения команды REINDEX.

Страницы B-tree индекса, которые стали полностью пустыми, возвращаются для повторного использования. Однако все еще существует возможность неэффективного использования дискового пространства: если на странице было удалено всего лишь несколько ключей индекса, то страница остается распределенной. Таким образом, использование модели, в которой всего лишь несколько ключей в каждой последовательности со временем удаляется, приведет к неэффективному использованию дискового пространства. Для подобных моделей рекомендуется периодически проводить перестроение индексов.

Потенциальные возможности разрастания не B-tree индексов недостаточно описаны. Следовательно, необходимо контролировать физический размер дискового пространства при использовании индексов типа не B-tree.

Кроме того, скорость доступа для свежепостроенных индексов B-tree выше чем у индексов, обновлявшихся много раз, вследствие того, что страницы, расположенные рядом логически, для вновь перестроенных индексов обычно расположены рядом физически. Данное соображение в настоящее время неприменимо для индексов не B-tree. Периодически полезно перестраивать индексы просто для повышения производительности.

13.3. Обслуживание файлов журнала

Рекомендуется сохранять журналы протоколирования сервера СУБД на внешних носителях, а не перенаправлять вывод на устройство /dev/null. Содержимое журналов

протоколирования бесценно при диагностировании проблем. Однако журнал протоколирования имеет тенденцию к увеличению в объеме (особенно при использовании высоких уровней протоколирования отладочных сообщений) и не требует бесконечного хранения. Необходимо осуществлять ротацию файлов журналов протоколирования таким образом, чтобы по истечении допустимого периода времени начинали вестись новые файлы журнала протоколирования, а старые удалялись.

При простом перенаправлении стандартного потока ошибок в файл будет осуществляться регистрация протоколируемых сообщений, но в этом случае единственным способом для отсечения файла журнала протоколирования является остановка и перезапуск сервера. Подобный подход может быть приемлемым при использовании PostgreSQL в системах разработки, но не применим для большинства промышленных серверов.

Лучший подход заключается в перенаправлении стандартного потока ошибок сервера в программу, осуществляющую ротацию файлов журнала протоколирования некоторого типа. Существует встроенная программа ротации файлов журнала протоколирования, которая может быть использована посредством установки значения `true` конфигурационного параметра `logging_collerctor` в файлу `postgresql.conf`. Параметры настройки данной программы описаны в 8.8.1. Существует возможность для сбора протоколируемых данных в формате CSV.

Альтернативой является использование внешней программы ротации файлов журнала протоколирования, которая уже может использоваться с другим серверным программным обеспечением. Например, средство ротации файлов журнала протоколирования, включенное в состав дистрибутива WEB-сервера Apache, может быть использовано с PostgreSQL. Для использования внешней программы необходимо просто перенаправить стандартный поток ошибок сервера в соответствующую программу. Если сервер запущен с помощью программы `pg_ctl`, то стандартный поток ошибок уже перенаправлен в стандартный поток вывода данной программы, таким образом, необходимо использовать команду перенаправления, например следующего вида:

```
pg_ctl start | rotatelogs /var/log/pgslq_log 86400
```

Другой подход к управлению журналом протоколирования заключается в выводе протоколируемых данных в `syslog` и возложении функций по ротации файлов на `syslog`. Для реализации данного подхода необходимо установить значение `syslog` для конфигурационного параметра `log_destination` (вывод только в `syslog`) в файле `postgresql.conf`. В дальнейшем можно отправлять сигнал `SIGHUP` демону `syslog` в случае необходимости начать запись в новый файл. Для автоматизации ротации файлов, программа ротации может быть сконфигурирована для работы с файлами журнала `syslog`.

Однако в многих системах `syslog` не очень надежен, особенно при большом ко-

личестве протоколируемых сообщений, и может обрезать или отбрасывать сообщения, а требуется протоколирование всех сообщений без исключения. К тому же в системах семейства Linux `syslog` записывает каждое сообщение на диск синхронно, что приводит к снижению производительности. Существует возможность использовать символ «-» в начале имени файла в конфигурационном файле для `syslog` в целях отключения режима синхронной записи.

Следует отметить, что в описанных выше решениях рассмотрены вопросы обеспечения записи в новые файлы журнала протоколирования через настраиваемые временные интервалы, но не рассмотрены вопросы удаления файлов журнала протоколирования, которые более не нужны. Существует возможность настроить пакетное задание для периодического удаления старых файлов журнала протоколирования. Другой возможностью является настройка программы ротации для циклической перезаписи старых файлов журнала протоколирования.

14. РЕЗЕРВНОЕ КОПИРОВАНИЕ И ВОССТАНОВЛЕНИЕ

В PostgreSQL есть три фундаментально отличающихся подхода к резервному копированию данных:

- SQL-дамп;
- резервное копирование на уровне ФС;
- непрерывное архивирование.

Каждый метод имеет свои сильные и слабые стороны. Все эти методы описаны далее.

ВНИМАНИЕ! Для создания и восстановления дампа, базы данных, содержащей мандатные метки, см. 97 01-1 «Операционная система специального назначения «Astra Linux Special Edition». Руководство по КСЗ. Часть 1».

14.1. SQL-дамп

Идея этого метода заключается в создании файла с командами SQL, при исполнении которого на сервере можно воссоздать БД в том состоянии, в котором она находилась перед дампом. Для этой цели в PostgreSQL представлена утилита `pg_dump`. Пример использования этой команды:

```
pg_dump dbname > outfile
```

Как видно, результаты работы `pg_dump` выводится в стандартный поток вывода. Ниже будет показано, насколько это может быть полезным. Если команда, указанная выше, создает текстовый файл, `pg_dump` может создавать файлы различных форматов, которые поддерживают параллелизм и детальный контроль восстановления объектов.

Программа `pg_dump` является обычным клиентским приложением PostgreSQL. Это означает, что возможен запуск процедуры резервного копирования с любого удаленного компьютера, который имеет доступ к БД. Однако, следует помнить, что `pg_dump` не работает с частично настроенными правами. В частности, он должен обладать правами на чтение всех таблиц, для которых создаются резервные копии, поэтому как правило практически всегда следует запускать его от имени суперпользователя. (Если у вас нет достаточных привилегий для резервного копирования всей базы данных, вы можете создать резервную копию части базы данных, к которой у вас есть доступ, используя такие параметры, как `-n` для схем `-t` для таблиц.)

Для указания, к какому серверу должен подключится `pg_dump`, используются опции командной строки `-h host` и `-p port`. В качестве узла по умолчанию используется локальный компьютер или то, что указано в переменной окружения `PGHOST`. Аналогично, порт по умолчанию определяется переменной окружения `PGPORT` или, если это не удалось, значением по умолчанию, указанным при компилировании. (Для удобства обычно у сервера

при компилировании указывается такое же значение по умолчанию.)

Как и любое другое клиентское приложение PostgreSQL, pg_dump по умолчанию устанавливает соединение с именем пользователя БД, соответствующим имени текущего пользователя ОС. Это можно изменить с помощью опции -U или заданием переменной окружения PGUSER. Следует также помнить, что соединения pg_dump подчиняются обычным механизмам аутентификации клиентов, которые описаны в 9.

Важным преимуществом pg_dump перед другими способами резервирования, является то, что результат выполнения pg_dump может быть естественным образом загружен в новые версии PostgreSQL, тогда как резервное копирование на уровне ФС или непрерывное архивирование сильно зависят от версии сервера. Помимо этого pg_dump служит единственным средством для переноса баз данных между различными архитектурами, например с 32-х разрядного сервера на 64-х разрядный.

Дампы, создаваемые с помощью pg_dump, обладают внутренней целостностью, т.к. дамп представляет собой «снимок» БД в состоянии на момент запуска pg_dump. Во время своей работы pg_dump не блокирует выполнение других операций. (Исключения составляют операции, требующие полной блокировки, среди которых почти все виды ALTER TABLE.)

14.1.1. Восстановление дампа

Текстовые файлы, создаваемые pg_dump, предназначены для исполнения программой psql. Восстановление дампа производится командой, общий вид которой представлен ниже:

```
psql dbname < infile
```

В данной команде infile является именем файла, которое было использовано в качестве outfile в команде pg_dump. При запуске этой команды не создается БД dbname, в связи с чем перед запуском psql требуется ее создание заранее из template0 (например, createdb -T template0 dbname). Утилита psql поддерживает опции командной строки подобно pg_dump, указывающие сервер БД, с которым следует устанавливать соединение, и имя пользователя, под которым соединение будет установлено. Более подробная информация содержится на справочной странице psql. Дампы, созданные в бинарном формате, могут быть восстановлены с помощью утилиты pg_restore (см. 18.17).

Перед восстановлением SQL-дампа необходимо, чтобы существовали все учетные записи пользователей, являющихся владельцами объектов или имеющих права на объекты восстанавливаемой БД. Если их не будет, процесс восстановления не сможет восстановить объекты с первоначальной принадлежностью и/или правами на них.

По умолчанию скрипт psql продолжает выполнение при обнаружении SQL-ошибки. При необъходимости это можно изменить, установив при запуске psql переменную ON_ERROR_STOP. После этого, при возникновении SQL-ошибки, приложение psql будет

прерывать работу и выходить со статусом 3:

```
psql --set ON_ERROR_STOP=on dbname < infile
```

В любом случае, в результате будет лишь частично восстановленная БД. С другой стороны, можете быть указано, чтобы восстановление всего дампа было проведено в одной транзакции, т.е. восстановление либо будет проведено полностью, либо не будет проведено вовсе. Этот режим может быть задан опциями `-1` или `-single-transaction` командной строки `psql`. Следует учитывать, что в таком случае малейшая ошибка может привести к отмене восстановления, продолжавшегося несколько часов. Однако, даже это может быть более предпочтительно чем ручная чистка сложной БД после частично восстановленного дампа.

Способность утилит `pg_dump` и `psql` записывать или читать из каналов делает возможным выполнять дамп БД непосредственно с одного сервера на другой. Например:

```
pg_dump -h host1 dbname | psql -h host2 dbname
```

Примечание. Дампы, создаваемые с помощью `pg_dump`, выполняются относительно `template0`. Это означает, что любые языки, процедуры и т.п., добавляемые через `template1`, также будут добавлены в дамп. В результате, при использовании модифицированной `template1`, при восстановлении необходимо создать пустую БД из `template0`, как показано в примере выше.

После восстановления резервной копии разумным шагом будет запуск `ANALYZE` для каждой БД, чтобы оптимизатор запросов получил нужную статистику. Проще всего это может быть сделано выполнением `vacuumdb -a -z`: что эквивалентно запуску `VACUUM ANALYZE` для каждой БД вручную.

14.1.2. Использование pg_dumpall

Утилита `pg_dump` создает за раз дамп только одной БД, при этом информация о ролях или табличных пространствах не сохраняется (т.к. эта информация относится ко всему кластеру, а не к каждой отдельной БД). Для обеспечения удобного сохранения дампа всего содержимого кластера предназначена программа `pg_dumpall`, которая создает резервную копию каждой БД кластера, а также сохраняет информацию о кластере, такую как определения ролей и табличных пространств. Основное использование команды осуществляется следующим образом:

```
pg_dumpall > outfile
```

Полученный дамп может быть восстановлен с помощью `psql`:

```
psql -f infile postgres
```

(В действительности, возможно указание в качестве стартовой БД любое имя, но при загрузке данных в пустой кластер, как правило, требуется указание postgres.) При восстановлении дампа, полученного с помощью `pg_dumpall`, необходимо обладать правами

суперпользователя БД, поскольку они требуются для восстановления информации о ролях и табличных пространствах. При использовании табличных пространств следует убедиться, что пути табличных пространств из дампа подходили для новой конфигурации.

Утилита pg_dumpall сначала выполняет команды для создания ролей,табличных пространств и пустых БД, и лишь затем запускает pg_dump для каждой БД. Это означает, что, хотя каждая БД будет обладать внутренней целостностью, «снимки» различных БД могут не быть полностью синхронизированы.

Данные кластера могут быть зарезервированы при использовании pg_dumpall с ключом --globals-only. Это необходимо для полного резервирования кластера, если запускается утилита pg_dump для каждой базы данных.

14.1.3. Поддержка больших БД

Поскольку PostgreSQL разрешает создавать таблицы большего размера, чем максимальный размер файла в системе, может оказаться проблематичным создание файла для дампа такой таблицы, поскольку размер этого файла может оказаться больше максимально допустимого в системе. Так как результаты работы pg_dump могут быть выведены в стандартный поток вывода, для решения этой вероятной проблемы существует возможность использования стандартных средств Unix. Существует несколько способов:

1) Использование сжатых дампов.

Существует возможность применения желаемой программы сжатия, например gzip:

```
pg_dump dbname | gzip > filename.gz
```

Загрузка выполняется следующим образом:

```
gunzip -c filename.gz | psql dbname
```

или:

```
cat filename.gz | gunzip | psql dbname
```

2) Использование split.

Команда split позволяет разбивать выходной файл на части допустимого для ФС размера. Например, для создания частей размером в 1 МБ:

```
pg_dump dbname | split -b 1m - filename
```

Загрузка выполняется следующим образом:

```
cat filename* | psql dbname
```

3) Использование особого формата дампа pg_dump.

PostgreSQL собран с установленной библиотекой сжатия zlib и особый формат дампа будет сжимать данные по мере их записи в результирующий файл. Таким образом, размер файла дампа будет сравним с размером, получаемым при использовании gzip, но при этом добавляется возможность выборочного восстановления таблиц. Следующая команда создает дамп БД с использованием особого формата

дампа:

```
pg_dump -Fc dbname > filename
```

Особый формат дампа не является скриптом psql, и вместо этого должен быть восстановлен с помощью утилиты pg_restore, например:

```
pg_restore -d dbname filename
```

Дополнительная информация приведена в страницах справки для программ pg_dump и pg_restore.

Для очень больших БД, возможно потребуется сочетание split с одним из двух других методов.

4) Использование режима параллельной работы pg_dump.

Для ускорения процесса получения дампа большой БД, существует возможность работы pg_dump в параллельном режиме. При этом дампы нескольких таблиц выполняются одновременно. Управление степенью параллелизма осуществляется с помощью параметра -j. Параллельный дамп поддерживается только для формата архива «directory».

```
pg_dump -j num -F d -f out.dir dbname
```

Для выполнения параллельного восстановления используется pg_restore -j. Этот способ возможен для форматов архива «directory» и «custom», даже если они не создавались с помощью pg_dump -j.

14.2. Резервное копирование на уровне ФС

Другой подход к резервному копированию заключается в копировании файлов, которые PostgreSQL использует для хранения данных в БД. Местонахождение этих файлов описано в 7.2. Возможно применение любого метода, который обычно используется для резервного копирования в ФС, например:

```
tar -cf backup.tar /usr/local/pgsql/data
```

Однако, есть два ограничения, которые делают этот метод непрактичным или, по крайней мере, уступающим pg_dump:

1) Для получения годной к употреблению копии, сервер БД должен быть остановлен. Такие полумеры, как отключение всех соединений, не сработают (отчасти потому, что tar и подобные ему инструменты не делают полный снимок состояния ФС, а также из-за внутренней буферизации на сервере). Информация об остановке сервера приведена в 7.5. Нет необходимости говорить, что и перед восстановлением данных сервер должен быть остановлен.

2) Обладая информацией о расположении БД в ФС, нельзя попытаться сохранить или восстановить лишь некоторые отдельные таблицы или БД из соответствующих файлов и каталогов. Это не сработает, поскольку в этих файлах содержится лишь

часть необходимой информации. Остальная часть находится в файлах журнала записи выполнения транзакций `pg_clog/*`, который содержит статус выполнения всех транзакций. Файл с таблицей может быть использован только вместе с этой информацией. Также невозможно восстановить одну только таблицу и соответствующие ей данные `pg_clog`, т.к. это нарушит все остальные таблицы кластера БД.

Таким образом, резервное копирование на уровне ФС работает только в случае полного копирования и восстановления всего кластера БД.

Другой метод резервного копирования на уровне ФС заключается в создании «согласованного снимка» каталога данных, при условии, что ФС поддерживает эту функциональность (и при уверенности, что она корректно реализована). Типичная процедура состоит в создании «замороженного снимка» тома, на котором находится БД, а затем копируется весь каталог данных (а не отдельные части) с этого снимка на устройство для сохранения резервной копии, после чего замороженный снимок освобождается. Этот метод работает из без остановки сервера БД. Однако, при таком копировании файлы БД оказываются в состоянии, как если бы сервер БД не был остановлен должным образом. Поэтому, при запуске сервера с сохраненными данными, он будет вести себя будто работа предыдущей версии сервера была некорректно прервана, и будет выполнен журнал WAL. Это не является проблемой, однако об этом следует помнить (и включать в резервную копию файлы WAL-журнала).

Если БД распределена среди нескольких ФС, может не оказаться возможности получить одновременные замороженные снимки всех томов. Например, если файлы данных и журнал WAL находятся на разных дисках, или табличные пространства находятся в разных ФС, может не получиться сделать копирование со снимков, потому что снимки должны быть одновременными. Требуется внимательное изучение документации по ФС, прежде чем доверять методу согласованных снимков в подобных ситуациях.

Если не существует возможности получения одновременных снимков, остается вариант остановки сервера БД на время, достаточное для того, чтобы создать все замороженные снимки. Другим вариантом является использование подхода резервного копирования с непрерывным архивированием (см. 14.3.2), поскольку подобное создание резервной копии не зависит от изменений, происходящих в системе во время копирования. Для этого требуется включение режима непрерывного архивирования только на время процесса резервного копирования. Восстановление производится с помощью непрерывного восстановления из архива (см. 14.3.3).

Другим вариантом является использование `rsync` для резервного копирования на уровне ФС. Это осуществляется путем запуска в первый раз `rsync` при работающем сервере, а затем остановкой сервера БД на время второго запуска `rsync`. Второй раз выполнение `rsync` потребует гораздо меньше времени, т.к. потребуется передать относительно неболь-

шой объем данных, а результат будет целостным, поскольку сервер был остановлен. Этот подход позволяет минимизировать время простоя сервера при создании резервной копии на уровне ФС.

Заметим, что резервная копия в ФС не обязательно будет иметь меньший объем, чем SQL-дамп. Наоборот, скорее всего, она займет больше места. (утилите `pg_dump`, например не требуется включать в дамп содержимое индексов, достаточно только команд для их воссоздания.) Однако, резервное копирование на уровне ФС может оказаться быстрее.

14.3. Непрерывное архивирование и восстановление до состояния на определенный момент времени (PITR — Point-In-Time Recovery)

В любой момент времени PostgreSQL ведет журнал опережающей записи (WAL — write ahead log) в подкаталоге `pg_xlog` каталога данных кластера. В журнале описываются все изменения, применяемые к файлам данных БД. Он необходим в первую очередь для защиты от сбоев: при возникновении сбоя системы, БД будет восстановлена с помощью «проигрывания» всех записей журнала, сделанных с момента последней контрольной точки. Кроме того, существование такого журнала делает возможным третий сценарий резервного копирования БД: сочетание резервного копирования на уровне ФС с резервным копированием файлов WAL. При необходимости восстановления БД, восстанавливается резервная копия, а затем проигрываются записи из сохраненных файлов WAL, для доведения восстановленной БД до актуального состояния. Применение этого подхода сложнее, чем предыдущих, но у него есть несколько значительных преимуществ:

1) На начальный момент времени не требуется идеально целостная резервная копия. Все внутренние несоответствия в резервной копии будут исправлены при проигрывании журнала (что не сильно отличается от восстановления после системного сбоя). Таким образом, не требуется наличия возможностей по созданию снимка ФС, достаточно `tar` или подобного инструмента для архивирования.

2) Так как нет ограничения на длину последовательности файлов WAL для последующего проигрывания, непрерывное резервное копирование может быть достигнуто просто непрерывным архивированием файлов WAL. Это особенно важно для больших БД, когда отсутствует возможность часто делать полное резервное копирование.

3) Не является необходимым обязательное проигрывание содержимого WAL до самого конца. Существует возможность остановить проигрывание в любой точке, и получить согласованный снимок БД, какой она была в тот момент времени. Таким образом, этот метод позволяет делать восстановление к состоянию на определенный момент времени: возможно восстановить БД до состояния на любой момент времени, начиная с момента создания резервной копии.

4) В случае постоянной передачи порции файлов WAL на другую систему, на которой была загружена та же резервная копия БД, возможно получение системы горячего резерва: в любой момент может быть подключена вторая система, при этом на ней будет ближайшая к текущей версия БД.

Примечание. `pg_dump` и `pg_dumpall` не создают резервные копии на уровне ФС и не могут быть использованы как часть решения по непрерывному архивированию. Подобные средства работают на логическом уровне и не содержат информации, необходимой для проигрывания журнала WAL.

Как и в случае резервного копирования на уровне ФС, этим методом можно восстановить только весь кластер БД, но не его подразделы. Кроме того, понадобится много места для хранения архивов: резервная копия БД может быть очень объемной, и работающая система может генерировать мегабайты информации для WAL, которую требуется архивировать. Тем не менее, во многих ситуациях с высокими требованиями к надежности этот метод является наиболее предпочтительным.

Для успешного восстановления при непрерывном архивировании (также зачастую называемом «резервным копированием на лету»), нужна непрерывная последовательность архивных файлов WAL, которая простирается как минимум до точки начала резервного копирования. Таким образом, требуется настроить и проверить процедуру архивирования файлов WAL перед тем, как сделать первую резервную копию БД. В соответствии с этим, в первую очередь рассматривается архивирование файлов WAL.

14.3.1. Настройка архивирования журнала WAL

В общем случае, работающая система PostgreSQL производит неограниченную по длине последовательность записей в WAL. Система физически делит эту последовательность на сегментные файлы WAL, которые обычно имеют размер 16 МБ (хотя размер сегмента может быть изменен при сборке PostgreSQL). Сегментные файлы получают численные имена, которые отражают их позицию в последовательности записей WAL. Если архивирование WAL не используется, система, как правило, создает несколько сегментных файлов, а затем очищает их, переопределяя ненужным сегментным файлам более высокие сегментные номера. При этом подразумевается, что сегментный файл, содержимое которого предваряет предпоследнюю контрольную точку, больше не нужен, и его можно очистить.

При архивировании данных WAL необходимо брать содержимое каждого сегментного файла по мере их заполнения и где-то сохранять эти данные перед тем, как очистить сегментный файл для повторного использования. В зависимости от приложения и доступного оборудования, сохранять эти данные можно несколькими способами: копированием сегментных файлов на подсоединенный через NFS каталог на другой системе, записью их на магнитную ленту (при условии, что существует способ идентификации первоначального

имени каждого файла) или сбором их вместе и записью на лазерный диск, или куда-либо еще полностью. Для того, чтобы предоставить администратору максимальную свободу выбора, PostgreSQL не делает никаких предположений о том, как именно производится архивирование. Напротив, PostgreSQL позволяет администратору определить, какую консольную команду следует использовать для копирования заполненного сегмента туда, куда он должен быть скопирован. Команда может быть простой, например, `cp`, или может вызывать в консоли сложный скрипт.

Включение архивирования WAL производится установкой конфигурационного параметра `wal_level` в значение `archive` или выше, параметра `archive_mode` в значение `on` и указанием используемой консольной команды в конфигурационном параметре `archive_command`. Как правило, эти настройки должны быть в файле `postgresql.conf`. В значении параметра `archive_command` комбинация `%p` заменяется на путь к файлу, который нужно архивировать, тогда как вместо `%f` записывается только имя файла. (Путь записывается относительно текущего рабочего каталога, т.е. каталога данных кластера.) При необходимости вставить в команду символ `%`, следует писать `%%`. Простейшая команда выглядит следующим образом:

```
archive_command = 'test ! -f /mnt/server/archivedir/%f \
                  && cp %p /mnt/server/archivedir/%f'
```

При этом архивируемые сегменты WAL будут скопированы в каталог `/mnt/server/archivedir`. (Это не рекомендация, а всего лишь пример, поэтому может не работать на некоторых платформах.) После заменены `%p` и `%f`, выполняемая команда может выглядеть так:

```
test ! -f /mnt/server/archivedir/00000001000000A900000065 \
 && cp pg_xlog/00000001000000A900000065
```

Подобная команда будет сформирована для каждого нового файла, который будет архивирован.

Архивирование будет выполняться с правами того же пользователя, от имени которого работает сервер PostgreSQL. Так как серия файлов WAL, которая будет архивирована, содержит практически всю информацию о БД, необходимо удостовериться, что архивированные данные защищены от посторонних, например, сохранены в каталоге, на который нет права чтения для группы или для всех.

Важно, чтобы команда архивирования возвращала нулевой код возврата только при успешном выполнении. Получив нулевой результат, PostgreSQL будет считать, что файл был успешно архивирован, и удалит или очистит его. В то же время, ненулевой результат указывает PostgreSQL, что файл не был архивирован, и сервер будет периодически повторять попытки.

Команда архивирования должна быть задана так, чтобы запрещать запись поверх любого из существующих файлов архива. Это важное условие для сохранения целостности вашего архива на случай ошибки администратора (например, если данные с двух различных серверов будут направлены в один и тот же архивный каталог).

Рекомендуется проверить команду архивирования и убедиться, что она не будет перезаписывать уже существующий файл, и что в этом случае она возвратит ненулевой код возврата. Приведенный ранее пример содержит для этого отдельное действие `test`. На некоторых Unix-платформах команда `cp` имеет ключ `-i`. который может быть использован для этого менее наглядно, но без проверки реального кода возврата не следует полагаться на такое решение. (Обычно, GNU `cp` возвращает нулевой код возврата при использовании ключа `-i` при существовании файла, что не является желаемым поведением.)

При создании настроек архивирования, необходимо рассмотреть, что произойдет, если архивирование будет постоянно выдавать ошибку, поскольку некоторые аспекты могут требовать вмешательства оператора или не будет места для записи. Например, это может произойти, при записи на магнитную ленту без автозамены. Как только лента будет заполнена, ничего больше нельзя будет архивировать, пока лента не будет заменена. Следует убедиться, что в любых условиях возникновения ошибки или если нужна реакция оператора, выдается сообщение, позволяющее максимально быстро решить проблему. Пока ситуация не будет разрешена, сегментные файлы WAL будут складываться в каталог `pg_xlog/`. (Если ФС, содержащая `pg_xlog/`, будет переполнена, сервер PostgreSQL остановиться со статусом `PANIC`. Предыдущие транзакции не будут потеряны, но БД будет недоступна, пока в ФС не появится свободное место.)

Скорость архивирования не важна до тех пор, пока она соответствует средней скорости, с которой сервер генерирует данные WAL. Обычная работа продолжается, даже если архивирование немного отстает. Если архивирование сильно отстанет, это увеличит объем данных, которые могут быть потеряны в случае отказа. Это также будет означать, что в каталоге `pg_xlog/` окажется слишком много сегментных файлов, ожидающих архивирования, т.е. она может занять все доступное дисковое пространство. Рекомендуется следить за процессом архивирования, чтобы знать, что все работает так, как было задумано.

При написании команды архивирования, следует иметь ввиду, что имена файлов, которые требуется архивировать, могут иметь длину до 64 символов и могут содержать любую комбинацию ASCII букв, чисел и точек. Нет необходимости запоминать относительный путь (`%p`), но требуется помнить имя файла (`%f`).

Следует заметить, что, хотя архивирование WAL позволяет восстановить любые изменения, произведенные над БД PostgreSQL, изменения, внесенные в конфигурационные файлы (т.е., `postgresql.conf`, `pg_hba.conf` и `pg_ident.conf`), восстановлены не будут,

поскольку они редактируются не с помощью операторов SQL, а вручную. Рекомендуется держать конфигурационные файлы в таком месте, где будет совершаться их резервная копия при регулярном резервном копировании на уровне ФС. В 8.2 описано, как перемещать конфигурационные файлы.

Команда архивирования вызывается только для заполненных сегментов WAL. Таким образом, если сервер генерирует незначительное количество информации для WAL (или это происходит только в определенные периоды), между проведением транзакции и ее безопасным сохранением в архиве может пройти довольно много времени. Для указания максимального возраста данных до архивации может быть использован конфигурационный параметр `archive_timeout`, указывающий серверу переключаться на новый сегмент WAL не реже указанного времени. Необходимо отметить, что архивные файлы, завершенные раньше принудительным переключением, имеют тот же размер, что и полные. В связи этим не рекомендуется устанавливать слишком короткое значение периода в `archive_timeout` во избежание раздувания архивного хранилища. Обычно используемое значение `archive_timeout` составляет порядка минут.

Хорошей идеей является исключение из резервной копии дампа файлов из каталога `pg_replslot` кластера, чтобы слоты репликации, которые существуют на ведущем сервере не стали частью резервной копии. В противном случае, последующее использование резервного копирования для создания резервной может привести к бесконечному задержки WAL файлов на ведомом сервере, и, возможно, раздуваться на ведущем сервере, если режим горячего резерва о включен, потому что клиенты, использующие эти слоты репликации по-прежнему будут подключаться для обновления слотов на ведущем сервере, а не на ведомом. Даже если резервное копирование предназначено только для использования при создании нового ведущего сервера, копирование слотов репликации не ожидается, так как содержание этих слотов скорее всего будет некорректно до тех пор, пока новый ведущий сервер не будет запущен.

Также существует возможность принудительного переключения на следующий сегмент вручную с помощью `pg_switch_xlog`, в случае необходимости обеспечить сохранение только что завершенной транзакции как можно скорее. Остальные функции, относящиеся к управлению WAL, приведены в таблице 91.

При установленном значении `minimal` конфигурационного параметра `wal_level`, некоторые SQL команды оптимизированы для исключения фиксации в журнале WAL. При включение режима архивирования или потоковой репликации в процессе выполнения одной из таких команд, журнал WAL не будет содержать достаточно информации для архивного восстановления. (Восстановление после сбоя от этого не пострадает). По этой причине, конфигурационный параметр `wal_level` может быть изменен только при запуске сервера.

В то же время, конфигурационный параметр archive_mode может быть изменен и при перезагрузке конфигурационного файла. В случае необходимости временной приостановки архивирования, одним из способов является установка в качестве значения конфигурационного параметра archive_command пустой строки (''). Это приводит к накоплению WAL-файлов в каталоге pg_xlog/ до восстановления корректного значения параметра archive_command.

14.3.2. Создание базовой резервной копии

Наиболее простым способом создания базовой резервной копии выступает использование утилиты pg_basebackup. Она может создавать базовую резервную копию в виде обычного файла или в виде архива tar. Если требуется большая гибкость, базовая резервная копия может быть создана с помощью прикладного программного интерфейса низкого уровня (Low Level API) (см. 14.3.2.1).

Нет необходимости беспокоится о том, сколько может занять времени создание базовой резервной копии. Однако, при запуске сервера с выключенным параметром full_page_writes, может быть замечена потеря производительности во время ее создания, т.к. в режиме создания резервной копии параметр full_page_writes включается принудительно.

Для использования резервной копии необходимо иметь практически все сегменты WAL, созданные во время и после резервного копирования на уровне ФС. Процесс создания базовой резервной копии создает файл истории резервной копии, который сохраняется в области архивирования WAL. Файл получает имя первого сегмента WAL, необходимого для использования резервной копии. Например, если первый необходимый файл WAL имеет имя 000000010000123400000055CD, файл истории резервной копии получит имя 000000010000123400000055CD.007C9330.backup. (Вторая часть имени файла содержит точную позицию в сегменте WAL, и обычно может быть проигнорирована.) После надежного архивирования резервной копии на уровне ФС и сегментов WAL, использованных в процессе создания резервной копии (определенных файлом истории резервной копии), все сегменты WAL, с численно меньшими номерами, становятся ненужными для восстановления резервной копии и могут быть удалены. Однако может быть рассмотрено хранение нескольких наборов резервных копии для полной уверенности в восстановлении данных.

Файл истории резервной копии представляет собой просто небольшой текстовый файл. Он содержит строку метки, передаваемую pg_basebackup, а также времена начала и завершения и список сегментов WAL резервной копии. Если метка идентифицировала место расположения ассоциированного файла резервной копии, файла истории резервной копии достаточно для определения этого файла при восстановлении.

Поскольку необходимо хранить все архивированные WAL файлы с момента создания

последней резервной копии, интервал между базовыми резервными копиями обычно должен быть выбран исходя из предполагаемого объема хранения для архивированных файлов WAL. Также следует учитывать время, затрачиваемое на восстановление. Если восстановление станет необходимым, система должна будет проиграть все WAL сегменты, а это может занять некоторое время, если последняя резервная копия была сделана достаточно давно.

14.3.2.1. Создание базовой резервной копии с помощью Low Level API

Процедура создания базовой резервной копии с помощью прикладного программного интерфейса низкого уровня содержит немного больше шагов, чем при использовании утилиты `pg_basebackup`, но при этом относительно проста. Важно следить, чтобы шаги выполнялись в строгой последовательности с проверкой успешности завершения каждого шага.

1) Необходимо убедиться, что архивирование журнала WAL включено и работает.
2) Установить соединение с БД от имени суперпользователя и выполнить команду:
```
SELECT pg_start_backup('label');
```
где `label` любая строка, уникально идентифицирующая операцию создания резервной копии. (Хорошей практикой является использование полного пути предположительного размещения дамп-файла.) `pg_start_backup` создает файл резервной копии `label`, называемый `backup_label`, в каталоге данных кластера с информацией о создаваемой резервной копии, включая время начала операции и строки идентификации. Данный файл критичен для целостности резервной копии, поскольку используется при восстановлении из нее.

Не имеет значения к какой из БД кластера установлено соединение при выполнении этой команды. Результат функции может быть игнорирован, но при сообщении об ошибке необходимо устранить ее перед продолжением действия.

По умолчанию выполнение `pg_start_backup` может занять значительное время. Это связано с тем, что в процессе создается контрольная точка и операции ввода/вывода, необходимые для создания контрольной точки, могут растягиваться на значительный период времени, по умолчанию на половину интервала создания контрольных точек (см. конфигурационный параметр `checkpoint_completion_target`). Обычно это соответствует желаемому, т.к. минимизирует влияние на выполнение запросов. При необходимости вызвать создание резервной копии как можно раньше:
```
SELECT pg_start_backup('label', true);
```
Это вызывает создание контрольной точки насколько возможно быстро.

3) Выполнить резервное копирование, используя любую подходящую утилиту создания резервной копии на уровне ФС, такой как `tar` или `cpio` (не `pg_dump` или `pg_dumpall`). Также не является необходимым или желательным остановка нормального выполнения операций с БД в процессе этого.

4) Снова установить соединение с БД от имени суперпользователя и выполнить команду:

```
SELECT pg_stop_backup();
```

При этом режим создания резервной копии прерывается и осуществляется автоматическое переключение на следующий сегмент журнала WAL. Необходимость такого переключения связана с подготовкой последнего записанного во время интервала создания резервной копии сегмента WAL к немедленному архивированию.

5) Как только файлы сегментов WAL, использованные во время интервала создания резервной копии, будут архивированы, то процесс создания резервной копии считается завершенным. Файл, идентифицируемый по результату выполнения `pg_stop_backup`, является последним сегментом журнала, требуемый для создания полного набора файлов резервной копии. При включенном `archive_mode` `pg_stop_backup` не возвращает результат до тех пор, пока последний сегмент не будет архивирован. Архивирование этих файлов происходит автоматически в соответствии с последним установленным значением конфигурационного параметра `archive_command`. В большинстве случаев это происходит быстро, но рекомендуется отслеживать архивирующую систему на предмет отсутствия задержек. В случае прерывания процесса архивирования из-за ошибки архивирующей команды, попытки архивирования будут продолжаться до достижения успеха и завершения создания резервной копии. При желании ограничить время выполнения команды `pg_stop_backup` следует установить соответствующее значение конфигурационному параметру `statement_timeout`.

Некоторые инструменты создания резервных копии выдают предупреждения или ошибки в случаях, когда копируемые ими файлы изменяются во время копирования. Подобная ситуация является нормальной, а не ошибочной, при создании резервной копии работающей БД; так что необходимо убедиться в возможности отличать подобные сообщения от реальных ошибок. Например, некоторые версии `rsync` возвращают отдельный код ошибки для «измененных исходных файлов», при этом возможно создание обрабатывающего скрипта для трактования этого кода возврата как не ошибочного. Также некоторые версии GNU `tar` возвращают код ошибки, неотличимый от фатальной ошибки в случае, когда файл усекается во время копирования его `tar`. GNU `tar` версий 1.16 и старше возвращают `1`, если файл был изменен во время создания резервной копии, и `2` для других ошибок. С GNU `tar` версий 1.23 и старше возможно использование опций `--warning=no-file-changed --warning=no-file-removed` для скрытия соответствующих предупреждений.

Необходимо следить за тем, чтобы резервная копия содержала все файлы, распо-

ложенные в каталоге кластера (например, /usr/local/pgsql/data). При использовании табличных пространств, не находящихся ниже указанного каталога, не следует забывать также включать их в резервную копию (при этом необходимо быть уверенным в том, что в резервной копии символические ссылки сохраняются как символические ссылки, иначе табличные пространства не будут восстановлены должным образом).

В то же время допустимо не включать в резервную копию файлы из подкаталога pg_xlog/ каталога данных кластера. Это незначительное усложнение имеет смысл, т.к. уменьшает риск ошибок при восстановлении, и может быть реализовано организацией pg_xlog/ в виде символической ссылки, указывающей за пределы каталога кластера, что и так является обычным по причинам улучшения производительности. Могут быть исключены и файлы postmaster.pid и postmaster.opts, содержащие информацию о запущенном процессе postmaster и не относящиеся непосредственно к резервной копии. (Указанные файлы могут нарушить работу pg_ctl.)

Стоит отметить, что функция pg_start_backup создает файл, который впоследствии удаляется при вызове pg_stop_backup, с именем backup_label в каталоге данных кластера. Этот файл также будет архивирован, как часть файла резервной копии. Файл backup_label содержит строку метки, заданной при вызове pg_start_backup, а также время запуска pg_start_backup и имя первого файла WAL. Поэтому при необходимости существует возможность найти в файле резервной копии эту информацию, и определить из какой сессии резервного копирования он был создан. Данный файл критичен для целостности резервной копии, поскольку используется при восстановлении из нее.

Также возможно создание дампа резервной копии и при остановленном сервере. В этом случае, естественно, нельзя воспользоваться функциями pg_start_backup и pg_stop_backup, поэтому требуется самостоятельно отслеживать на устройствах хранения файлы резервных копий и на какое время назад для них должны быть сохранены файлы WAL. В общем случае рекомендуется следовать описанной выше процедуре непрерывного резервного копирования.

14.3.3. Восстановление с использованием непрерывного резервного копирования

Рассмотрим процедуру восстановления из резервной копии в случае возникновения сбоя:

1) Необходимо остановить сервер, если он запущен.

2) При наличии достаточного места на диске скопировать весь каталог данных кластера и все табличные пространства во временное место хранения на случай, если они понадобятся впоследствии. Необходимо отметить, что эти меры предосторожности потребуют наличия места в системе для хранения двух копий существующей

БД. В случае отсутствия необходимого дискового пространства, необходимо как минимум скопировать содержимое подкаталога `pg_xlog` каталога данных кластера, т.к. он может содержать журналы WAL, которые не были архивированы до момента сбоя системы.

3) Удалить все файлы и подкаталоги в каталоге данных кластера и корневых каталогах всех используемых табличных пространств.

4) Восстановить файлы БД из резервной копии. Важно следить, чтобы они были восстановлены с правильными правами владения (системного пользователя БД, а не привилегированного пользователя `root`) и с верными правами доступа. При использовании табличных пространств, необходимо убедиться, что были корректно восстановлены символические ссылки в подкаталоге `pg_tblspc/`.

5) Удалить все файлы из `pg_xlog/`; т.к. они извлечены из резервной копии и потому, вероятно, содержат устаревшую информацию. Если не осуществлялось архивирование каталога `pg_xlog/`, необходимо его создать, следя за тем, чтобы он был восстановлен в виде символической ссылки, если ранее была такая конфигурация.

6) При наличии не архивированных файлов сегментов WAL, сохраненных на втором шаге, требуется скопировать их в `pg_xlog/`. (Следует их копировать, а не перемещать, так что в случае возникновения проблем, они останутся не измененными, что позволит начать операцию заново.)

7) Создать командный файл восстановления `recovery.conf` в каталоге данных кластера (см. 16). Возможно также потребуется временная модификация `pg_hba.conf` для предотвращения подключения обычных пользователей до момента восстановления работоспособности.

8) Запустить сервер. Сервер переходит в режим восстановления и при необходимости осуществляет чтение архивированных файлов WAL. Если процесс восстановления был прерван внешней ошибкой, сервер может быть просто перезапущен и продолжит восстановление. После завершения процесса восстановления сервер переименовывает файл `recovery.conf` в `recovery.done` (для предотвращения случайного перехода в режим восстановление в случае сбоя) и переходит в режим нормального функционирования.

9) Необходимо проверить содержимое БД для уверенности в том, что восстановление прошло успешно. Если это не так, производится возврат к первому шагу. В случае успешного восстановления необходимо восстановить доступ обычных пользователей путем модификации файла `pg_hba.conf`.

Ключевой частью является настройка командного файла восстановления, который описывает, как, и как долго должно производиться восстановление. В качестве про-

тотипа можно использовать файл recovery.conf.sample (обычно он устанавливается в каталог установки share/). Единственное, что абсолютно необходимо определить в recovery.conf, является команда восстановления restore_command, указывающая ка должны быть извлечены из архива файлы сегментов WAL. Как и archive_command — это командная строка для командной оболочки. Она может содержать параметр %f, замещаемый именем файла журнала, и %p, замещаемый путем, в который осуществляется копирование файла журнала. (Путь копирования указывается относительно текущего рабочего каталога, т.е. каталога данных кластера.) При необходимости вставить в команду символ %, следует писать %%. Простейшая команда выглядит следующим образом:

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
```

Команда копирует предварительно архивированный сегмент WAL из каталога /mnt/server/archivedir. В работе конечно используются более сложные команды, вплоть до скриптов, требующих оператора смонтировать соответствующую ленту.

Важно, чтобы команда возвращала ненулевой код возврата при сбое. У команды будет запрошен несуществующий в архиве файл, в этом случае она должна вернуть ненулевой результат. Это не является ошибочной ситуацией. Исключением является, если команда была завершена с сигналом (чаще всего с SIGTERM, который используется частично при выключении сервера базы данных) или ошибки командной оболочки (команда не найдена). После этих ошибок восстановление прекращается и сервер не стартует.

Не все запрашиваемые файлы будут файлами сегментов WAL, следует ожидать запросов файлов с расширениями .backup и .history. Также важно знать что базовое имя пути %p будет отлично от %f, не следует ожидать, что они взаимозаменяемы.

Сегменты WAL, не найденные в архиве, будут обнаруживаться в pg_xlog; это позволяет использовать последние не архивированные сегменты. Однако, доступные в архиве сегменты будут использованы прежде файлов из pg_xlog.

Обычно при восстановлении обрабатываются все доступные сегменты WAL для восстановления БД к текущему моменту времени (или настолько близко, насколько доступны сегменты WAL.) Таким образом, нормальное восстановление завершается сообщением «файл не найден», точное содержание сообщение зависит от выбора команды restore_command. Также может быть получено сообщение об ошибке при старте восстановления для файла с именем типа 00000001.history. Это считается нормальным и в сложных ситуациях восстановления не указывает на ошибку. Обсуждение этого приведено в 14.3.4.

При необходимости восстановления на определенный момент времени (например, до удаления основной таблицы неопытным администратором), требуется указать желаемую точку останова в recovery.conf. Точка останова, известная как «цель восстановления»,

может быть указана как в виде дата/время, так и завершением транзакции с заданным идентификатором ID. Из этого более применимо указание в виде дата/время, т.к. не существует инструментов, которые могут помочь в точном определении идентификатора транзакции ID.

Примечание. Точка останова должна быть позднее времени завершения базовой резервной копии, т.е. времени завершения pg_stop_backup. Базовая резервная копия не может быть использована для восстановления к моменту, когда она еще создавалась. (Для восстановления к этому времени требуется использование предыдущей базовой резервной копии и восстановления от нее.)

Если при восстановлении было обнаружено разрушение данных WAL, восстановление завершается на этом месте и сервер не запускается. В подобном случае процесс восстановления для нормального завершения должен быть запущен заново с указанием «цели восстановления» ранее точки сбоя. При сбое восстановления из-за внешней ошибки, такой как сбой системы или при потере доступности архива WAL, восстановление может быть просто перезапущено и продолжиться практически с места остановки. Перезапуск восстановления работает очень похоже точкам останова при нормальном функционировании: сервер периодически принудительно сбрасывает на диск свое состояние, и обновляет файл pg_control для отметки тех обработанных данных WAL, которые не требуется просматривать заново.

14.3.4. Линии времени

Способность восстанавливать БД к предыдущему состоянию по сложности сродни научно фантастическим историям о путешествии во времени или параллельным мирам. В первоначальной истории БД, возможно была удалена критичная таблица в 17:15 вечером во вторник, но это не было признано ошибкой до полудня среды. Было произведено восстановление БД на момент времени 17:14 вторника, после чего сервер был запущен заново. В этой истории таблица не была удалена. Позднее пришло осознание, что этого делать не стоило, и возникла потребность вернуться к определенному времени на утро среды. Не существует возможности это осуществить при работающей БД, т.к. при этом перезаписывается последовательность файлов сегментов WAL, ведущих ко времени, к которому необходимо вернуться. Таким образом, возникает необходимость различать серии записей WAL, сформированные после восстановления на конкретный момент времени, и существовавших в первоначальной истории.

Для разрешения рассмотренной проблемы PostgreSQL привносит понятие линии времени. Как только завершается восстановление, создается новая линия времени для идентификации серии записей WAL, формирующихся после восстановления. Идентификатор ID линии времени используется как часть имени файла сегмента WAL, таким образом, новая линия времени не перезаписывает данные WAL, сформированные предыдущими

линиями времени. Это позволяет архивировать несколько линий времени. И хотя это может казаться не очень полезным, часто это сильно помогает. Рассмотрим ситуацию, когда не известно точно, на какой момент времени необходимо восстановление, так что приходится выполнять несколько попыток восстановления на разное время до обнаружения наилучшего места ответвления от старой истории. Без использования линий времени, подобный процесс быстро привел бы к беспорядку. С линиями времени существует возможность восстановления к любому предыдущему состоянию, включая состояния на линии времени, от которых в последствии было принято решение отказаться.

Каждый раз после создания новой линии времени, PostgreSQL создает файл «истории линии времени», показывающий какая линия времени, из какой линии и в какой момент времени была ответвлена. Такие файлы истории необходимы системе для возможности отбора правильных файлов сегментов WAL при восстановления из архива, содержащего множество линий времени. Поэтому они архивируются в архив WAL аналогично файлам сегментов. Подобные файлы представляет собой небольшие текстовые файлы, не требующих много места и подходящих для практически бесконечного хранения (в отличии от файлов сегментов, которые значительно больше). При желании существует возможность добавления комментариев в файл истории для отметки о том, когда и зачем была создана конкретная линия времени. Такие комментарии особенно полезны в случае наличия большого количества различных линий времени, созданных в результате экспериментов.

Поведением по умолчанию является восстановление вдоль той же линии времени, в которой была создана резервная копия. При желании восстановления в некоторую порожденную линию времени (например, при возврате к некоторому состоянию, полученному после попытки восстановления), необходимо указать идентификатор `ID` целевой линии времени в `recovery.conf`. Не существует возможности восстановления в линии времени, ответвленные ранее, чем была создана базовая резервная копия.

14.3.5. Советы и примеры

В разделе приведены советы по конфигурированию непрерывного архивирования.

14.3.5.1. Автономные «горячие» резервные копии

Существует возможность использования средств резервного копирования PostgreSQL по созданию автономных «горячих» резервных копий. Такие копии не могут быть использованы для восстановления на заданный момент времени, но обычно требуют гораздо меньше времени на резервное копирование и восстановление, чем дампы `pg_dump`. (Также они значительно больше дампов `pg_dump`, и в некоторых случаях выигрыша по скорости может и не быть.)

Как и в случае с базовыми резервными копиями, самым простым способом создания «горячей» резервной копии является использование утилиты `pg_basebackup`. При

указании при вызове параметра -X, в резервную копию автоматически будет добавлен журнал транзакций, необходимый для ее использования. При этом не потребуется никаких дополнительных действий для последующего восстановления из резервной копии.

Если требуется большая гибкость, может быть использован более низкоуровневый подход. Для подготовки к автономным «горячим» резервным копиям необходимо установить режим wal_level в archive или выше и задать команду архивирования archive_command, осуществляющую архивирование только при существовании «переключающего» файла. Например:

```
archive_command = 'test ! -f /var/lib/pgsql/backup_in_progress ||
cp -i %p /var/lib/pgsql/archive/%f < /dev/null'
```

Указанная команда выполняет архивирование при существовании файла /var/lib/pgsql/backup_in_progress, а в другом случае просто возвращает нулевой результат выхода (позволяя PostgreSQL повторно использовать ненужные уже файлы WAL).

После такой подготовки, резервная копия может быть получена использованием следующего скрипта:

```
touch /var/lib/pgsql/backup_in_progress
psql -c "select pg_start_backup('hot_backup');"
tar -cf /var/lib/pgsql/backup.tar /var/lib/pgsql/data/
psql -c "select pg_stop_backup();"
rm /var/lib/pgsql/backup_in_progress
tar -rf /var/lib/pgsql/backup.tar /var/lib/pgsql/archive/
```

Сначала создается «переключающий» файл, разрешающий архивирование полных файлов WAL при их возникновении (/var/lib/pgsql/backup_in_progress). После выполнения резервного копирования, «переключающий» файл удаляется. Затем выполняется добавление архивированных фалов WAL в архив с резервной копией, чтобы и базовая резервная копия и все необходимые файлы WAL были частью одного архивного файла tar. Не следует забывать о добавлении в скрипт создания резервной копии обработки ошибок.

14.3.5.2. Сжатые архивные файлы журнала

Если имеет значение эффективное использование устройства хранения, существует возможность использования gzip для сжатия архивных файлов журнала:

```
archive_command = 'gzip < %p > /var/lib/pgsql/archive/%f'
```

При этом в процессе восстановления понадобится использование gunzip:

```
restore_command = 'gunzip < /mnt/server/archivedir/%f > %p'
```

14.3.5.3. Скрипты archive_command

В большинстве случаев для определения параметра archive_command выбирается скрипт, что делает запись в postgresql.conf достаточно простой:

```
archive_command = 'local_backup_script.sh "%p" "%f"'
```

Применение отдельного файла скрипта целесообразно при использовании в процессе архивирования более одной команды. Это позволяет управлять всей сложностью процесса внутри скрипта, который может быть написан на популярном скриптовом языке, таком как Bash или Perl.

Примеры требований, которые могут быть решены с помощью скрипта:

- копирование данных на безопасное внешнее хранилище данных;
- пакетирование файлов WAL для передачи их каждые три часа, т.к. это предпочтительнее чем по одному за раз;
- взаимодействие со сторонним ПО для создания резервных копий и восстановления;
- взаимодействие с ПО мониторинга для выдачи ошибок.

Примечание. При использовании скрипта `archive_command` рекомендуется включать `logging_collector`. Все сообщения, выдаваемые скриптом в стандартный поток ошибок `stderr`, будут появляться в журнале работы сервера, что позволяет легко диагностировать ошибки сложных конфигураций.

14.3.6. Предостережения

При такой записи существуют некоторые ограничения метода непрерывного архивирования. Следующие возможно будут решены в будущих версиях:

- Операции с хэш-индексами в настоящее время не записываются в журнал WAL, поэтому при восстановлении такие индексы обновлены не будут. Это означает, что любые добавляемые записи будут игнорированы индексом, модифицированные по-видимому исчезнут, а удаленные все еще будут доступны. Другими словами, в случае модификации таблицы с хэш-индексом возможно получение некорректных результатов при выполнении запросов на резервном сервере. Подобное ограничение рекомендуется обходить ручным выполнением `REINDEX` для каждого подобного индекса после завершения процедуры восстановления.
- Если команда `CREATE DATABASE` выполнялась во время получения базовой резервной копии, а затем шаблон БД, скопированный командой `CREATE DATABASE`, был модифицирован до завершения резервного копирования, существует вероятность при восстановлении распространения таких модификаций и на созданную БД, что нежелательно. Во избежание подобного риска не следует модифицировать шаблон БД во время получения базовой резервной копии.
- Команды `CREATE TABLESPACE` сохраняются в журнале WAL с абсолютными путями, и, следовательно, при восстановлении также создаются с сохраненными абсолютными путями. Это может быть нежелательно при восстановлении на отличной по конфигурации системе. Это может быть опасным даже при восстановлении

на той же самой системе, но в другой каталог данных: процесс восстановления перезапишет содержимое первоначальных табличных пространств.

Также следует отметить, что формат журнала WAL по умолчанию совершенно громоздкий, т.к. включает большое количество снимков дисковых страниц. Эти снимки страниц разработаны для поддержки восстановления после сбоев, т.к. может потребоваться исправление частично записанных страниц. В зависимости от программно-аппаратного обеспечения системы, подобный риск может быть пренебрежительно мал, и в этом случае возможно значительное уменьшение совокупного объема архивированных журналов путем отключения снимков страниц с помощью параметра `full_page_writes`. Выключение создания снимков страниц не мешает использовать журналы для восстановления на указанный момент времени. Областью будущих разработок является сжатие архивных файлов WAL, удалением ненужных копий страниц даже при установленном параметре `full_page_writes`. Тем временем администратор при желании может уменьшить количество снимков страниц, включенных в WAL, увеличением интервала создания контрольных точек насколько это возможно.

15. ВЫСОКАЯ ДОСТУПНОСТЬ, БАЛАНСИРОВКА НАГРУЗКИ И РЕПЛИКАЦИЯ

Серверы БД могут работать совместно для обеспечения возможности быстрого включения в работу вторичного сервера при отказе основного (высокая доступность), или для обеспечения возможности нескольких серверов поддерживать один и тот же набор данных (балансировка нагрузки). В идеале серверы БД должны очень тесно взаимодействовать. Веб-серверы, работающие со статическими веб-страницами, могут быть объединены достаточно легко путем балансировки нагрузки веб-запросов ко множеству серверов. Фактически, серверы БД только для чтения могут быть объединены так же относительно легко. В то же время, большинство серверов БД обрабатывают смесь запросов чтения/записи, а серверы, одновременно и читающие и пишущие, комбинировать значительно сложнее. Это связано с тем, что данные только для чтения требуют однократного их размещения на каждом сервере, тогда как операция записи в какой-нибудь сервер должна быть распространена на все серверы, чтобы впоследствии запросы на чтение к ним возвращали согласованные результаты.

Проблема синхронизации является сложной для совместной работы серверов. Поскольку не существует единственного решения, устраняющего влияния проблемы синхронизации во всех вариантах использования, существует множество таких решений. Каждое решение по-своему рассматривает эту проблему и минимизирует ее влияние для каждого конкретного случая.

Некоторые решения борются с проблемой синхронизации, допуская модификацию данных только одним сервером. Сервер, модифицирующий данные, называется сервер чтения/записи или «ведущий» (master). Серверы, отвечающие только на запросы чтения, называются резервными или «ведомыми» (slave). Серверы, которые недоступны до тех пор пока не станут «ведущими», называются серверы «теплого» резерва (warm standby), а те, которые могут принимать соединения для читающих запросов, называются серверы «горячего» резерва (hot standby).

Существуют синхронные решения, подразумевающие, что транзакция модификации данных не считается подтвержденной до тех пор, пока все серверы ее не подтвердят. Это гарантирует, что в процессе восстановления после отказа не будет потери данных и все серверы, обеспечивающие балансировку нагрузки, вернут согласованные данные, независимо от того, какой сервер будет опрошен. Напротив, асинхронные решения допускают некоторую задержку между подтверждением транзакции и ее распространением на другие серверы, что создает возможность потери ряда транзакций при переключении на резервный сервер, и того, что серверы, обеспечивающие балансировку нагрузки, могут возвращать немного устаревшие результаты. Асинхронное взаимодействие используется в случае, когда

синхронизация может быть слишком медленной.

Решения могут различаться по степени детализации. Некоторые имеют дело только с сервером БД в целом, тогда как другие обеспечивают контроль на уровне таблиц или БД.

В любом случае необходимо рассматривать вопросы производительности. Как обычно, существует определенный компромисс между функциональностью и производительностью. Например, полностью синхронное решение по медленному соединению более чем в два раза может снизить производительность, тогда как асинхронное приведет только к незначительным ее потерям.

15.1. Сравнение различных решений

Далее описываются различные решения по восстановлению после сбоев, репликации и балансировке нагрузки.

1) *Дисковая система коллективного доступа*.

Подобное решение не требует затрат на синхронизацию, т.к. при этом используется только одна копия БД. Применяется единая дисковая подсистема, разделяемая множеством серверов. В случае сбоя основного сервера БД, резервный сервер имеет возможность подхватить БД и восстановить работу с ней, как будто она была восстановлена после сбоя. Это позволяет осуществить быстрое восстановление без потери данных.

Функциональность разделяемого оборудования обычно обеспечивается сетевым хранилищами данных. Также возможно использование сетевой ФС, хотя при этом необходимо уделить внимание тому, чтобы ФС поддерживала всю функциональность POSIX (см. 7.2.1). Существенным ограничением этого метода является то, что при отказе или разрушении разделяемой дисковой подсистемы перестают функционировать как основные, так и резервные серверы. Другим следствием является требование отсутствия доступа резервного сервера к хранилищу до тех пор, пока работает основной.

2) *Блочная репликация ФС*.

Модифицированной версией функциональности разделяемой дисковой подсистемы является репликация на уровне ФС, когда все изменения в ФС отражаются в ФС, расположенной на другом компьютере. Единственное ограничение касается требования реализации отражения способом, гарантирующим согласованную копию ФС — особо важно, чтобы запись в резервную ФС осуществлялась в том же порядке, что и в основную. Популярным решением для Linux по репликации на уровне ФС является DRBD.

3) *Передача журнала транзакций*.

Серверы «теплого» и «горячего» резерва могут поддерживать согласованное состояние путем чтения потока записей журнала транзакций (WAL). При отказе основного сервера, резервный сервер содержит почти все данные основного сервера, и может быть быстро сделан основным. Это решение асинхронное и может быть применено только для всего сервера БД.

Резервный сервер может быть реализован с помощью передачи журнала на уровне ФС (см. 15.2) или с помощью потоковой репликации (см. 15.2.5), или комбинацией этих способов. Дополнительная информация приведена в 15.5.

4) *Репликация «Ведущий-резервный» (Master-Standby) на основе триггеров*.

В конфигурации репликации «Ведущий-резервный» все запросы модификации данных посылаются ведущему серверу. Ведущий сервер асинхронно посылает информацию о модификации данных резервному серверу. Резервный сервер может отвечать на запросы чтения во время работы ведущего. Ведомый сервер идеален для запросов к «хранилищу данных».

Slony-I является примером реализации подобной репликации, с детализацией до таблицы и поддержкой множества резервных серверов. Поскольку обновление резервных серверов осуществляется асинхронно (частями), возможна потеря данных в процессе восстановления после сбоя.

5) *Средства по-запросной репликации*.

При по-запросной репликации программа промежуточного уровня перехватывает каждый SQL-запрос и отсылает его одному или всем серверам. Каждый сервер функционирует независимо. Запросы чтения-записи отсылаются всем серверам, тогда как запросы чтения могут быть посланы одному серверу, обеспечивая распределение нагрузки при чтении.

Если запросы передаются немодифицированными, функции такие как `random()`, `CURRENT_TIMESTAMP`, и последовательности будут иметь различные значения на разных серверах. Это обусловлено тем, что каждый сервер функционирует независимо, а SQL запросы разбрасываются по ним (т.к. не модифицируют данные). Если это неприемлемо, либо ПО промежуточного уровня, либо приложение должны получать подобные значения от одного сервера и затем использовать их в запросах записи. Другой вариант заключается в использовании подобной репликации с традиционным конфигурацией ведущий-резервный, т.е. запросы модификации данных отправляются только мастеру и распространяются на резервные серверы с помощью репликации ведущий-резервный, а не с помощью ПО промежуточного уровня. Также необходимо обратить внимание на то, что все транзакции подтверждаются или откатываются на всех серверах, поскольку применяется двухфазное подтверждение

транзакций (PREPARE TRANSACTION и COMMIT PREPARED). Примерами подобных приложений являются Pgpool-II и Continuent Tungsten.

6) *Асинхронная репликация со множеством «ведущих» серверов*.

Для серверов, которые не имеют постоянного соединения, таких как ноутбуки и удаленные серверы, сохранение данных в согласованном состоянии является сложной задачей. При использовании асинхронной репликации со множеством ведущих серверов, каждый сервер работает независимо, и периодически связывается с другими серверами для идентификации конфликтующих транзакций. Конфликты могут разрешаться пользователями или с помощью правил разрешения конфликтов. Примером подобной репликации является Bucardo.

7) *Синхронная репликация со множеством ведущих серверов*.

При синхронной репликации со множеством ведущих серверов, каждый сервер может принимать запросы на запись, и модифицированные данные пересылаются с этого сервера на все остальные перед подтверждение транзакции. Высокая активность записи может вызвать чрезмерные блокировки, ведущие к ухудшению производительности. Фактически, производительность записи зачастую хуже чем при использовании одного сервера. Запросы на чтение могут передаваться любому серверу. Некоторые реализации используют разделяемую дисковую подсистему для снижения затрат на взаимодействие. Синхронная репликация со множеством ведущих серверов является наилучшим вариантом для большинства нагрузок чтения, хотя наибольшим преимуществом является то, что любой сервер может принимать запросы на запись — не требуется разделения нагрузки между ведущими и ведомыми серверами, а поскольку модификация данных передается от одного сервера другим, не возникает проблемы с не детерминированными функциями типа random().

PostgreSQL не предоставляет подобной репликации, хотя механизмы двухфазного подтверждения транзакций (PREPARE TRANSACTION и COMMIT PREPARED) могут быть использованы в приложениях или ПО промежуточного уровня.

8) *Коммерческие решения*.

Поскольку PostgreSQL имеет открытый исходный код и легко расширяем, ряд компаний использует PostgreSQL и создают коммерческие решения с закрытым кодом со своими уникальными возможностями восстановления после сбоев, репликации и балансировки нагрузки.

В таблице 109 приведены возможности рассмотренных решений.

Таблица 109 – Матрица свойств методов обеспечения высокой доступности, балансировки нагрузки и репликации

Свойство	Общая дисковая система	Репликация ФС	Передача журнала транзакций	Репликация «Ведущий-резервный»	По-запросная репликация	Асинхронная репликация с множеством ведущих серверов	Синхронная репликация с множеством ведущих серверов
Наиболее общая реализация	NAS	DRDB	Потоковая репликация	Slony	pgpool-II	Bucardo	
Способ взаимодействия	Общие диски	Дисковые блоки	WAL	Строки таблиц	SQL	Строки таблиц	Строки таблиц и блокировки строк
Не нужно специального оборудования		*	*	*	*	*	*
Поддержка множества ведущих серверов					*	*	*
Не нужен ведущий сервер	*		*		*		
Не нужно ожидание множества серверов	*		`sync=off`	*		*	
Отказ ведущего сервера не влечет потери данных	*	*	`sync=on`		*		*
Ведомые сервера принимают только запросы на чтение			с `hot`	*	*	*	*
По-табличная детализация				*		*	*
Не нужно разрешение конфликтов	*	*	*	*			*

Существует несколько решений, не попавших в указанные категории:

- *Разделение данных*.

При разделении данных таблица разбивается на наборы данных. Каждый набор

может модифицироваться только одним сервером. Например, данные могут быть разбиты по территориальному признаку (офису), в Лондоне и Париже, с серверами в каждом офисе. При необходимости объединения данных из Лондона и Парижа, приложение должно запросить оба сервера, или может быть использована репликация «ведущий/резервный» для хранения копии только для чтения данных, расположенных в другом офисе.

- *Параллельное исполнение запросов на множестве серверов*.

Большинство рассмотренных решений позволяют множеству серверов обслуживать множество запросов, но ни одно не позволяет выполнять один запрос одновременно множеством серверов для быстрого его завершения. Это обычно совершается разделением данных между серверами и выполнением каждым сервером своей части запроса и возвратом результатов центральному серверу, где они объединяются и передаются пользователю. Pgpool-II поддерживает такую возможность. Также это может быть реализовано с помощью инструмента PL/Proxy.

15.2. Резервные серверы на основе передачи журнала транзакций

Непрерывное архивирование может быть использовано для создания конфигураций кластера высокой готовности с одним или более резервными серверами, готовых перехватить функционирование при сбое основного сервера. Эта возможность широко известна под именем «теплый» режим резервирования или передача журнала транзакций.

Для обеспечения этого основной и резервный серверы работают совместно, хотя связь между ними при этом достаточно слабая. Основной сервер работает в режиме непрерывного архивирования, а каждый резервный в режиме непрерывного восстановления, читая файлы WAL с основного. Такая реализация не требует никаких изменений в БД или таблицах, и потому предполагает меньшие затраты на администрирование по сравнению с некоторыми другими способами репликации. К тому же подобная конфигурация относительно слабо влияет на производительность основного сервера.

Непосредственное перемещение записей WAL с одного сервера БД на другой обычно описывается как передача журнала транзакций. PostgreSQL реализует передачу журнала транзакций на уровне файлов, что означает передачу записей WAL одним файлом (сегментом WAL) за раз. Файлы WAL (16 МБ) могут быть легко переданы на любое расстояние будь то соседняя система, система расположенная в том же месте, или расположенная на большом расстоянии. Требуемая пропускная способность при такой способе зависит от количества исполняемых транзакций в единицу времени на основном сервере. Передача журнала транзакций на уровне записей более детализирована и обеспечивает потоковую передачу изменений WAL по сети (см. 15.2.5).

Следует заметить, что передача журнала транзакций носит асинхронный характер, т.е. записи WAL передаются только после подтверждения транзакции. Таким образом, существует временное окно для потери данных в случае сбоя основного сервера: еще не переданные транзакции будут потеряны. Размер такого окна потери данных может быть ограничен с помощью параметра `archive_timeout`, который может быть уменьшен вплоть до нескольких секунд, если это требуется. Однако, такие малые значения в значительной мере увеличивают требования к пропускной способности для передачи журнала транзакций. Потоковая репликация (см. 15.2.5) обеспечивает гораздо меньшее окна потери данных.

Скорость восстановления достаточно хорошая тогда, когда резервный сервер от полной доступности отделяет обычно только небольшое время с момента активации. В результате такую способность можно скорее рассматривать как конфигурацию «теплого» резервирования, чем решение высокой готовности. Восстановление сервера из архивной базовой резервной копии и доведение его до рабочего состояния может занять значительное время, так что этот способ предлагает решение только восстановлению после сбоя, а не для высокой готовности. Резервный сервер может также быть использован для исполнения запросов «только чтения», в этом случае он называется сервером «горячего» резерва. Дополнительная информация приведена в 15.5.

15.2.1. Планирование

Обычно разумно создавать основной и резервные серверы насколько возможно похожими, как минимум с точки зрения сервера БД. В частности, пути, ассоциированные с табличными пространствами, передаются как есть, так что и основной и резервный серверы должны иметь одинаковые пути монтирования табличных пространств, если они используются. Следует обратить внимание, что, если на основном сервере выполняется `CREATE TABLESPACE`, новая точка монтирования должна быть создана перед ее выполнением как на основном сервере, так и на всех резервных серверах. Аппаратное обеспечение не обязательно должно быть одинаковым, но опыт показывает, что поддержка двух идентичных систем на протяжении их жизни и функционирования гораздо легче чем различных. В любом случае архитектура систем должна быть одинаковой — передача журнала с 32-х разрядной системы в 64-х разрядную не работает.

В общем случае, передача журнала между отличающимися основной версией серверами PostgreSQL невозможна. Политикой группы разработки PostgreSQL является отказ от внесения изменений в форматы хранения на диске при обновлениях неосновной версии, поэтому с большой вероятностью запущенные на основном и резервном серверах отличающиеся неосновной версией варианты PostgreSQL будут взаимодействовать успешно. В то же время нормальной поддержки такой ситуации не предлагается, в связи с чем рекомендуется держать основной и резервный серверы по возможности на одной версии PostgreSQL.

15.2.2. Функционирование резервного сервера

В режиме резервирования сервер непрерывно применяет журнал транзакций WAL, полученный с основного сервера. Резервный сервер может получать WAL из архива (с помощью команды `restore_command`) или непосредственно с основного сервера через TCP соединение (потоковая репликация). Резервный сервер также может пытаться восстановиться с журнала WAL, расположенного в подкаталоге `pg_xlog` кластера резервного сервера. Это обычно происходит после перезапуска сервера, когда резервный сервер снова выполняет WAL, полученный с основного перед перезапуском, но существует возможность помещать вручную файлы в `pg_xlog` в любое время для их применения.

В процессе запуска резервный сервер начинает восстанавливать весь журнал WAL, доступный в архиве, путем вызова команды `restore_command`. По достижению конца журнала WAL в архиве и получению отказа команды `restore_command`, он пытается восстановить журнал WAL из каталога `pg_xlog`. Если это не удалось, и настроена потоковая репликация, резервный сервер пытается установить соединения с основным сервером и запустить потоковую репликацию с последней корректной записи, найденной в архиве или `pg_xlog`. Если это не удалось, или не настроена потоковая репликация, или соединение было завершено, резервный сервер возвращается к первому шагу и снова пытается восстановить файл WAL из архива. Это цикл по опросу архива, `pg_xlog` и потоковой репликации продолжается до остановки сервера или обнаружения необходимости процедуры восстановления после сбоя с помощью триггерного файла.

Режим резервирования завершается, и сервер переключается в нормальный режим функционирования, в случае запуска `pg_ctl promote` или обнаружения триггерного файла (`trigger_file`). Перед началом процедуры восстановления после сбоя выполняются весь журнал WAL, доступный в архиве или `pg_xlog`, при этом попытка соединения с основным сервером не выполняется.

15.2.3. Подготовка ведущего сервера

На основном сервере необходимо настроить непрерывное архивирование в архивный каталог, доступный для резервных серверов, как описано в 14.3. Архивный каталог должен быть доступен с резервного сервера, даже если основной не функционирует, т.е. он должен располагаться на самом резервном сервере или на другом доверенном сервере, но не на основном.

При использовании потоковой репликации на основном сервере должна быть настроена аутентификация, позволяющая устанавливать соединения для репликации с резервных серверов. т.е. должна быть создана роль и для нее соответствующая запись в `pg_hba.conf` со значением `replication` в поле `database`. Также параметр `max_wal_senders` должен быть установлен в достаточно большое значение в конфи-

гурационном файле основного сервера. Если используются слоты репликации, убедитесь, что параметр max_replication_slots установлен в достаточно большое значение.

Примечание. Использование слотов репликации возможно только в СУБД версии 9.6.

Для инициализации резервного сервера должна быть создана базовая резервной копия, как описано в 14.3.2.

15.2.4. Настройка резервного сервера

Для настройки резервного сервера необходимо в первую очередь восстановить базовую резервную копию с основного сервера (см. 14.3.3). В каталоге кластера резервного сервера должен существовать командный файл восстановления recovery.conf, при этом должен быть включен конфигурационный параметр standby_mode. В recovery.conf должна быть указан простая команда копирования файлов из архива WAL. При планировании использования множества резервных серверов в целях повышения отказоустойчивости, требуется установить recovery_target_timeline в значение 'latest', чтобы позволить резервному серверу следовать изменению линии времени, возникающей в процессе восстановления после сбоя на другом резервном сервере.

Примечание. Не следует использовать pg_standby или схожие средства совместно с описываемым встроенным режимом резервирования. restore_command в случае отсутствия файла должна незамедлительно возвращать управление; сервер при необходимости может попытаться выполнить команду повторно. Использование средств типа pg_standby описано в 15.4.

Для использования потоковой репликации требуется указать в параметре primary_conninfo строку соединения в формате libpq, включая имя сервера (или IP-адрес) и другую необходимую для установления соединения с основным сервером информацию. Если ведущий сервер требует парольную аутентификацию, в primary_conninfo также должен быть указан пароль.

Если резервный сервер планируется использовать в целях обеспечения отказоустойчивости, настройки архивирования WAL, соединений и аутентификации должны быть такими же, что и на основном сервере, поскольку по завершению процедуры восстановления после сбоя резервный сервер может стать основным.

При использовании архива WAL его размер может быть минимизирован с помощью команды, заданной в archive_cleanup_command и удаляющей файлы, которые больше не требуются резервному серверу. Утилита pg_archivecleanup специально разработана для использовании в archive_cleanup_command в обычной конфигурации с одним резервным сервером. Не следует забывать, что при использования архивирования WAL для создания резервных копий, необходимо иметь полный набор файлов, начиная как минимум

с последней базовой резервной копии, даже если они не больше не требуются резервному серверу.

Пример

Простой пример recovery.conf:

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
restore_command = 'cp /path/to/archive/%f %p'
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

Существует возможность использования любого количества резервных серверов, но в случае применения потоковой репликации, на основном сервере следует устанавливать значение max_wal_senders достаточно большим для того, чтобы резервные серверы могли подключаться одновременно.

15.2.5. Потоковая репликация

Потоковая репликация позволяет резервному серверу находиться в более актуальном состоянии, чем при использовании передачи журнала WAL. Резервный сервер подключается к основному, который передает записи журнала WAL незамедлительно после их создания без ожидания заполнения файла WAL.

По умолчанию потоковая репликация работает асинхронно (см. 15.2.8), что вносит небольшую задержку между подтверждением транзакции на основном сервере и видимостью этих изменений на резервном. Однако, задержка в этом случае значительно меньше, чем при передаче журнала WAL файлами, обычно порядка секунды, в случае достаточной производительности резервного сервера для работы с такой нагрузкой. При потоковой репликации параметр archive_timeout не требуется для уменьшения окна потерь.

При использовании потоковой репликации без архивирования журнала WAL на уровне файлов, сервер может утилизировать старые сегменты WAL, прежде чем резервный получил их. Если это происходит, в резервному серверу необходимо будет повторно инициализироваться новой базовой резервной копии. Этого можно избежать, установив wal_keep_segments до достаточно большого значения, чтобы убедиться, что сегменты WAL не удалены слишком рано или настроив слот для репликации в режиме ожидания. Если создан архив WAL, который может быть применен на резервном сервере, то эти действия не требуются, так как этот архив может быть распакован для того, чтобы догнать основной.

Для потоковой репликации требуется настроить резервный сервер для передачи журнала с помощью файлов, как описано в 15.2. Шагом, переводящим резервный сервер на потоковую репликацию, служит задание строки соединения с основным сервером primary_conninfo в файле recovery.conf. На основном сервере должны быть соответствующим образом настроены параметр listen_addresses и опции аутентификации в

pg_hba.conf для возможности подключения резервного сервера к псевдоБД replication на основном сервере (см. 15.2.5.1).

На системах, поддерживающих опцию сокетов keepalive, задание опций tcp_keepalives_idle, tcp_keepalives_interval и tcp_keepalives_count помогает основному серверу быстро обнаружить разорванные соединения.

Следует устанавливать максимальное количество одновременных подключений от резервных серверов (см. max_wal_senders).

При запуске резервного сервера в случае корректного задания параметра primary_conninfo, резервный сервер подключается к основному после обработки доступных в архиве WAL-файлов. Если соединение установлено успешно, на резервном сервере порождается процесс walreceiver, а на основном соответствующий ему процесс walsender.

15.2.5.1. Аутентификация

Очень важно обеспечить настройку прав доступа для репликации таким образом, чтобы только доверенные пользователи могли читать поток WAL, т.к. из этого потока может быть легко получена защищенная информация. Резервные серверы должны аутентифицироваться на основном с помощью учетной записи суперпользователя или учетной записи, обладающей привилегией REPLICATION. Рекомендуется для выполнения репликации создавать выделенную учетную запись с привилегиями REPLICATION и LOGIN. Несмотря на то, что привилегия REPLICATION предоставляет очень высокие полномочия, она не позволяет пользователю модифицировать какие-либо данные на основной системе, тогда как привилегия SUPERUSER дает такую возможность.

Аутентификация клиента для репликации управляется с помощью строки файла pg_hba.conf, содержащей в поле database значение replication. Например, если резервный сервер функционирует на системе с IP-адресом 192.168.1.100, и для репликации используется учетная запись foo, администратор может добавить следующую запись в файл pg_hba.conf на основном сервере:

```
# Позволяет пользователю "foo" с адреса 192.168.1.100 устанавливать
# соединение к основному серверу для репликации при указании пароля
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    replication     foo             192.168.1.100/32        md5
```

Имя системы и номер порта основного сервера, имя пользователя и его пароль для установки соединения задаются в файле recovery.conf. Пароль также может быть задан в отдельном файле пароля ~/.pgpass на резервном сервере (в качестве имени БД используется replication). Например, если основный сервер запущен на порту 5432 по адресу 192.168.1.50, имя пользователя foo и пароль foopass, администратор может

добавить следующую запись в файл recovery.conf на резервном сервере:

```
# Резервный сервер соединяется с основным сервером по адресу 192.168.1.50
# и порту 5432 от имени пользователя "foo" с паролем "foopass".
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
```

15.2.5.2. Контроль

Важным индикатором правильного функционирования потоковой репликации является количество записей WAL, сгенерированных основным сервером, но еще не примененных на резервном. Вычислить этот лаг можно путем сравнения текущей позиции WAL на основном сервере с последней полученной позицией WAL на резервном. Указанные показатели могут быть получены с помощью функций pg_current_xlog_location на основном сервере и pg_last_xlog_receive_location на резервном соответственно (см. таблицы 91 и 92). Последняя полученная позиция WAL на резервном сервере также отображается в статусе процесса WAL receiver, выводимом с помощью утилиты ps.

Список процессов WAL sender может быть выбран с помощью представления pg_stat_replication. Большое различие между полями pg_current_xlog_location и sent_location указывает на работу основного сервера под большой нагрузкой, а разница между sent_location и pg_last_xlog_receive_location на резервном сервере может указывать на сетевую задержку или работу резервного сервера под большой нагрузкой.

15.2.6. Слоты репликации

Примечание. Механизм слотов репликации может быть использован только в СУБД версии 9.6.

Слоты репликации гарантируют, что основной сервер не удалит сегменты WAL, пока они не были получены всеми резервными, и что основной сервер не удалит строки, которые могут привести к конфликту восстановления (см. 15.5.2), даже когда резервный отключен.

Вместо использования слотов репликации, возможно удаление старых сегментов WAL помощью wal_keep_segments(см 8.6.1), или с помощью сохранения сегментов в архиве с помощью archive_command(см. 8.5.3). Однако, эти методы часто приводят к сохранению большего числа сегментов WAL, чем требуется, в то время как слоты репликации сохраняют только необходимое количество сегментов. Преимущество этих методов является то, что они связаны требованием пространства для pg_xlog; в настоящее время нет способов сделать это с помощью слотов репликации.

Точно так же, hot_standby_feedback и vacuum_defer_cleanup_age (см. 8.6.2) обеспечивают защиту против удаления последних релевантных строк с помощью вакуума, но если первый параметр не дает никакой защиты в течение любого периода времени, когда резервный сервер выключен, то последний параметр часто должен быть установлен на

высокое значение, чтобы обеспечить адекватную защиту. Использование слотов репликации позволяет преодолеть эти недостатки.

15.2.6.1. Управление слотами репликации

Каждый слот репликации имеет имя, которое может содержать буквы в нижнем регистре, цифры и символы подчеркивания.

Существующие слоты репликации и их состояния отображаются в представлении `pg_replication_slots`.

Слоты могут быть созданы и удалены как через протокол репликации, так и через SQL функции (см. 6.26.6).

15.2.6.2. Пример конфигурации

Вы можете создать слот репликации наподобие этого:

```
postgres=# SELECT * FROM pg_create_physical_replication_slot('node_a_slot');
  slot_name  | xlog_position
-------------+--------------
 node_a_slot |

postgres=# SELECT * FROM pg_replication_slots;
  slot_name  | slot_type | datoid | database | active | xmin | restart_lsn
-------------+-----------+--------+----------+--------+------+-------------
 node_a_slot | physical  |        |          | f      |      |
(1 row)
```

Что резервный сервер использовал этот слот, имя слота `primary_slot_name` должно быть указано в `recovery.conf` резервного сервера. Вот простой пример:

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
primary_slot_name = 'node_a_slot'
```

15.2.7. Каскадная репликация

Механизм каскадной репликации позволяет резервному серверу принимать соединения для репликации и передавать записи WAL другим резервным серверам, выступая в качестве ретранслятора.

Резервный сервер функционирующий одновременно и как получатель и как отправитель называется каскадным резервным сервером. Резервные серверы, непосредственно подключенные к основному серверу, называются вышестоящими, а те, что подключены к другим резервным серверам — нижестоящими. Каскадная репликация не накладывает ограничения на количество или взаимное расположение нижестоящих серверов, хотя каждый нижестоящий должен быть соединен с одним вышестоящим, каждый из которых в свою очередь должен быть подключен к одному основному/основному серверу.

Каскадный резервный сервер отправляет не только записи WAL, полученные с основного сервера, но и те, которые восстанавливаются им из архива. Таким образом, даже если соединение репликации с вышестоящим резервным сервером разрывается, потоковая репликация продолжается на нижестоящие до тех пор, пока доступны новые записи WAL.

Каскадная репликация в настоящий момент асинхронная. Настройки синхронной репликации (см. 15.2.8) для каскадной репликации не действуют.

Сервер «горячего» резерва производит уведомление вышестоящего сервера независимо от своего каскадного расположения.

Если вышестоящий резервный сервер становится новым основным, нижестоящие резервные серверы продолжают потоковую репликацию с нового основного в случае указания значения `'latest'` в параметре `recovery_target_timeline`.

Для использования каскадной репликации, требуется настроить каскадный резервный сервер так, чтобы он мог принимать соединения репликации (параметры `max_wal_senders`, `hot_standby` и опции аутентификации `pg_hba.conf`). На нижестоящих резервных серверах должна быть задана строка соединения с вышестоящим каскадным резервным сервером `primary_conninfo`.

15.2.8. Синхронная репликация

По умолчанию в PostgreSQL потоковая репликация асинхронная. В случае сбоя основного сервера, некоторые подтвержденные транзакции могут быть еще не реплицированы на резервный сервер, что приводит к потере данных. Количество потерянных данных пропорционально задержки репликации на момент сбоя.

Синхронная репликация предлагает возможность подтверждения о том, что все совершенные в транзакции изменения были переданы одному синхронному резервному серверу. Это обеспечивает стандартный уровень долговечности (durability), предполагаемый механизмом транзакций. Данный уровень защиты известен в теории как «2-safe» репликация.

При запросе синхронной репликации каждое подтверждение пишущей транзакции будет ожидать, пока не получено уведомление о том, что подтверждение было записано в журнал транзакций на диске на обоих (основной и резервном) серверах. Потеря данных возможна только в случае одновременного сбоя основного и резервного серверов. Это может обеспечить значительно более высокий уровень надежности при должном внимании и управлении администратором двумя серверами. Ожидание подтверждения увеличивает уверенность пользователя в том, что данные не будут потеряны в случае сбоя, но это также увеличивает время отклика для такой транзакции. Минимальное время ожидания формируется из времени прохождения сигнала основного сервера до резервного и обратно.

Только читающие транзакции и откаты транзакций не требуют ожидания подтвержде-

ния от резервных серверов. Завершения вложенных транзакций не ожидают подтверждения от резервных серверов, подтверждения ожидают только завершения транзакции верхнего уровня. Продолжительные операции, такие как загрузка данных или построение индекса, не ожидают подтверждения до финального завершения. Все операции с двух-фазными транзакциями требуют ожидания подтверждения, включая как их подготовку, так и их завершение.

15.2.8.1. Базовая конфигурация

После настройки потоковой репликации для настройки синхронной репликации требуется только один конфигурационный шаг: для параметра `synchronous_standby_names` должно быть задано непустое значение. Параметр `synchronous_commit` должен быть установлен в значение `on`, но поскольку это значение по умолчанию, обычно изменений не требуется (см. 8.5.1 и 8.6.2). Такая настройка приводит к тому, что каждое завершение транзакции ожидает подтверждение записи резервным сервером этого изменения в надежное хранилище. Параметр `synchronous_commit` может быть задан отдельными пользователями, так что он может задаваться в конфигурационном файле для конкретных пользователей или баз данных, или динамически с помощью приложений, в целях контроля надежности по каждой транзакции.

После фиксации завершения транзакции на диске основного сервера резервному серверу посылается соответствующая запись WAL. Резервный сервер посылает ответное сообщение каждый раз, когда новая партия WAL данных записывается на диск, если параметр `wal_receiver_status_interval` на резервном сервере не установлен в значение `0`. Если резервный сервер является первым сервером, указанным в `synchronous_standby_names` на основном сервере, ответное сообщение от этого сервера будет использовано для прекращения ожидания пользователями подтверждения того, что уведомление о фиксации нужной записи завершения транзакции было получено. Эти параметры позволяют администратору задать, какие из резервных серверов должны быть синхронными. Следует отметить, что данная конфигурация предназначена только для основного сервера. Перечисленные резервные серверы должны быть подключены к основному напрямую; основной сервер ничего «не знает» о нижестоящих резервных серверах при использовании каскадной репликации.

Установка `synchronous_commit` в значение `remote_write` приводит к тому, что каждое завершение транзакции ожидает подтверждения о получении этой записи резервным сервером и передачи ее в его ОС, но без ожидания сброса этих данных на дисковую систему резервного сервера. Такое значение параметра обеспечивает меньшую гарантию надежности, чем значение `on`: резервный сервер может потерять данные вследствие сбоя ОС, но не сбоя PostgreSQL. Однако, на практике это полезная опция, поскольку позволяет уменьшить время ответа для транзакций. Потеря данных может возникнуть только в случае

одновременного сбоя основного и резервного серверов и разрушения БД на основном сервере.

Ожидание пользователей будет прервано, если будет запрошено быстрое выключение. Однако, как и при использовании асинхронной репликации, сервер не будет полностью остановлен до тех пор, пока все невыполненные WAL записи не будут переданы подключенным в данный момент резервным серверам.

15.2.8.2. Планирование для производительности

Синхронная репликация, как правило, требует тщательного планирования и правильного размещения резервных серверов для обеспечения приемлемой производительности приложений. Ожидание не использует системные ресурсы, но блокировки транзакции сохраняются до подтверждения передачи. В результате неосторожное использование синхронной репликации может снизить производительность приложений баз данных из-за увеличения времени отклика и высокой конкуренции транзакций.

PostgreSQL позволяет разработчикам приложений определить требуемый уровень надежности с помощью репликации. Он может быть определен для системы в целом, а также и для конкретного пользователя или подключения, или даже для отдельных операций.

Если синхронная репликация определена на прикладном уровне (на основном сервере) реализуется синхронная репликация наиболее важных изменений, не замедляя остальную часть общей нагрузки. Возможность управления репликацией на прикладном уровне является важным и практичным инструментом для совмещения преимущества синхронной репликации и обеспечения высокой производительности приложения.

Следует учитывать, что пропускная способность сети должна быть выше, чем скорость генерации данных WAL.

15.2.8.3. Планирование для высокой надежности

Завершения транзакции, выполненные при значении `on` параметра `synchronous_commit`, будут ожидать ответа от синхронного резервного сервера. Ответ может не прийти, если последний, или единственный резервный сервер выйдет из строя.

Лучшее решение для предотвращения потери данных — убедиться, что не потерян последний оставшийся синхронный резервный сервер. Это может быть достигнуто перечислением нескольких потенциальных синхронных резервных серверов с помощью `synchronous_standby_names`. Первый указанный резервный сервер будет использоваться как синхронный резервный сервер. Остальные серверы, перечисленных после него, возьмут на себя роль синхронного резервного сервера, если первый выйдет из строя.

Когда резервный сервер первый раз подключается к основному, он еще не синхронизирован с ним должным образом. Это описывается как режим наверстывания. После того

как разрыв между резервным сервером и основным достигает нуля, происходит переход в режим потоковой репликации реального времени. Период наверстывания может продолжаться достаточно долго с момента создания резервного сервера. Если резервный сервер выключен, то период наверстывания будет увеличен на то время, пока резервный сервер выключен. Резервный сервер может стать синхронным только после того, как он достигает режима потоковой репликации реального времени.

Если основный сервер перезагружается во время ожидания подтверждения, то эти ожидающие транзакции будут помечены полностью подтвержденными только после того, как база данных мастера будет восстановлена. Не существует способа убедиться, что все резервные серверы получили все необработанные данные WAL на момент падения основного сервера. Некоторые транзакции могут не быть отмеченными как подтвержденные на резервном сервере, даже если они отражены как подтвержденные на основном. Существует лишь гарантия, что приложение не получит явное подтверждение успешной транзакции, до тех пор, пока WAL-данные не будут точно полученными резервным сервером.

В случае потери последнего резервного сервера, следует отключить `synchronous_standby_names` и перезагрузить конфигурационный файл на основном сервере.

Если основный сервер изолирован от остальных резервных серверов, следует выполнить процедуру восстановления после сбоя на лучшем кандидате из оставшихся резервных серверов.

Если нужно заново создать резервный сервер во время ожидания транзакций, следует убедиться, что команды `pg_start_backup()` и `pg_stop_backup()` выполняются в сессии с `synchronous_commit = off`, в противном случае эти запросы будут вечно ждать появления резервного сервера.

15.3. Восстановление после сбоев

В случае отказа основного сервера, резервный сервер должен начать процедуру восстановления после сбоя.

В случае отказа резервного сервера процедура восстановления не должна осуществляться. Если резервный сервер может быть перезапущен, даже если спустя некоторое время, сразу же начнется процедура восстановления, используя преимущества перезапускаемого восстановления. Если же резервный сервер не может быть перезапущен, требуется создание нового резервного сервера.

В случае сбоя основного сервера и перехода резервного в режим основного, после чего был перезапущен старый основный, требуется наличие механизма, информирующего его о том, что он перестал быть основным. Иногда это называется `STONITH`, что является

акронимом выражения «Shoot The Other Node In The Head». Это необходимо для исключения ситуации, когда обе системы считают себя основными, что приводит к конфликтам и, в конечном счете, к потере данных.

Множество отказоустойчивых кластеров используют только две системы, основную и резервную, соединенных каким-либо механизмом непрерывной проверки соединения между ними и жизнеспособности основного сервера. Также возможно использование третьей системы (называемой следящим сервером) для предотвращения некоторых ситуаций нежелательного восстановления после сбоя, но дополнительная сложность такого решения не заслуживает особого внимания, если оно не было реализовано без должного внимания и серьезного тестирования.

PostgreSQL не предоставляет системного ПО, необходимого для обнаружения отказа основного сервера и уведомления резервной системы и резервного сервера БД. Существует большое количество подобных инструментов, которые хорошо интегрируются с ОС для успешного восстановления после сбоев, такие, например, как миграция IP-адреса.

После операции восстановления на резервом сервере функционирующим остается только один сервер. Такое состояние известно как неполноценное, вырожденное состояние. Прежний резервный сервер становится основным, а основный находится и может продолжать находиться в нерабочем состоянии. Для возвращения к нормальному функционированию требуется полное пересоздание резервного сервера, как из бывшей основной системы, когда она восстановится, так и возможно из новой системы. После завершения, основный и резервный сервер могут рассматриваться как поменявшиеся ролями. В некоторых случаях для обеспечения резервного копирования с нового основного сервера, во время создания нового резервного сервера, используется третья система, хотя очевидно это является усложнением конфигурации и действующих процессов.

Таким образом, переключение с основного на резервный сервер может быть выполнено быстро, но требует некоторого времени для подготовки отказоустойчивого кластера. Периодическое переключение с основного сервера на резервный удобно, т.к. это позволяет попеременно останавливать каждую систему для регламентного обслуживания. Это обеспечивает тестирование механизма восстановления после сбоя, для гарантирования его работы на случай необходимости. Рекомендуется фиксировать процедуры администрирования в письменном виде.

Для запуска процедуры восстановления после сбоя на резервном сервере на основе передачи журнала транзакций, необходимо выполнить `pg_ctl promote` или создать триггерный файл, с именем, заданным параметром `trigger_file` в файле `recovery.conf`. Если предполагается применение `pg_ctl promote`, параметр `trigger_file` не требуется. При использовании дополнительных серверов, предназначенных только для выполнения

запросов на чтение с целью разгрузки основного, а не для обеспечения отказоустойчивости, запускать на них процедуру восстановления после сбоя не требуется.

15.4. Альтернативные способы передачи журнала

Альтернативой встроенному режиму резервирования, описанному в предыдущих разделах, служит использование команды `restore_command`, опрашивающей архив журнала WAL. Этот способ был единственным в версии 8.4 и ниже. Для его использования требуется установка `standby_mode` в значение `off`, поскольку необходимые для резервирования действия задаются внешними средствами. Пример подобной реализации приведен в модуле `pg_standby`.

Следует отметить, что в этом режиме, сервер применяет за раз целый файл WAL так, что в случае использования резервного сервера для выполнения запросов («горячее» резервирование), будет задержка между изменениями на основном сервере и их видимостью на резервном, соответствующая времени полного заполнения WAL-файла. Параметр `archive_timeout` может быть использован для уменьшения этой задержки. Потоковая репликация не может комбинироваться с этим методом.

Операция осуществляется одинаково на основном и резервных серверах в рамках общей нормальной процедуры непрерывного архивирования и восстановления. Единственной точкой взаимодействия между серверами является разделяемые архивы файлов WAL: основной пишет в архив, резервные — читают из него. Внимание должно быть уделено тому, чтобы архивы WAL от различных основных серверов не перемешивались и не конфликтовали. Архив не обязательно должен быть большим, поскольку требуется только для резервирования.

Механизмом, позволяющим двум слабосвязанным серверам работать совместно, является команда `restore_command`, реализованная на резервном сервере таким образом, что при запросе нового файла WAL она ждет когда он станет доступен на основном сервере. Параметр `restore_command` задается в файле `recovery.conf` резервного сервера. Нормальная процедура восстановления запрашивает файл из архива WAL, сообщая об ошибке, если он недоступен. При работе резервного сервера нормальным является недоступность следующего файла WAL, в связи с чем требуется ожидание его появления. Для файлов, заканчивающихся на `.backup` или `.history`, ожидания не требуется, и должен возвращаться ненулевой код возврата. Ожидающая команда `restore_command` может быть написана в виде скрипта, в цикле опрашивающего наличие следующего файла WAL. Также должно быть предусмотрено обнаружение сбоя, при котором выполнение команды `restore_command` прерывается, цикл завершается и резервному серверу возвращается ошибка отсутствия файла. Это завершает восстановление, и после этого резервный сервер

начинает функционировать как нормальный.

Псевдокод подходящей команды `restore_command` может быть следующим:

```
triggered = false;
while (!NextWALFileReady() && !triggered)
{
    sleep(100000L);         /* wait for ~0.1 sec */
    if (CheckForExternalTrigger())
        triggered = true;
}
if (!triggered)
        CopyWALFileForRecovery();
```

Работающий пример ожидающей команды `restore_command` предлагается в модуле `pg_standby`. Он может использоваться в качестве примера корректной реализации описанной выше логики. При необходимости он может быть расширен для поддержки особенностей конфигурации или окружения.

Способ запуска восстановления после сбоя является важной частью планирования и разработки. Одним из возможных вариантов является команда `restore_command`. Она выполняется один раз для каждого файла WAL, но процесс, порождаемый этой командой, создается и разрушается для каждого файла, в связи с этим не существует демона или серверного процесса, и возможности использования сигналов и обработчиков. Для запуска восстановления после сбоя требуется более надежное уведомление. Существует возможность использования простого механизма таймаута, особенно в сочетании с известным значением параметра `archive_timeout` на основном сервере. При этом какой-нибудь ошибки, связанной с сетевыми проблемами или загруженностью основного сервера, может быть достаточно для инициирования процесса восстановления. Механизмы нотификации, подобные созданию «включающего» файла, при хорошей реализации менее подвержены подобным ошибкам.

`restore_command`. Опция указывает последнее имя архивного файла, необходимого для корректного запуска восстановления. Это может быть использовано для удаления из архива файлов, которые больше не требуются, если архив доступен резервному серверу для записи.

15.4.1. Реализация

Далее описана короткая процедура конфигурирования резервного сервера. Более детальное описание каждого шага было описано в предыдущем разделе.

1) Установить основной и резервный сервер одинаково, насколько возможно, включая две одинаковых копии PostgreSQL одной версии.

2) Настроить непрерывное архивирование с основного сервера в архив WAL, расположенный в каталоге резервного сервера. Обеспечить соответствующую настройку параметров `archive_mode`, `archive_command` и `archive_timeout` на резервном сервере (см. 14.3.1).

3) Создать базовую резервную копии на основном сервере (см. 14.3.2), и загрузить полученные данные на резервный сервер.

4) Запустить процедуру восстановления на резервном сервере из локального архива WAL, указанием в качестве параметра `restore_command` ожидающей команды, как описано ранее (см. 14.3.3).

Восстановление рассматривает архив WAL как доступный только на чтение, так что WAL-файл после копирования на резервную систему может быть скопирован на ленту одновременно с его чтением резервным сервером. Таким образом, запуск резервного сервера для обеспечения высокой готовности может быть выполнен одновременно с сохранением фалов в долговременном хранилище для целей восстановления после аварии.

Существует возможность с целью тестирования запустить и резервный сервер на одной системе. Но это не несет особого смысла с точки зрения повышения работоспособности сервера и не может расцениваться как решение высокой готовности.

15.4.2. Передача журнала транзакций на уровне записей

Также возможна реализация передачи журнала транзакций на уровне записей с помощью альтернативного метода, хотя это и требует особой разработки, и изменения будут доступны на сервере «горячего» резерва только после передачи файла WAL полностью.

Внешняя программа может использовать функцию `pg_xlogfile_name_offset()` (см. 6.26) для определения имени файла и точного смещения в нем текущего конца журнала WAL. После чего получить доступ к каталогу журнала WAL и осуществить копирование данных на резервный сервер (или серверы) с последнего известного конца журнала WAL до текущего. При таком подходе окно потери данных сокращается до времени цикла работа передающей программы, которое может быть достаточно мало, при этом не расточается пропускная способность на принудительную передачу частично заполненных файлов сегментов для архивирования. Следует отметить, что скрипт `restore_command` резервного сервера продолжает обрабатывать файлы WAL целиком, так что инкрементно скопированные данные обычно не становятся доступны резервным серверам. Они используются только при отказе основного сервера — последний частичный файл WAL передается резервному перед его полным вводом в строй. В связи с этим корректная реализация такого процесса требует взаимодействия скрипта `restore_command` с копирующей программой.

Начиная с версии PostgreSQL 9.0, для реализации похожей функциональности с меньшими затратами может быть использована потоковая репликация (см. 15.2.5).

15.5. Серверы «горячего» резерва

Термин «горячий» резерв используется для описания возможности установки соединения с сервером и выполнения на нем запросов только чтения в процессе его функционирования в режиме восстановления или резервирования. Это удобно как для целей репликации, так и для восстановления резервной копии к требуемому состоянию с высокой точностью. Также этот термин относится к возможности перехода сервера от режима резервирования к нормальному функционированию в условиях выполнения запросов пользователей и/или без разрыва их соединений.

Выполнение запросов в режиме «горячего» резервирования похоже на обычное исполнение запросов, хотя существуют некоторые описываемые далее отличия в использовании и администрировании.

15.5.1. Общее представление для пользователя

При установленном на резервном сервере в `true` параметре `hot_standby`, он начинает принимать соединения, как только восстановление приведет систему в согласованное состояние. Все подобные соединения строго только для чтения; не могут быть использованы даже временные таблицы.

Появление данных на резервном сервере требует некоторого времени для получения их с основного, поэтому между основным и резервным сервером существует измеримая задержка. Один и тот же запрос, почти одновременно запущенный на основном и резервном серверах, может, таким образом, вернуть различный результат. В конечном счете данные на резервном сервере становятся согласованными с основным. Как только запись WAL о завершении транзакции воспроизведена на резервном сервере, внесенные этой транзакцией изменения становятся видны в новых снимках БД на резервном сервере. Снимки БД могут быть сделаны при старте каждого запроса или каждой транзакции, в зависимости от уровня изоляции транзакции.

Транзакции, запущенные в режиме «горячего» резервирования, могут выполнять следующие команды:

- запросы — `SELECT`, `COPY TO`;
- управление курсорами — `DECLARE`, `FETCH`, `CLOSE`;
- управление параметрами — `SHOW`, `SET`, `RESET`;
- управление транзакциями:
    - `BEGIN`, `END`, `ABORT`, `TRANSACTION`.
    - `SAVEPOINT`, `RELEASE`, `ROLLBACK TO SAVEPOINT`.
    - `EXCEPTION` блоки и прочие вложенные транзакции.
- управление блокировками — `LOCK TABLE`, но только для одного из следующих

режимов: ACCESS SHARE, ROW SHARE или ROW EXCLUSIVE;

- планы и ресурсы — PREPARE, EXECUTE, DEALLOCATE, DISCARD.

- модули расширения — LOAD.

Транзакциям, запущенным в режиме «горячего» резервирования, никогда не присваивается идентификатор транзакции (transaction ID) и они не фиксируются в журнале транзакций. Следовательно, следующие действия вызовут сообщение об ошибке:

- команды модификации данных (DML) — INSERT, UPDATE, DELETE, COPY FROM, TRUNCATE. Следует отметить, что недопустимы никакие действия, вызывающие триггера в режиме восстановления. Это ограничение относится даже к временным таблицам, потому что строки таблицы не могут быть прочитаны или записаны без назначения идентификатора транзакции, что невозможно в режиме «горячего» резервирования;

- команды определения структуры данных (DDL) — CREATE, DROP, ALTER, COMMENT. Это ограничение относится даже к временным таблицам, поскольку выполнение перечисленных операций требует внесения изменений в таблицы системного каталога;

- SELECT ... FOR SHARE | UPDATE, т.к. получение блокировок на строках не может быть выполнено без модификации файлов, содержащих их данные;

- правила для запросов SELECT, генерирующие DML команды;

- блокировки LOCK, явно требующие режимов выше ROW EXCLUSIVE MODE;

- блокировки LOCK в краткой форме, т.к. они требуют ACCESS EXCLUSIVE MODE;

- команды управления транзакциями, явно требующие режима записи:

  - BEGIN READ WRITE, START TRANSACTION READ WRITE.

  - SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE, SET TRANSACTION READ WRITE.

  - SET transaction_read_only = off.

- команды управления двухфазными транзакциями — PREPARE TRANSACTION, COMMIT PREPARED, ROLLBACK PREPARED, поскольку даже только читающие транзакции требуют записи в журнал транзакций WAL на фазе подготовки (первой фазе двухфазной транзакции);

- команды изменения значения последовательностей — nextval(), setval();

- LISTEN, UNLISTEN, NOTIFY.

При нормальном функционировании транзакции «только чтения» могут выполнять изменение значений последовательностей и использовать LISTEN, UNLISTEN, NOTIFY, так что сессии в режиме «горячего» резервирования работают в условиях более жестких ограничений, чем обычные сессии только чтения. Возможно в последующих версиях некоторые ограничения могут быть снижены.

В процессе «горячего» резервирования, параметр `transaction_read_only` всегда имеет значение `true` и не может быть изменен. До тех пор пока не выполняется попыток модификации БД, соединения в режиме «горячего» резервирования действуют как любые другие соединения с БД. В случае выполнения процедуры восстановления после сбоя и выхода из резервирования, БД переходит в режим нормального функционирования. Сессии остаются подключенными в ходе смены сервером режима работы. Как только закончен режим «горячего» резервирования, становится возможным создание пишущих транзакций (даже в сессиях, начатых в ходе режима «горячего» резервирования).

Пользователь может узнать, что его сессия только для чтения, с помощью команды `SHOW transaction_read_only`. Дополнительно существует набор функций (см. таблицу 92), позволяющих пользователю получить доступ к информации о резервном сервере. Они дают возможность разрабатывать приложения, учитывающие текущее состояние БД. Эти функции могут быть использованы для мониторинга хода процесса восстановления, или для создания сложных программ, восстанавливающих БД к заданному состоянию.

15.5.2. Разрешение конфликтов запросов

В большинстве случаев основный и резервные серверы слабо связаны. Действия на основном сервере отражаются на резервных. В результате существует возможность отрицательных взаимодействий или конфликтов между ними. Наиболее простым для понимания конфликтом является производительность: при загрузке большого количества данных на основном сервере он генерирует такой же большой поток записей WAL для резервного, в связи с чем запросы к резервному серверу могут бороться за системные ресурсы, такие как устройства ввода/вывода.

Существует еще несколько дополнительных типов конфликтов, которые могу возникнуть в случае применения «горячего» резервирования. Эти конфликты относятся к серьезным, т.к. могут требовать для своего разрешения отмены запроса, и в некоторых случаях даже завершения сессии пользователя. Пользователю предоставляется несколько способов разрешения этих конфликтов. Конфликтные случаи включают в себя:

- явные блокировки на основном сервере, включая явные команды `LOCK` и различные DDL действия, конфликтуют с доступом к таблицам в запросах на резервном сервере;
- удаление табличного пространства на основном сервере конфликтует с запросами на резервном, использующими это табличное пространство для временных рабочих файлов;
- удаление базы данных на основном сервере конфликтует с открытыми к ней сессиями на резервном;
- применение из журнала WAL операции `VACUUM` конфликтует с транзакциями на

резервном сервере, чьи снимки БД все еще видят какие-либо из удаляемых этой операцией строки;

- применение из журнала WAL операции VACUUM конфликтует с запросами на резервном сервере, обращающимися к затрагиваемым этой операцией страницам БД, независимо от видимости удаляемых данных.

На основном сервере эти конфликты выражаются только в задержке; пользователь может выбрать отмену любого из конфликтующих действий. В то же время, на резервном возможности такого выбора нет: переданные через WAL действия выполнены на основном сервере, так что резервный обязан применить их без ошибки. Более того, бесконечная задержка применения WAL может быть весьма нежелательна, поскольку при этом состояние резервного сервера будет все больше отставать от состояния основного. Поэтому предлагаемый механизм принудительно отменяет те запросы к резервному серверу, которые конфликтуют с применяемым записями WAL.

Примером проблемной ситуации служит случай, когда администратор выполняет команду DROP TABLE над таблицей, которая в это время запрашивается на резервном. Очевидно, запрос на резервном сервере не может быть продолжен, если операция DROP TABLE будет применена на резервном сервере. При возникновении такой ситуации на основном сервере, команда DROP TABLE будет ожидать завершения другого запроса. Но когда команда DROP TABLE запущена на основном сервере, он не обладает информацией, какие запросы запущены на резервном, и поэтому не будет ожидать их завершения. Соответствующая запись WAL достигнет резервного сервера во время выполнения на нем запросов, вызвав конфликт. Резервный сервер должен отложить применение этой записи WAL (и, соответственно, следующих за ней тоже) или отменить конфликтующий запрос, чтобы операция DROP TABLE могла быть выполнена.

Если конфликтующий запрос непродолжителен, как правило, желательно позволить ему завершиться путем небольшой задержки применения WAL; в то же время длительная задержка применения WAL обычно нежелательна. В связи с этим, механизм отмены запросов имеет параметры max_standby_archive_delay и max_standby_streaming_delay, которые задают максимально допустимую задержку применения WAL. Конфликтующие запросы будут отменены при превышении их времени выполнения заданных настроек задержки от последних полученных данных WAL. Определены два параметра для задания различной величины задержки, в случае чтения данных WAL из архива (т.е. при восстановлении из базовой резервной копии, или когда резервный сервер должен «догнать» основный в случае большого отставания), и в случае получения записей WAL посредством потоковой репликации.

На резервном сервере, созданном в первую очередь для повышения надежности,

рекомендуется задавать с помощью этих параметров относительно короткие задержки, чтобы он не мог сильно отстать от основного вследствие исполнения запросов. Напротив, если резервный сервер предназначен для выполнения долгих запросов, предпочтительнее могут быть большие или даже бесконечные значения задержек. Следует учитывать, что выполнение длинных запросов, если они задерживают применение WAL, может вызвать ситуацию, при которой другие сессии на резервном сервере не смогут видеть недавних изменений на основном.

Как только задержка, заданная параметрами `max_standby_archive_delay` или `max_standby_streaming_delay`, будет достигнута, конфликтующие запросы будут отменены. Это обычно приводит только к отмене запроса, хотя в случае применения `DROP DATABASE`, конфликтующая сессия будет полностью завершена. Кроме того, если конфликт вызван блокировкой, удерживаемой ожидающей транзакцией, конфликтующая сессия завершается (это поведение может измениться в будущем).

Отмененные запросы могут быть сразу повторены (после начала новой транзакции). Так как отмена запроса зависит от характера воспроизводимой записи WAL, отмененный запрос при повторном вызове может быть выполнен успешно.

Следует учитывать, что значения параметров задержки сравниваются со временем, прошедшим с момента получения резервным сервером последних данных WAL. Таким образом, допустимый период ожидания для каждого отдельного запроса на резервном сервере никогда не превышает значения параметра задержки, и может быть значительно меньше, если резервный сервер уже отстает в результате ожидания завершения предыдущих запросов, или в результате отставания из-за невозможности успевать за основным вследствие большой нагрузки.

Наиболее распространенной причиной для конфликта между запросами на резервном сервере и воспроизведением WAL является «ранняя очистка». Обычно PostgreSQL позволяет удаление старых версий записей, когда отсутствуют транзакции, которые должны видеть их для обеспечения корректной видимости данных в соответствии с правилами MVCC. Однако это правило может быть применено только для транзакций, выполняющихся на главном сервере. Так что не исключено, что очистка на главном сервере удалит версии записей, которые все еще видны в транзакциях на резервном.

Опытные пользователи должны отметить, что и очистка и «замораживание» версий записей одинаково могут потенциально конфликтовать с запросами на резервном сервере. Ручной запуск команды `VACUUM FREEZE` может вызвать конфликты даже в таблицах, не содержащих обновленных или удаленных записей.

Пользователю должно быть ясно, что таблицы, которые регулярно и сильно обновляются на основном сервере, быстро вызывают отмену долго работа-

ющих запросов на резервном. В таких случаях установку конечных значений max_standby_archive_delay или max_standby_streaming_delay можно считать аналогичной настройке statement_timeout.

Существуют возможности решения проблемы в случае, когда когда большое количество отмен запросов на резервном сервере неприемлемо. Первым вариантом служит установка параметра hot_standby_feedback, который предотвращает очистку недавно удаленных версий записей операцией VACUUM, что устраняет конфликты из-за очистки. При этом возникает задержка очистки «мертвых» версий записей на основном сервере, что может вызвать нежелаемый рост размера таблиц. Однако, ситуация с очисткой не будет хуже, чем если запросы шли прямо на основной сервер, и вы также получали преимущества исполнения на резервном сервере. max_standby_archive_delay должно быть большим в этом случае, т.к. задерживаемые WAL файлы могут уже содержать записи, которые будут конфликтовать с резервными запросами.

Другой вариант заключается в увеличении vacuum_defer_cleanup_age на основном сервере, так что мертвые записи не будут очищены так быстро, как это делается обычно. Это дает больше времени для выполнения запросов, прежде чем они будут отменены на резервном сервере, без необходимости установки большого значения max_standby_streaming_delay. Однако при таком подходе трудно гарантировать какие-либо конкретные временные окна выполнения, т.к. vacuum_defer_cleanup_age измеряется в транзакциях, совершенных на основном сервере.

Количество отмененных запросов и причины их отмены могут быть просмотрены с помощью системного представления pg_stat_database_conflicts на резервном сервере. Системное представление pg_stat_database содержит краткую информацию.

15.5.3. Общее представление для администратора

Если в postgresql.conf включен параметр hot_standby и существует файл recovery.conf, сервер запускается в режиме «горячего» резервирования. При этом требуется некоторое время до возможности установки с ним соединения, поскольку сервер не принимает соединения до тех пор, пока он не выполнит восстановление до согласованного состояния, в котором могут быть выполнены запросы. В течение этого периода клиентам, которые пытаются подключиться, будет отказано с сообщением об ошибке. Для того, чтобы определить, что сервер достиг согласованного состояния, необходимо либо продолжать попытки установки соединения, либо ожидать следующих сообщений в логе сервера:

```
LOG:  entering standby mode
... then some time later ...
LOG:  consistent recovery state reached
LOG:  database system is ready to accept read only connections
```

Информация о целостности (согласованности) записывается один раз для каждой контрольной точки на основном сервере. Не существует возможности включить режим «горячего» резервирования в процессе чтения журнала WAL, записанного в период времени, когда на основном сервере параметр `wal_level` не был установлен значение `hot_standby` и `logical`. Достижение согласованного состояния также может быть задержано при наличии следующих двух условий:

- пишущая транзакция имеет более 64 вложенных транзакций;
- существует долгоживущая пишущая транзакция.

ВНИМАНИЕ! Не рекомендуется использование логического декодирования, т.к. в данном режиме изменения отправляются в виде высокоуровневых запросов, что может стать причиной утечки информации.

Если запущен процесс передачи журнала транзакций с помощью файлов («теплый» режим резервирования), возможно потребуется ожидать до поступления следующего файла WAL, что может произойти вплоть до указанного в параметре `archive_timeout` времени.

Некоторые параметры на резервном сервере требуют изменения в процессе перевода его в основный. Для этих параметров значения на резервном сервере должны быть равными или большими чем на основном. Если эти параметры не будут установлены в достаточно большое значение, резервный сервер может не запуститься. В этом случае значения должны быть увеличены, после чего сервер должен быть перезапущен для повтора попытки восстановления. К этим параметрам относятся:

- `max_connections`
- `max_prepared_transactions`
- `max_locks_per_transaction`

Важно, чтобы администратор выбрал соответствующие настройки для `max_standby_archive_delay` и `max_standby_streaming_delay`. Выбор может варьироваться в зависимости от бизнес-приоритетов. Например, если сервер в первую очередь позиционируется как сервер высокой доступности, то требуются низкие настройки задержки, возможно, даже нулевые, хотя это очень экстремальная конфигурация. Если резервный сервер позиционируется в качестве дополнительного сервера для запросов поддержки принятия решений, то приемлемым вариантом может быть установка максимальных значений задержки до большого количества часов, или даже -1, что означает бесконечное ожидание завершения запросов.

Записанный на основном сервере статус транзакции «hint bits» не фиксируется в журнале WAL, т.к. данные на резервном сервере, скорее всего, снова перепишут его. Таким образом, резервный сервер будет по-прежнему выполнять запись на диск, даже если все пользователи работают в режиме только чтения; никаких изменений с самими значениями

данных не происходит. Пользователи будут продолжать писать временные файлы больших сортировок и обновлять информационные файлы кэша таблиц, так что ни одна из частей базы данных не является в действительности только для чтения в режиме горячего резерва. Отметим также, что запись к удаленным базам данных с использованием модуля dblink и другие операции вне базы данных с использованием PL-функции будут по-прежнему доступны, даже если транзакция является локально только для чтения.

Следующие типы команд администрирования не допускаются в процессе восстановления:

- команды языка определения данных (DDL) — например, CREATE INDEX;
- управление привилегиями и правом владения — GRANT, REVOKE, REASSIGN;
- команды обслуживания — ANALYZE, VACUUM, CLUSTER, REINDEX.

Следует обратить внимание, что некоторые из этих команд фактически допускаются в транзакциях «только для чтения» на основном сервере.

В результате отсутствует возможность создания дополнительных индексов или статистики, которые существуют исключительно на резервном сервере. Если эти команды администрирования необходимы, они должны быть выполнены на основном сервере, и в конечном итоге эти изменения будут распространены на резервный.

Функции pg_cancel_backend() и pg_terminate_backend() будут работать на пользовательском уровне, но не в стартовом процессе, который выполняет восстановление. pg_stat_activity не показывает запись для стартового процесса, а также не отображает транзакции восстановления как активные. В результате pg_prepared_xacts всегда пусто во время восстановления. Для решения проблем с некорректно подготовленными транзакциями, следует использовать pg_prepared_xacts на основном сервере, и на нем же выполнять команды для их разрешения.

Представление pg_locks отображает блокировки пользовательских процессов. Также pg_locks отображает виртуальную транзакцию стартового процесса, которая владеет всеми явными блокировками AccessExclusiveLock, удерживаемыми транзакциями в ходе восстановления. Следует отметить, что стартовый процесс не захватывает блокировок для изменений в БД, следовательно, отличные от AccessExclusiveLock блокировки для него не отображаются в pg_locks; подразумевается, что они просто есть.

Nagios плагин check_pgsql работает, поскольку существует проверяемая им простая информация. Скрипт проверки check_postgres также работает, хотя некоторые выводимые значения могут давать разные или путанные результаты. К примеру, время последней операции VACUUM не поддерживается, т.к. на резервном сервере она не выполняется. Исполняемые на основном сервере операции VACUUM пересылают свои изменения на резервный.

В процессе восстановления не работают файловые команды управления журналом

WAL (например, pg_start_backup, pg_switch_xlog и т.п.).

Динамически загружаемые модули работают, включая pg_stat_statements.

Прикладные блокировки при восстановлении работают нормально, включая механизм обнаружения взаимных блокировок. Следует отметить, что прикладные блокировки никогда не фиксируются в журнале транзакций WAL, вследствие чего прикладные блокировки на основном и резервном серверах не конфликтуют с восстанавливаемым журналом WAL. Также возможно получение прикладной блокировки на основном сервере и получение той же блокировки на резервном. Прикладные блокировки относятся только к тому серверу, на котором они были получены.

Системы репликации с помощью триггеров, такие как Slony, Londiste и Bucardo вообще не могут быть запущены на резервном сервере, хотя до тех пор пока изменения не переданы для применения на резервный сервер, они могут успешно функционировать на основном. Воспроизведение журнала транзакций не использует триггеры, поэтому невозможно применение систем, требующих дополнительной записи в БД или использования триггеров.

Новые идентификаторы OID не могут быть назначены, хотя UUID генераторы могут функционировать до тех пор, пока не полагаются на запись статуса в БД.

В настоящее время создание временных таблиц не допускается в транзакция только для чтения, поэтому в некоторых случаях существующие скрипты могут выполнятся некорректно. Эти ограничения могут быть смягчены в последующих версиях. Это как технических вопрос, так и вопрос совместимости с SQL-стандартом.

Команда DROP TABLESPACE может быть выполнена успешно только для пустых табличных пространств. Некоторые пользователи резервных серверов активно используют табличные пространства с помощью параметра temp_tablespaces. В случае наличия временных файлов в табличном пространстве, все активные запросы завершаются для гарантированного удаления временных файлов, чтобы табличное пространство могло быть удалено, и восстановление журнала WAL могло быть продолжено.

Выполнение DROP DATABASE или ALTER DATABASE ... SET TABLESPACE на основном сервере создает элемент WAL, вызывающий при воспроизведении на резервном принудительное завершение соединений пользователей с указанной БД. Эти действия выполняются незамедлительно, не зависимо от параметра max_standby_streaming_delay. Следует отметить, что ALTER DATABASE ... RENAME не приводит к отключению пользователей, которые в большинстве случаев не получают оповещения, но может в некоторых случаях приводить к сбою в программах, зависящих тем или иным образом от имени БД.

В нормальном режиме (не в режиме восстановления) при выполнении DROP USER или DROP ROLE для роли с привилегией установки соединения, в случае существования соединения от ее имени, с подключенным пользователем ничего не происходит — он

остается подключенным. В то же время пользователь лишается возможности переустановить соединение. Это поведение применяется и на резервном сервере, так что DROP USER на основном сервере не приводит к отключения этого пользователя на резервном.

Сборщик статистики активен в ходе восстановления. Все операции сканирования, чтения, блокировки, использования индексов и т.п. будут нормально записаны на резервном сервере. Воспроизводимые действия не повторяют своего действия на основном, таким образом, воспроизведение вставки не увеличит значение столбца Inserts представления pg_stat_user_tables. Файл статистики перед выполнением восстановления удаляется, так что статистика собранная на основном и резервном серверах будет различаться; это рассматривается как особенность, а не ошибка.

Автовакууминг в процессе восстановления не активен. Он будет запущен штатным образом после завершения восстановления.

Процесс фоновой записи активен в ходе восстановления и создает точки перезапуска (схожие с контрольными точками на основном сервере) и блокировку действий по очистке. Это может включать «hint bit» информацию, сохраняемую на резервном сервере. Команда CHECKPOINT допустима в ходе восстановления, хотя она создает точку перезапуска, а не новую контрольную точку.

15.5.4. Параметры горячего резервирования

Различные параметры были упомянуты выше в разделах 15.5.2 и 15.5.3.

На основном сервере могут быть использованы параметры wal_level и vacuum_defer_cleanup_age. max_standby_archive_delay и max_standby_streaming_delay на основном сервере не действуют.

На резервном сервере могут быть использованы параметры hot_standby, max_standby_archive_delay и max_standby_streaming_delay. vacuum_defer_cleanup_age не действует пока сервер находится в режиме резервирования, хотя будет действовать, если резервный сервер станет основным.

15.5.5. Предостережения

В режиме горячего резервирования существуют некоторые ограничения. Они могут и возможно будут устранены в будущих версиях:

1) Операции с хэш-индексами в настоящее время не фиксируются в журнале транзакций WAL, поэтому его восстановление подобные индексы не обновляет.

2) Для создания снимка БД требуется полная информация о запущенных транзакциях. Транзакции, имеющие большое количество вложенных транзакций (в текущий момент более 64-х) будут задерживать установку соединений только чтения до завершения наиболее продолжительной пишущей транзакции. При возникновении

данной ситуации в лог сервера будут отправлены пояснительные сообщения.

3) Допустимые точки старта запросов на резервном сервере формируются в каждой контрольной точке на основном. Если резервный сервер останавливается, когда основной находится в состоянии остановки, может быть невозможен переход в состояние горячего резервирования до запуска основного сервера, который сформирует дополнительную стартовую точку в журнале WAL. Эта ситуация не является проблемой в наиболее распространенных ситуациях. Как правило, если основный сервер не работает и больше не доступен, что происходит, скорее всего, из-за серьезного сбоя, итак требуется преобразование резервного сервера в новый основный. И в ситуации, когда основный сервер намеренно остановлен, проверка того, что резервный сервер плавно становится новым основным, также является стандартной процедурой.

4) В конце восстановления, блокировки `AccessExclusiveLock`, удерживаемые подготовленными транзакциями, требуют вдвое большее число записей о блокировках. При планировании запуска либо большого количества одновременных подготовленных транзакций, захватывающих блокировки типа `AccessExclusiveLock`, либо одной большой транзакции, которая захватывает большое количество блокировок типа `AccessExclusiveLock`, рекомендуется выбирать большое значение параметра `max_locks_per_transaction`, как минимум в два раза большее чем значение этого параметра на основном сервере. Выше сказанное не имеет значения в случае установки `max_locks_per_transaction` в значение `0`.

5) Уровень изоляции транзакция `Serializable` недоступен на сервере горячего резерва. При попытке задания транзакции уровня изоляции `Serializable` в режиме горячего резервирования будет выдана ошибка.

16. НАСТРОЙКА ПАРАМЕТРОВ ВОССТАНОВЛЕНИЯ

В разделе описываются настройки, доступные в файле recovery.conf. Они применяются только в процессе восстановления. Установки должны быть сброшены для последующих операций восстановления, и они не могут быть изменены после запуска процесса восстановления.

Настройки в файле recovery.conf задаются в формате имя = 'значение' по одному в каждой строке. Знак (#) обозначает остаток строки как комментарий. Для вставки в значение параметра одинарной кавычки, она должна быть удвоена ('').

В состав PostgreSQL входит файл примера recovery.conf.sample.

16.1. Параметры архивирования

Параметры:

- restore_command (строка) — определяет команду локальной оболочки, выполняемую для извлечения архивированных сегментов из серии WAL-файлов. Параметр является обязательным для непрерывного архивирования и необязательным для потоковой репликации. Каждое появление %f заменяется именем файла журнала, а %p заменяется путем, в который осуществляется копирование файла журнала. (Путь копирования указывается относительно текущего рабочего каталога, т.е. каталога данных кластера.) Каждое появление %r заменяется именем файла, содержащего последнюю верную точку перезапуска. Это наиболее ранний файл, который должен быть сохранен для возможности повторного восстановления, так что эта информация может быть использована для усечения архива до минимального требуемого для поддержки перезапуска с момента текущего восстановления. %r обычно используется только в конфигурации «теплого» резерва (см. 15.2). При необходимости вставить в команду символ %, следует писать %%.

  Важно, чтобы команда возвращала нулевой код возврата только в случае успешного выполнения. У команды будет запрошен несуществующий в архиве файл, в этом случае она должна вернуть ненулевой результат.

  ```
  restore_command = 'cp /mnt/server/archivedir/%f "%p"'
  ```

  Исключением является ситуация, если команда была завершена по сигналу (чаще всего с SIGTERM, который используется при выключении сервера баз данных) или ошибкой командной оболочки (команда не найдена). В этих случаях восстановление прекращается и сервер не будет запущен.

- archive_cleanup_command (string) — необязательный параметр, задающий команду оболочки, которая будет выполняться по каждой точке перезапуска. Назначением archive_cleanup_command является обеспечение механизма очистки

старых архивный файлов журнала WAL, которые больше не требуются серверу «горячего» резерва. Каждое появление %r заменяется именем файла, содержащего последнюю верную точку перезапуска. Это наиболее ранний файл, который должен быть сохранен для возможности повторного восстановления, и следовательно все файлы, более ранние чем %r, могут быть безопасно удалены. Эта информация может быть использована для усечения архива до минимального размера, требуемого для поддержки перезапуска с момента текущего восстановления. Модуль pg_archivecleanup часто используется в archive_cleanup_command для конфигурации «горячего» резерва, состоящей из одного сервера, например:

```
archive_cleanup_command = 'pg_archivecleanup /mnt/server/archivedir %r'
```

Примечание. Следует учитывать, что если несколько резервных серверов восстанавливаются из одного архивного каталога, необходимо гарантировать существование файлов WAL до тех пор, пока они требуются каким-либо из серверов. archive_cleanup_command обычно используется в конфигурации «теплого» резерва (см. 15.2). При необходимости вставить в команду символ %, следует писать %%.

Если команда возвращает ненулевой код возврата, то в журнал записывается предупреждение (WARNING).

- recovery_end_command (string) — определят команду оболочки, которая должна быть выполнена один раз после завершения восстановления. Параметр не является обязательным. Назначением recovery_end_command является обеспечение механизма очистки после репликации или восстановления. Каждое %r заменяется именем файла, содержащего последнюю верную точку перезапуска, как и в archive_cleanup_command.

Если команда возвращает ненулевой код возврата, в журнал записывается предупреждение (WARNING), и осуществляется запуск нормальной работы БД. Исключением является прерывание команды сигналом, в этом случае запуск нормальной работы БД не производится. Исключением является ситуация, когда команда была завершена по сигналу или с ошибкой командной оболочки (команда не найдена). В таком случае будет выведена фатальная ошибка.

16.2. Параметры цели восстановления

По умолчанию, восстановление будет производиться до конца журнала WAL. Следующие параметры могут быть использованы для определения ранней точки завершения. Один из recovery_target, recovery_target_name, recovery_target_time или recovery_target_xid может быть использован; если более одного указано в конфигура-

ционном файле, последняя значение будет использоваться.

Параметры:

- recovery_target = 'immediate' — определяет, что восстановление должно завершиться как только, как достигнуто определенное состояние, т.е. как можно раньше. При восстановлении из резервной копии, это означает точку, где кончается копия.

Технически, это строковый параметр, но только immediate в настоящее время является разрешенным значением.

Примечание. Данный параметр используется только в СУБД версии 9.6.

- recovery_target_name (string) — задает имя точки восстановления, созданной с помощью pg_create_restore_point(), до которой должно быть выполнено восстановление.

- recovery_target_time (timestamp) — задает время, в течение которого должно быть выполнено восстановление. Точная точка завершения также зависит от recovery_target_inclusive.

- recovery_target_xid (string) — Этот параметр определяет идентификатор транзакции, до которой восстановление должно быть выполнено. Имейте в виду, что в то время как идентификаторы транзакций присваиваются последовательно в начале транзакции, транзакции могут завершиться в другом числовом порядке. Восстановленными операциями будут являться те, которые были совершены до этого номера транзации (и, возможно, включая этот номер транзакции) указано одно. Точная точка завершения также зависит от recovery_target_inclusive.

- recovery_target_time (timestamp) — задает метку времени, до которой должно быть произведено восстановление. Одновременно может быть определен только один из параметров recovery_target_time, recovery_target_name или recovery_target_xid. По умолчанию восстановление осуществляется до конца журнала WAL. Точное место завершения также определяется с помощью параметра recovery_target_inclusive.

- recovery_target_xid (string) — определяет идентификатор транзакции ID, до которой должно быть произведено восстановление. Необходимо иметь ввиду, что, несмотря на то, что ID назначается последовательно в момент старта транзакции, транзакции могут завершатся в другом номерном порядке. Будут восстановлены те транзакции, которые были завершены ранее (и возможно включая) указанной. Одновременно может быть определен только один из параметров recovery_target_xid, recovery_target_name или recovery_target_time. По умолчанию восстановление осуществляется до достижения конца журна-

ла WAL. Точное место завершения также определяется с помощью параметра recovery_target_inclusive.

Следующие опции дополнительно указывают цель восстановления и влияют на то, что происходит, когда цель достигнута:

- recovery_target_inclusive (boolean) — определяет, как будет осуществляться остановка, сразу после заданной цели восстановления (true), или непосредственно перед ней (false). Применяется к обоим параметрам recovery_target_time или recovery_target_xid, какой бы ни был задан для конкретного восстановления. Указывает будут ли транзакции, имеющие точно совпадающие с указанным временем завершения или ID, соответственно, включены в восстановление. По умолчанию true.
- recovery_target_timeline (string) — определяет восстановление в конкретную линию времени. По умолчанию восстановление осуществляется в ту линию времени, которая была на момент создания резервной копии. Установка этого параметра в значение latest осуществляет восстановление к последней линии времени, найденной в архиве, что удобно для резервного сервера. Кроме того, необходимость установки параметра может потребоваться только в сложных ситуациях повторного восстановления, когда требуется возврат к состоянию, достигнутому после восстановления на указанный момент времени (point-in-time recovery). Дополнительная информация приведена в 14.3.4.
- pause_at_recovery_target (boolean) — задает необходимость приостановки восстановления по достижению цели восстановления. По умолчанию — true. Это сделано для того, чтобы обеспечить возможность проверки с помощью запросов, достигнута ли желаемая точка восстановления. Процесс восстановления может быть продолжен с помощью pg_xlog_replay_resume() (см. 93), обеспечивающего выполнение восстановления до конца. Если достигнутая цель восстановления не является желаемой, необходимо остановить сервер, изменить настройки восстановления с указанием более поздней цели и запустить сервер для продолжения восстановления.

  Этот параметр не действует, если не включен hot_standby, или не указана цель восстановления.

16.3. Параметры резервного сервера

Параметры:

- standby_mode (boolean) — задает запуск сервера в качестве резервного. При установке значения on, сервер не остановит процесс восстановления по достижению

конца архивного журнала WAL, а продолжит попытки восстановления выбором новых сегментов WAL с помощью restore_command и/или с помощью соединения с ведущим сервером в соответствии с primary_conninfo.

- primary_conninfo (string) — задает строку соединения резервного сервера с ведущим. Строка использует стандартный формат клиентской библиотеки libpq. В случае отсутствия с строке какой-нибудь опции ее значение берется из соответствующей переменной окружения. Если требуемая переменная окружения не установлена, берется значение по умолчанию.

Строка соединения должна содержать имя или адрес ведущего сервера, порт для установки соединения, если он отличается от заданного по умолчанию. Также должно быть указано имя пользователя, соответствующего роли ведущего сервера, имеющей необходимые привилегии (см. 15.2.5.1). Если ведущий сервер требует парольную аутентификацию, должен быть задан пароль. Он может быть предоставлен как в строке primary_conninfo, так и в отдельном файле пароля ~/.pgpass на резервном сервере (в качестве имени БД используется replication). Имя БД в primary_conninfo указываться не должно.

Этот параметр не действует при значении off параметра standby_mode.

- primary_slot_name (string) — опционально указывает имя существующего слота репликации, который используется при подключении к основному серверу по протоколу потоковой репликации для управления удаляемыми ресурсами на вышестоящем узле (см. 15.2.6). Этот параметр не имеет смысла, если параметр primary_conninfo не установлен.

Примечание. Данный параметр используется только в СУБД версии 9.6.

- trigger_file (string) — задает имя триггерного файла, появление которого прекращает восстановление на резервном сервере. Даже если это значение не указано, существует возможность изменить состояние резервного сервера с помощью pg_ctl. Этот параметр не действует при значении off параметра standby_mode.

- recovery_min_apply_delay (integer)

По умолчанию резервный сервер восстанавливает WAL записей основного насколько это возможно. Полезно иметь задержку при копировании данных, предлагая возможности для исправления ошибок потери данных. Этот параметр позволяет отложить восстановление на период времени, который измеряется в миллисекундах, если ни одна единица измерения не указана. Например, если вы установите этот параметр в 5min, то резервный сервер будет проигрывать каждую транзакцию только тогда, когда системное время на 5 минут позднее времени транзакции основного сервера. Также возможно, что задержка репликации между серверами превышает значение

этого параметра, и в этом случае не будет задержки. Следует отметить, что задержка рассчитывается между меткой времени WAL и текущего времени на режим ожидания. Задержки при передаче по сети или каскадной репликации могут привести к значительному снижению фактического времени ожидания. Если системные часы основного сервера и резервного не синхронизированы, это может привести к применению записей WAL раньше, чем ожидалось; но это не главная проблема, потому что полезные настройки этого параметра значительно больше, чем типичное время отклонения между серверами.

Задержка только связана с записями WAL применения транзакций. Другие записи проигрываются настолько быстро, как это возможно, что не является проблемой, потому что правила видимости MVCC гарантируют, что их последствия не видны до тех пор, соответствующая запись не применится.

Задержка происходит до тех пор, пока резервный сервер не будет активен или или не отработал. После этого резервный сервер закончит восстановление без дальнейшего ожидания.

Этот параметр предназначен для использования с потоковой репликацией. Однако, если он задан, он будет принят во всех случаях. Синхронная репликация не влияет на этот параметр, потому что еще не любой параметр для синхронного запроса применяется в транзакции. Параметр `hot_standby_feedback` будет откладывать использование этой особенности, которая может привести к раздуванию WAL файлов на основном сервере. Рекомендуется использовать оба параметра вместе с осторожностью.

П р и м е ч а н и е. Данный параметр используется только в СУБД версии 9.6.

17. МОНИТОРИНГ ИСПОЛЬЗОВАНИЯ ДИСКА

Данный раздел содержит описание отслеживания использование диска СУБД PostgreSQL.

17.1. Определение использования диска

Каждая таблица имеет первичный дисковый файл, в котором хранится большая часть данных. Если таблица имеет какие-либо столбцы с потенциально большими по длине значениями, то также может существовать TOAST-файл, ассоциированный с этой таблицей, который используется для хранения значений, которые слишком длинны, чтобы размещаться в главной таблице. На каждую TOAST-таблицу, если она существует, существует один индекс. Также там могут быть и индексы, ассоциированные с базовой таблицей. Каждая таблица и индекс хранятся в отдельном дисковом файле — возможно более чем в одном файле, если размер этого файла превышает 1 ГБ.

Определение используемого дискового пространства может быть выполнено тремя способами: используя SQL-функции, перечисленные в таблице 96, используя модуль `oid2name`, или, используя ручной обзор системных каталогов. Вышеупомянутые SQL-функции являются наиболее простыми и рекомендованными для использования.

Используя `psql` после недавнего применения команд `VACUUM` или `ANALYZE`, можно выполнить следующий запрос для определения используемого дискового пространства какой-либо таблицей:

```
SELECT pg_relation_filepath(oid), relpages
 FROM pg_class WHERE relname = 'customer';

 pg_relation_filepath | relpages
----------------------+----------
 base/16384/16806     |       60
(1 row)
```

Каждая страница обычно занимает 8 КБ. (поле `relpages` обновляется только командами `VACUUM`, `ANALYZE`, и несколькими DDL-командами, такими как `CREATE INDEX`). Путь к файлу представляет интерес, если требуется проанализировать непосредственно файл на диске.

Для отображения пространства, используемого TOAST-таблицами, применяется следующий запрос:

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid
      FROM pg_class
```

```
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
      oid = (SELECT indexrelid
             FROM pg_index
             WHERE indrelid = ss.reltoastrelid)
ORDER BY relname;

       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

You can easily display index sizes, too:

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;

       relname        | relpages
----------------------+----------
 customer_id_indexdex |       26
```

Размеры индексов могут быть получены следующим образом:

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;

       relname        | relpages
----------------------+----------
 customer_id_indexdex |       26
```

Также легко определить таблицы и индексы, занимающие наибольшее место:

```
SELECT relname, relpages
FROM pg_class
```

```
ORDER BY relpages DESC;

       relname         | relpages
-----------------------+----------
 bigtable              |     3290
 customer              |     3144
```

17.2. Ошибка переполнения диска

Наиболее важной задачей мониторинга диска для администратора баз данных состоит в том, чтобы быть уверенным в наличии свободного места на диске. Заполненность диска не приведет к повреждению данных, но может остановить нормальное функционирование. Если диск, содержащий WAL-файлы, заполнится, сервер СУБД может аварийно завершить свою работу.

Если отсутствует возможность освободить дополнительное пространство на диске, удалив какие-либо ненужные файлы, следует перенести часть файлов базы данных на другие ФС с помощью создания табличных пространств. Подробности об этом приведены в 11.6.

Примечание. Некоторые ФС плохо работают, когда они почти или совсем заполнены, поэтому не следует ждать пока диск заполнится полностью для выполнения необходимых действий.

Если система поддерживает дисковые квоты для пользователей, следует установить достаточную квоту, выделенную для пользователя, из под которого запускается сервер СУБД. Исчерпание квоты вызовет такие же последствия как и отсутствие непосредственно места на диске.

18. КЛИЕНТСКИЕ ПРИЛОЖЕНИЯ POSTGRESQL

В разделе содержится справочная информация о клиентских приложениях и утилитах PostgreSQL. Не все из перечисленных команд являются общедоступными утилитами, некоторые могут требовать специальных привилегий. Общей чертой этих приложений является то, что они могут быть запущены на любом хосте не зависимо от расположения сервера БД.

Для работы с СУБД PostgreSQL предусмотрен ряд утилит командной строки, программные интерфейсы для создания клиентских приложений и серверных расширений. Также предоставляется утилита администрирования с визуальным пользовательским интерфейсом.

18.1. Аргументы командной строки для установки соединения

При использовании клиентских утилит командной строки задание свойств соединения осуществляется с помощью аргументов (опций) командной строки, приведенных в таблице 110.

Таблица 110

Аргумент	Описание
`-h host` `--host=hoste`	Указывает имя сервера БД или каталог сокетов UNIX, если начинается с символа «/».
`-p port` `--port=port`	Указывает порт сервера БД или расширение имени сокета UNIX, по которым сервер принимает соединения.
`-U username` `--username=username`	Указывает имя пользователя для установки соединения.
`-w, --no-password`	Подавление запроса пароля пользователя. В случае, когда установка соединения с сервером требует ввода пароля, а пароль недоступен, например из файла `.pgpass`, попытка установки соединения завершается ошибкой. Опция полезна при выполнении пакетов заданий или скриптов, в процессе обработки которых отсутствует пользователь, который может вводить пароль.
`-W, --password`	Принудительный запрос пароля при установке соединения. Опция не является существенной, т. к. утилита по умолчанию всегда запрашивает пароль в случае, когда сервер требует ввода пароля при установке соединения. В тоже время для определения необходимости ввода пароля утилита делает дополнительный запрос к серверу, которого можно избежать, указав эту опцию.

При отсутствии перечисленных аргументов используются переменные окружения (`PGDATABASE`, `PGHOST`, `PGPORT`, `PGUSER`), определяющие параметры соединения по умолчанию.

Информацию о версии и способе вызова утилит и допустимых аргументов можно

получить с помощью аргументов:

`--help` — показать справку по вызову команды;

`--version` — показать версию.

18.2. `clusterdb` - кластеризация таблицы или БД

Для кластеризации (оптимизации индексов) ранее кластеризованных таблиц в БД используется утилита `clusterdb`.

Она находит таблицы, которые были ранее кластеризованы, и кластеризует их заново по тем же индексам, которые были указаны до этого. Таблицы, которые до этого не были кластеризованы, не затрагиваются.

Утилита является оболочкой для вызова SQL-команды `CLUSTER`, при этом нет разницы в производительности при использовании утилиты или при ином доступе к серверу.

Синтаксис:

```
clusterdb [option...] [ --table | -t table ] ... [dbname]
clusterdb [option...] --all | -a
```

Аргументы командной строки приведены в таблице 111.

Таблица 111

Аргумент	Описание
`dbname`	Указывает имя используемой БД. По умолчанию в качестве имени БД используется имя пользователя, заданное при установке соединения.
`-a` `--all`	Кластеризовать все БД.
`[-d] dbname` `[--dbname=] dbname`	Указывает имя БД для кластеризации. Если аргумент не указан, и не указаны аргументы (`-a, --all`), имя БД берется из переменной окружения `PGDATABASE`. Если она не установлена, в качестве имени БД используется имя пользователя, заданное при установке соединения.
`-e` `--echo`	Показывать команды, которые формируются и отправляются серверу.
`-q` `--quiet`	Не выводить сообщений в процессе работы.
`-t table` `--table=table`	Кластеризовать только указанную таблицу.
`-v` `--verbose`	Выводить дополнительную информацию в процессе работы.
`--maintenance-db=dbname`	Задает имя БД, к которой необходимо подключится для определения баз данных, которые должны быть кластеризованы. Если не указано, используется `postgres`, а если она отсутствует, то `template1`.

Утилита также принимает аргументы командной строки, используемые для установки соединения (см. 18.1).

Примеры:

1. Кластеризации БД `test`:

```
$ clusterdb test
```

2. Кластеризация одной таблицы `foo` в БД `xyzzy`:

```
$ clusterdb --table foo xyzzy
```

18.3. `createdb` - создание БД

Для создания новой БД используется утилита `createdb`.

По умолчанию владельцем новой БД становится пользователь, выполняющий команду. В то же время в качестве владельца новой БД может быть указан другой пользователь с помощью опции `-O`, если выполняющий команду пользователь обладает соответствующими привилегиями.

Утилита является оболочкой для вызова SQL-команды `CREATE DATABASE`, при этом нет разницы в производительности при использовании утилиты или при ином доступе к серверу.

Синтаксис:

```
createdb [option...] [dbname [description]]
```

Аргументы командной строки приведены в таблице 112.

Таблица 112

Аргумент	Описание
`dbname`	Указывает имя создаваемой БД. Имя должно быть уникальным среди БД одного кластера. По умолчанию БД создается с таким же именем, как и текущий системный пользователь.
`description`	Указывает комментарий к создаваемой БД.
`-D tablespace` `--tablespace=tablespace`	Указывает табличное пространство по умолчанию для создаваемой БД.
`-e` `--echo`	Показывать команды, которые формируются и отправляются серверу.
`-E encoding` `--encoding=encoding`	Указывает кодировку символов для создаваемой БД.
`-l locale` `--locale=locale`	Указывает локаль для создаваемой БД.
`--lc-collate=locale`	Указывает `LC_COLLATE` для БД.
`--lc-ctype=locale`	Указывает `LC_CTYPE` для БД.
`-O owner` `--owner=owner`	Указывает пользователя, который будет владельцем создаваемой БД.

Окончание таблицы 112

Аргумент	Описание
`-T template` `--template=template`	Указывает имя БД, используемой как шаблон.

Опции `-D`, `-E`, `-l`, `-O` и `-T` соответствуют опциям SQL-команды `CREATE DATABASE`.

Утилита также принимает аргументы командной строки, используемые для установки соединения (см. 18.1).

Примеры:

1. Создание БД `demo` на текущем сервере:

```
$ createdb demo
```

2. Создание БД `demo` на сервере `eden`, порт 5000 с кодировкой `LATIN1`:

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE demo ENCODING 'LATIN1';
```

18.4. `createlang` - установка поддержки процедурного языка в базу данных

Для установки поддержки процедурного языка в БД используется утилита `createlang`.

Утилита является оболочкой для вызова SQL-команды `CREATE LANGUAGE`, при этом нет разницы в производительности при использовании утилиты или при ином доступе к серверу.

ВНИМАНИЕ! `createlang` является устаревшей утилитой и будет устранена в будущих версиях PostgreSQL. Вместо нее рекомендуется использовать команду `CREATE EXTENSION`.

Синтаксис:

```
createlang [option...] langname [dbname]
createlang [option...] --list | -l [dbname]
```

Аргументы командной строки приведены в таблице 113.

Таблица 113

Аргумент	Описание
`langname`	Указывает устанавливаемый процедурный язык.
`dbname`	Указывает имя БД, в которую устанавливается поддержка процедурного языка. По умолчанию в качестве имени БД используется имя пользователя, заданное при установке соединения.
`[-d] dbname` `[--dbname=] dbname`	Указывает имя БД, в которую устанавливается поддержка процедурного языка. По умолчанию БД создается с таким же именем, как и текущий системный пользователь.

Окончание таблицы 113

Аргумент	Описание
`-e` `--echo`	Показывать команды, которые формируются и отправляются серверу.
`-l` `--list`	Показывать список уже установленных в БД процедурных языков.

Утилита также принимает аргументы командной строки, используемые для установки соединения (см. 18.1).

Пример

Установка поддержки процедурного языка в БД `template1`:

```
$ createlang pltcl template1
```

Примечание. Процедурные языки, установленные в БД `template1`, автоматически устанавливаются в БД, создаваемые впоследствии на ее основе.

18.5. `createuser` - создание роли

Для создания нового пользователя или роли используется утилита `createuser`.

Только суперпользователи и пользователи с привилегией `CREATEROLE` могут создавать новых пользователей и роли.

Для создания нового суперпользователя команда должна выполняться суперпользователем, привилегии `CREATEROLE` для этого недостаточно. Поскольку суперпользователь получает возможность обходить все проверки прав доступа к БД, подобная привилегия не должна легко предоставляться.

Утилита является оболочкой для вызова SQL-команды `CREATE ROLE`, при этом нет разницы в производительности при использовании утилиты или при ином доступе к серверу.

Синтаксис:

```
createuser [option...] [username]
```

Аргументы командной строки приведены в таблице 114.

Таблица 114

Аргумент	Описание
`username`	Указывает имя роли для создания. Имя роли должно быть уникально в пределах существующей конфигурации СУБД PostgreSQL.
`-c number` `--connection-limit=number`	Указывает ограничение количества соединений для роли (по умолчанию — без ограничений).
`-d` `--createdb`	Указывает, что роль может создавать новые БД.

Окончание таблицы 114

Аргумент	Описание
-D --no-createdb	Указывает, что роль не может создавать новые БД (по умолчанию).
-e --echo	Показывать команды, которые формируются и отправляются серверу.
-E --encrypted	Зашифровывает пароль пользователя при хранении в БД. Если не указано, используется поведение пароля по умолчанию.
-g --role	Указывает роль, к которой эта роль будет добавлена сразу в качестве нового члена. Можно указать несколько ролей с помощью множественных опций -g. Примечание. Данный параметр может быть использован только в версии СУБД 9.6.
-i --inherit	Указывает, что роль унаследует привилегии от ролей, членом которых является (по умолчанию).
-I --no-inherit	Указывает, что роль не унаследует привилегии от ролей, членом которых является.
--interactive	При отсутствии в вызове имени пользователя или необходимых аргументов (-d, -D, -r, -R, -s, -S) утилита запросит значения отсутствующих аргументов (такое поведение было по умолчанию до версии PostgreSQL 9.1).
-l --login	Указывает, что новой роли разрешен вход (по умолчанию).
-L --no-login	Указывает, что новой роли не разрешен вход (роль без разрешения входа может использоваться для управления и группировки прав доступа к БД).
-N --unencrypted	Не зашифровывает пароль пользователя при хранении в БД. Если не указано, используется поведение пароля по умолчанию.
-P --pwprompt	Указывает пароль для создаваемой роли. Не является необходимым, если не планируется использование парольной аутентификации.
-r --createrole	Указывает, что роль может создавать новые роли (что означает предоставление создаваемой роли привилегии CREATEROLE).
-R --no-createrole	Указывает, что роль не может создавать новые роли (по умолчанию).
-s --superuser	Указывает, что роль будет суперпользователем.
-S --no-superuser	Указывает, что роль не будет суперпользователем.
--replication	Указывает, что роль будет иметь привилегию установки соединений для репликации REPLICATION.
--no-replication	Указывает, что роль не будет иметь привилегию установки соединений для репликации REPLICATION.

Утилита также принимает аргументы командной строки, используемые для установки соединения (см. 18.1).

Примеры:

1. Создание роли joe на текущем сервере:

```
$ createuser joe
```

2. Интерактивное создание роли joe на текущем сервере:

```
$ createuser --interactive joe
Должна ли роль быть суперпользователем? (y/n) n
Разрешить новой роли создавать базы данных? (y/n) n
Разрешить новой роли создавать пользователей? (y/n) n
```

3. Создание роли joe на сервере eden, порт 5000 без дополнительных вопросов:

```
$ createuser -h eden -p 5000 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

4. Создание суперпользователя joe и задание ему пароля:

```
$ createuser -P -s -e joe
Введите пароль для новой роли: xyzzy
Введите снова: xyzzy
CREATE ROLE joe PASSWORD 'md5b5f5ba1a423792b526f799ae4eb3d59e'
  SUPERUSER CREATEDB CREATEROLE INHERIT LOGIN;
```

В последнем примере вводимый пароль не отображается на экране, он шифруется перед передачей серверу. При использовании аргумента --unencrypted пароль появится в выводимых командах (и возможно в журнале сервера и т. п.), в этом случае не рекомендуется использовать опцию -e, если кто-нибудь может видеть экран.

18.6. dropdb - удаление базы данных

Для удаления существующей БД используется утилита dropdb.

Только суперпользователь или владелец БД может удалить ее.

Утилита является оболочкой для вызова SQL-команды DROP DATABASE, при этом нет разницы в производительности при использовании утилиты или при ином доступе к серверу.

Синтаксис:

```
dropdb [option...] [dbname]
```

Аргументы командной строки приведены в таблице 115.

Таблица 115

Аргумент	Описание
dbname	Указывает имя удаляемой БД.

Окончание таблицы 115

Аргумент	Описание
-e --echo	Показывать команды, которые формируются и отправляются серверу.
-i --interactive	Запрашивать подтверждение перед удалением.
--if-exists	При отсутствии БД ошибка не вызывается, выводится только сообщение.

Утилита также принимает аргументы командной строки, используемые для установки соединения (см. 18.1).

Примеры:

1. Удаление БД demo на текущем сервере:

```
$ dropdb demo
```

2. Удаление БД demo на сервере eden, порт 5000 с подтверждением:

```
$ dropdb -p 5000 -h eden -i -e demo
База данных "demo" будет навсегда удалена.
Вы уверены? (y/n)   y
DROP DATABASE demo;
```

18.7. droplang - удаление поддержки процедурного языка из БД

Для удаления существующей поддержки процедурного языка из БД используется утилита droplang.

Утилита является оболочкой для вызова SQL-команды DROP LANGUAGE.

ВНИМАНИЕ! droplang является устаревшей утилитой и будет устранена в будущих версиях PostgreSQL. Вместо нее рекомендуется использовать команду DROP EXTENSION.

Синтаксис:

```
droplang [option...] langname [dbname]
droplang [option...] --list | -l [dbname]
```

Аргументы командной строки приведены в таблице 116.

Таблица 116

Аргумент	Описание
langname	Указывает удаляемый процедурный язык.
dbname	Указывает имя БД, из которой удаляется поддержка процедурного языка.
[-d] dbname [--dbname=] dbname	Указывает имя БД, из которой удаляется поддержка процедурного языка.
-e --echo	Показывать команды, которые формируются и отправляются серверу.

Окончание таблицы 116

Аргумент	Описание
`-l` `--list`	Показывать список уже установленных в БД процедурных языков.

Утилита также принимает аргументы командной строки, используемые для установки соединения (см. 18.1).

Пример

Удаление поддержки процедурного языка `pltcl` из БД по умолчанию:

```
$ droplang pltcl dbname
```

18.8. `dropuser` - удаление роли

Для удаления существующего пользователя или роли используется утилита `dropuser`.

Только суперпользователи и пользователи с привилегией `CREATEROLE` могут удалять пользователей и роли. Для удаления суперпользователя команда должна выполняться суперпользователем.

Утилита является оболочкой для вызова SQL-команды `DROP ROLE`, при этом нет разницы в производительности при использовании утилиты или при ином доступе к серверу.

Синтаксис:

```
dropuser [option...] [rolename]
```

Аргументы командной строки приведены в таблице 117.

Таблица 117

Аргумент	Описание
`rolename`	Указывает имя удаляемой роли.
`-e` `--echo`	Показывать команды, которые формируются и отправляются серверу.
`-i` `--interactive`	Запрашивать подтверждение перед удалением.
`--if-exists`	При отсутствии роли ошибка не вызывается, выводится только сообщение.

Утилита также принимает аргументы командной строки, используемые для установки соединения (см. 18.1).

Примеры:

1. Удаление пользователя `joe` на сервере по умолчанию:

```
$ dropuser joe
```

2. Удаление роли `joe` на сервере `eden`, порт 5000 с подтверждением:

```
$ dropuser -p 5000 -h eden -i -e joe
Роль "joe" будет навсегда удалена.
Вы уверены? (y/n)
DROP ROLE joe;
```

18.9. ecpg - препроцессор встроенного SQL для программ, написанных на C

В составе PostgreSQL есть утилита `ecpg` — препроцессор встроенного SQL для программ, написанных на C.

Утилита конвертирует C-программы с внедренными SQL-командами в полноценный код C, заменяя SQL-вставки вызовами специальных функций. Результирующий файл может быть обработан в дальнейшем компилятором C.

Утилита конвертирует каждый файл, заданный в командной строке, в соответствующий выходной C-файл. Предпочтительно использовать в качестве расширения входных файлов `.pgc`, в этом случае расширение входных файлов заменяется на `.c` для получения имен результирующих файлов. В противном случае, расширение `.c` присоединяется к полным именам входных файлов. Имя результирующего файла также может быть указано с помощью опции `-o`.

Синтаксис:

```
ecpg [option...] file...
```

Аргументы командной строки приведены в таблице 118.

Таблица 118

Аргумент	Описание
`-c`	Автоматически генерирует код C из встроенного кода SQL; это отражается на `EXEC SQL TYPE`.
`-C mode`	Устанавливает режим совместимости; `mode` может быть одним из `INFORMIX` или `INFORMIX_SE`.
`-D symbol`	Определяет параметр препроцессора C.
`-h`	Разбирает файл заголовка, опция включает в себя `-c`.
`-i`	Разбирает файлы заголовков системы.
`-I directory`	Задает каталог `directory` поиска файлов заголовков, включаемых командой `EXEC SQL INCLUDE`. По умолчанию «.» (текущий каталог), `/usr/local/include`, каталог заголовков PostgreSQL (например, `/usr/include/postgresql`) и `/usr/include` в указанном порядке.
`-o outfile`	Указывает результирующий файл.
`-r option`	Задает поведение времени исполнения; `option` может быть: `no_indicator`, `prepare`, `questionmarks`.

Окончание таблицы 118

Аргумент	Описание
`-t`	Включает автоматическое подтверждение транзакций. В этом режиме SQL-команда автоматически подтверждается, как находящаяся внутри выделенного блока транзакции. По умолчанию команды подтверждаются только после команды `EXEC SQL COMMIT`.

При компиляции полученных C-файлов компилятор должен иметь возможность находить заголовочные файлы `ECPG` в каталоге заголовочных файлов PostgreSQL. В связи с чем, он должен быть указан с помощью опции `-I` при вызове компилятора (например, `-I/usr/include/postgresql`).

Программы, разработанные с использованием встроенного SQL, должны быть собраны с библиотекой `libecpg`, например, с помощью опции сборки `-L/usr/lib -lecpg`.

Значения всех указанных каталогов, соответствующих текущей конфигурации, могут быть получены вызовом `pg_config`.

При наличии C-файла `prog1.pgc` с исходным кодом, использующим встроенный SQL, возможно получение работающей программы следующей последовательностью команд:

```
ecpg prog1.pgc
cc -I/usr/include/postgresql -c prog1.c
cc -o prog1 prog1.o -L/usr/lib -lecpg
```

18.10. pg_basebackup - создание базовой резервной копии кластера

Для создания базовой резервной копии кластера PostgreSQL используется утилита `pg_basebackup`.

Утилита используется для создания базовой резервной копии запущенного кластера PostgreSQL. Это не оказывает влияния на других клиентов БД, и может быть использовано как для непрерывного архивирования и восстановления (см. 14.3), так и для стартовой точки резервных серверов, использующих передачу журналов или потоковую репликацию (см. 15.2).

`pg_basebackup` создает бинарную копию файлов кластера БД, обеспечивая автоматических вход и выход системы из режима создания резервной копии. Резервная копия снимается со всего кластера БД, не существует возможности сделать резервную копию отдельной БД или ее объектов. Для резервирования отдельных БД предназначена утилита `pg_dump`.

Резервная копия создается с помощью обычного соединения с PostgreSQL и использует протокол репликации. Соединение должно быть создано от имени суперпользователя или пользователя с привилегией `REPLICATION` (см. 10.2), при этом `pg_hba.conf` должен

явно разрешать соединения для репликации. Сервер должен быть настроен с достаточными значениями max_wal_senders (как минимум одно соединения для целей резервирования).

Может быть запущено несколько утилит pg_basebackups одновременно, но с точки зрения производительности рекомендуется сделать одну резервную копию, а потом копировать ее.

pg_basebackup может создавать базовую резервную копию не только с основного сервера, но и с резервного. Для снятия резервной копии с резервного сервера необходимо настроить резервный сервер для приема соединения на репликацию (установить max_wal_senders и настроить аутентификацию). Может потребоваться включение full_page_writes на основном сервере.

Существует ряд ограничений при снятии резервной копии с резервного сервера:

- файл истории резервных копий в копируемом кластере не создается;
- не гарантируется, что все необходимые WAL файлы будут помещены по завершению создания резервной копии. При планировании использования резервной копии для восстановления и гарантирования наличия всех необходимых файлов, следует включать их в резервную копию опцией -x;
- если резервный сервер переводится в режим основного во время снятия резервной копии, она завершается с ошибкой;
- все необходимые для резервной копии записи WAL должны содержать достаточное количество операций записи полных страниц, что может требовать включения full_page_writes на основном сервере и неиспользования утилит типа pg_compresslog в качестве archive_command для удаления таких операций из файлов WAL.

Синтаксис:

```
pg_basebackup [option...]
```

Аргументы командной строки приведены в таблице 119.

Таблица 119

Аргумент	Описание
-D directory --pgdata=directory	Задает каталог, в котором будет создаваться резервная копия. pg_basebackup создаст каталог и все необходимые вышестоящие. Каталог может существовать, но, если он не пуст, будет выведена ошибка. Если резервирование выполняется в режиме tar, и каталог задан как -, tar-файл будет выведен в стандартный поток вывода. Опция является обязательной.

Продолжение таблицы 119

Аргумент	Описание
`-F format` `--format=format`	Указывает формат выходного файла. Может быть один из: `p, plain` — вывод в виде обычных файлов, с тем же расположением, как в исходном каталоге данных и табличных пространствах. Если кластер не содержит дополнительных табличных пространств, вся БД помещается в целевой каталог. При наличии дополнительных табличных пространств, основной каталог данных помещается в целевой каталог, а другие табличные пространства помещаются по тем же абсолютным путям, что и на сервере. Является форматом по умолчанию. `t, tar` — вывод в виде архива tar в целевой каталог. Основной каталог данных помещается в файл `base.tar`, а другие табличные пространства будут именоваться согласно их OID. Если целевой каталог задан как `-`, tar-файл будет выведен в стандартный поток вывода для перенаправления его, например в `gzip`. Это доступно только для кластеров, не содержащих дополнительных табличных пространств.
`-r rate --max-rate=rate`	Определяет максимальную скорость передачи данных с сервера. Значения имеют размерность килобайт за секунду. Использование суффикса `M` позволяет установить размерность мегабайт за секунду. Суффикс `k` тоже может быть использован, но не даст никакого эффекта. Корректными значениями являются значения от 32 килобайт до 1024 мегабайт за секунду. Целью параметра является ограничение `pg_basebackup` на работающем сервере. Эта опция влияет на перенос каталога данных. На передачу WAL этот параметр окажет эффект, если методом сбора устнановлен `fetch`. Примечание. Данный параметр может быть использован только в версии СУБД 9.6.
`-R` `--write-recovery-conf`	Создать минимальный файл `recovery.conf` в целевом каталоге (или внутри архивного файла `base`) для упрощения создания резервного сервера.

`-T olddir=newdir`
`--tablespace-mapping=`
`olddir=newdir`
	Переносит табличное пространство из `olddir` в `newdir` в течение создания резервной копии. Для достижения наилучшего результата, `olddir` должен соответствовать пути, определенным при создании этого табличного пространства. (Но это не ошибка, если нет табличного в `olddir` в резервной копии.) Оба значения `olddir` и `newdir` должны быть абсолютными путями. Эта опция может быть указано несколько раз для нескольких табличных пространств. Смотрите примеры ниже.
	Если табличное перенесено таким образом, символьные ссылки внутри основного каталога данных будут обновляены, указывая на новое расположение. Таким образом, новый каталог данных готов к использованию для нового экземпляра сервера со всеми табличными с обновленными расположениями.
	Примечание. Данный параметр может быть использован только в версии СУБД 9.6.

`--xlogdir=xlogdir`	Определяет расположение директории для логов транзации. Значение `xlogdir` должно определять абсолютный путь. Директория для логов транзакции может быть определена только когда производится резервирование в текстовом режиме. Примечание. Данный параметр может быть использован только в версии СУБД 9.6.
`-x` `--xlog`	Использование опции равноценно использованию опции `-X` с методом `fetch`.

Продолжение таблицы 119

Аргумент	Описание
-X method --xlog-method=method	Указывает включение в резервную копию необходимых файлов журнала транзакций (WAL-файлов). Будут включены все журналы транзакции, сформированные во время создания резервной копии. При указании этой опции, возможен запуск postmaster непосредственно в распакованном каталоге без необходимости использования архива журнала, таки образом создается полностью самостоятельная резервная копия. Поддерживаются следующие методы получения журнала транзакций: f, fetch — файлы журнала транзакций выбираются в конце создания резервной копии. Поэтому необходимо, чтобы параметр wal_keep_segments был установлен в значение, при котором не будет удаления журнала транзакций до завершения создания резервной копии. Если между передачей журнала будет выполнена его ротация, создание резервной копии завершится с ошибкой, и копия будет неработоспособной. s, stream — потоковое получения журнала транзакций во время создания резервной копии. При этом с сервером устанавливается второе соединение для параллельного получения журнала транзакций. Следовательно, утилита будет использовать два слота соединений, задаваемых параметром max_wal_senders. Поскольку журнал транзакций держит у себя непосредственно клиент, данный режим не требует сохранения отдельного журнала транзакций на сервере.
-z --gzip	Включает применение сжатия gzip при выводе tar-файла с уровнем сжатия по умолчанию. Сжатие доступно только при использовании формата tar.
-Z level --compress=level	Включает применение сжатия gzip при выводе tar-файла с заданным уровнем сжатия (от 1 до 9, где 9 задает максимальное сжатие). Сжатие доступно только при использовании формата tar.
-c fast|spread --checkpoint=fast|spread	Устанавливает режим создания контрольных точек в fast или spread (по умолчанию).
-l label --label=label	Задает метку резервной копии. Если не задано, используется метка по умолчанию "pg_basebackup base backup".
-P --progress	Включает отображение хода процесса. Включение приводит к выводу ориентировочной информации о ходе процесса создания резервной копии. Поскольку БД могут меняться во время выполнения операции, эта информация носит ориентировочный характер и не может быть верной на 100%. Обычно, при включении в копию файлов WAL, итоговое количество данных не может быть рассчитано заранее, и в этом случае рассчитанный конечный размер будет увеличен после завершения оценки без учета WAL. При включении параметра создание резервной копи начинается с подсчета размера всей БД, и только после этого возвращается к передаче содержимого. Это может привести к увеличению времени создания резервной копии.
-v --verbose	Определяет режим детализированной информации. Будут выведены дополнительные этапы в начале и завершении, с указанием имени обрабатываемого файла, если задан режим отображения хода процесса.
-d connstr --dbname=connstr	Задает строку соединения с сервером. Опция названа --dbname для единообразия с остальными клиентскими приложениями, но поскольку pg_basebackup не устанавливает соединение с конкретной БД, имя БД в строке соединения может быть опущено.

Окончание таблицы 119

Аргумент	Описание
`-s interval` `--status-interval=interval`	Задает время в секундах между отправкой информации о состоянии серверу. Это позволяет облегчить мониторинг ходе процесса на сервере. Нулевое значение полностью отключает обновление статуса, хотя она и отправляется на запрос сервера в целях предотвращения разрыва соединения по таймауту. Значение по умолчанию — 10 секунд.

Утилита также принимает аргументы командной строки, используемые для установки соединения (см. 18.1).

Резервная копия включает все файлы в каталоге данных и табличных пространствах, включая конфигурационные файлы и дополнительные файлы, помещенные в них. В каталоге данных допускаются только обычные файлы и каталоги без символических ссылок или специальных файлов устройств.

Способ управления табличными пространствами PostgreSQL требует, чтобы пути ко всем дополнительным табличным пространствам были идентичны где бы не выполнялось восстановление. При этом каталог данных может располагаться в любом месте.

`pg_basebackup` работает с серверами той же версии или меньше (до 9.1). Потоковый режим WAL (`-X stream`) работает только с серверами версии 9.6.

Примеры:

1. Создание базовой резервной копии сервера `mydbserver` и сохранение его в локальном каталоге `/usr/local/pgsql/data`:

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
```

2. Создание базовой резервной копии локального сервера с одним tar-файлом для каждого табличного пространства в каталоге `backup` с отображением хода процесса:

```
$ pg_basebackup -D backup -Ft -z -P
```

3. Создание базовой резервной копии локальной базы с одним табличным пространством и сжатием ее с помощью `bzip2`:

```
$ pg_basebackup -D - -Ft | bzip2 > backup.tar.bz2
```

Это команда может завершится ошибкой при наличии нескольких табличных пространств в БД.

4. Создание резервной копии локальной базы данных, табличное пространство `/opt/ts` которой переносится в `./backup/ts`:

```
$ pg_basebackup -D backup/data -T /opt/ts=$(pwd)/backup/ts
```

18.11. `pg_config` - получение информации о текущей конфигурации

Для получения информации о текущей конфигурации PostgreSQL используется утилита `pg_config`.

Утилита используется программными средствами, взаимодействующими с

PostgreSQL, для получения информации, например, о путях к заголовочным файлам и библиотекам.

Синтаксис:

```
pg_config [option...]
```

Аргументы командной строки приведены в таблице 120.

Таблица 120

Аргумент	Описание
`--bindir`	Показывает расположение исполняемых файлов, например, для поиска команды `psql`.
`--docdir`	Показывает расположение файлов документации.
`--htmldir`	Показывает расположение HTML-файлов документации.
`--includedir`	Показывает расположение файлов-заголовков C (`.h`) клиентских интерфейсов.
`--pkgincludedir`	Показывает расположение прочих файлов-заголовков C (`.h`).
`--includedir-server`	Показывает расположение файлов-заголовков C (`.h`) сервера.
`--libdir`	Показывает расположение библиотек объектного кода.
`--pkglibdir`	Показывает расположение динамически загружаемых модулей.
`--localedir`	Показывает расположение файлов локализации.
`--mandir`	Показывает расположение страниц `man`.
`--sharedir`	Показывает расположение файлов поддержки независимости от архитектуры.
`--sysconfdir`	Показывает расположение общесистемных конфигурационных файлов.
`--pgxs`	Показывает расположение `makefile` для расширений.
`--configure`	Показывает опции, использованные скриптом `configure` при сборке PostgreSQL. Это может быть использовано для сборки аналогичной конфигурации или получения информации о том, с какими параметрами был собран бинарный пакет.
`--cc`	Показывает использованное при сборке PostgreSQL значение `CC`.
`--cppflags`	Показывает использованное при сборке PostgreSQL значение `CPPFLAGS`.
`--cflags`	Показывает использованное при сборке PostgreSQL значение `CFLAGS`.
`--cflags_sl`	Показывает использованное при сборке PostgreSQL значение `CFLAGS_SL`.
`--ldflags`	Показывает использованное при сборке PostgreSQL значение `LDFLAGS`.
`--ldflags_ex`	Показывает использованное при сборке PostgreSQL значение `LDFLAGS_EX`.
`--ldflags_sl`	Показывает использованное при сборке PostgreSQL значение `LDFLAGS_SL`.

Окончание таблицы 120

Аргумент	Описание
`--libs`	Показывает использованное при сборке PostgreSQL значение `libs`.

При задании более одного аргумента вывод осуществляется построчно в указанном порядке. При отсутствии аргументов будут показаны все известные значения с указанием их наименований.

Опция `--includedir-server` введена в PostgreSQL 7.2. В более ранних версиях заголовочные файлы сервера устанавливаются в тот же каталог, что и заголовочные файлы клиента, получаемые опцией `--includedir`. Для обеспечения поддержки обоих случаев необходимо сначала проверять результат выполнения с новой опцией.

Опции `--docdir`, `--pkgincludedir`, `--localedir`, `--mandir`, `--sharedir`, `--sysconfdir`, `--cc`, `--cppflags`, `--cflags`, `--cflags_sl`, `--ldflags`, `--ldflags_sl` и `--libs` введены в PostgreSQL 8.1. Опция `--htmldir` введена в PostgreSQL 8.4. Опция `--ldflags_ex` введена в PostgreSQL 9.0.

Пример

Сборка конфигурации, аналогичной установленной:

```
eval ./configure 'pg_config --configure'
```

Вывод утилиты `pg_config --configure` содержит кавычки для того, чтобы аргументы, содержащие пробелы, были представлены корректно. Таким образом, использование `eval` необходимо для получения правильного результата.

18.12. `pg_dump` - резервное копирование

Для создания резервной копии БД в виде файла в текстовом или других форматах используется утилита `pg_dump`.

Утилита создает согласованную копию, даже если БД используется, при этом доступ к ней других пользователей (как читающих, так и пишущих) не блокируется.

Резервная копия может создаваться в виде скрипта или форматах упакованного файла. Скрипт резервной копии представляет собой текст, содержащий последовательность SQL-команд, необходимых для воссоздания БД до состояния, в котором она была сохранена. Для восстановления из скрипта он подается на вход утилиты `psql`. Скрипт может быть использован для воссоздания БД даже на другом сервере или архитектуре, и с небольшими изменениями на других СУБД.

Альтернативные форматы упакованного файла могут быть использованы утилитой `pg_restore` для пересоздания БД. Они позволяют выбирать, что именно восстанавливать, или даже менять порядок элементов перед восстановлением. Форматы упакованного файла

разработаны переносимыми между архитектурами.

При использовании форматов упаковки файла pg_dump совместно с pg_restore предоставляет гибкий механизм архивирования и переноса. pg_dump может быть использована для создания резервной копии всей БД, после чего утилита pg_restore может быть использована для просмотра и/или выбора частей резервной копии для восстановления. Наиболее гибкими форматами являются "custom" (-Fc) и "directory" (-Fd). Они позволяют выбирать и менять порядок элементов, поддерживают параллельное восстановление и по умолчанию используют сжатие. Формат "directory" является единственным форматом, поддерживающим параллельное создание резервной копии.

При выполнении утилиты pg_dump необходимо проверять наличие предупреждений, выводимых в стандартных поток ошибок.

Синтаксис:

```
pg_dump [option...] [dbname]
```

Аргументы командной строки приведены в таблице 121.

Таблица 121

Аргумент	Описание
dbname	Указывает имя БД, для которой создается резервная копия. По умолчанию в качестве имени БД используется имя пользователя, заданное при установке соединения.
-a --data-only	Выгрузить только данные, без схемы. Выгружаются данные из таблиц, большие объекты и значения последовательностей. Опция похожа, но в силу исторических причин не идентична --section=data.
-b --blobs	Включать большие объекты в выгрузку. Является поведением по умолчанию, за исключением случаев использования --schema, --table или --schema-only. Таким образом, опция -b применима только для добавления больших объектов в случае создания выборочной резервной копии.
-c --clean	Очищать (удалять) объекты БД перед пересозданием. (Если указана --if-exist, в процессе восстановления может генерироваться некоторые безвредные сообщения об ошибках, если какие-либо объекты не были представлены в базе данных назначения.) Опция имеет смысл только для текстового формата. Для архивных форматов это может быть задано при последующем вызове pg_restore.
-C --create	Включить в выгрузку команды создания БД и подключения к ней. (Для подобного скрипта не имеет значения, к какой БД было установлено соединение перед его выполнением.) При одновременном указании --clean, база данных удаляется и создается заново перед подключением к ней. Опция имеет смысл только для текстового формата. Для архивных форматов это может быть задано при последующем вызове pg_restore.

Продолжение таблицы 121

Аргумент	Описание
`-E encoding` `--encoding=encoding`	Выгружать данные в указанной кодировке. По умолчанию резервная копия создается в кодировке БД. (Другим путем получения подобного результата является установка переменной окружения `PGCLIENTENCODING` в желаемую кодировку для резервной копии.)
`-f file` `--file=file`	Указывает имя выходного файла. Для базовых форматов опция может быть опущена, в этом случае используется стандартный поток вывода. При использовании формата `"directory"` требуется обязательное указание опции, при этом она задает целевой каталог для создания резервной копии, а не файл. Указанный каталог создается `pg_dump` и не должен существовать заранее.
`-F format` `--format=format`	Указывает формат выходного файла. Может быть один из: `p, plain` — вывод в виде тестового SQL-скрипта (по умолчанию); `c, custom` — вывод в виде архива, предназначенного для последующего использования утилитой `pg_restore`. Как и формат `directory` является наиболее гибким форматом и позволяет выбирать и менять порядок объектов при восстановлении. Упакован по умолчанию; `d, directory` — вывод в виде упакованного каталога, предназначенного для последующего использования утилитой `pg_restore`. Создается каталог с отдельными файлами для каждой таблицы или большого объекта и так называемого файла состава резервной копии, содержащего описания сохраненных объектов в формате, пригодном для использования утилитой `pg_restore`. Архив может быть модифицирован стандартными утилитами Unix. К примеру, файлы в неупакованном архиве могут быть сжаты с помощью `gzip`. Архив упакован по умолчанию и поддерживает параллельное сохранения резервной копии; `t, tar` — вывод в виде архива, предназначенного для последующего использования утилитой `pg_restore`. Формат совместим с `directory`; распакованный tar-архив представляет собой обычный `directory` архив. В то же время, данный формат не поддерживает сжатие и имеет ограничение на размер файла таблицы в 8 ГБ. Относительный порядок таблиц не может быть изменен во время восстановления.
`-i` `--ignore-version`	Устаревшая опция, игнорируется.

Продолжение таблицы 121

Аргумент	Описание
`-j njobs` `--jobs=njobs`	Задание параллельного режима создания резервной копии. При этом одновременно сохраняется `njobs` таблиц. Опция уменьшает время создания резервной копии, но в то же время увеличивает нагрузку на сервер БД. Опция может использоваться только с форматом `directory`, позволяющим одновременную запись несколькими процессами. а `pg_dump` открывает `njobs + 1` соединений с БД, поэтому параметр `max_connections` должен быть установлен в достаточное для этого значение. Запрос монопольных блокировок на объектах БД в процессе параллельного создания резервной копии может привести к сбою `pg_dump`. Это связано с тем, что главный процесс `pg_dump` запрашивает разделяемую блокировку на каждый объект, который будет впоследствии сохранен рабочими процессами, для уверенности в том, что никто их не удалит, пока идет процесс создания резервной копии. Если другой клиент после этого запрашивает монопольную блокировку на таблице, блокировка не дается, но ставится в очередь, до снятия блокировки главного процесса. Следовательно, все последующие обращения к таблице не будут предоставлены, а будут выстроены в очередь за монопольной блокировкой. Это относится и к рабочему процессу, пытающемуся сохранить таблицу. Без дополнительных проверок это вызовет классическую взаимную блокировку. Для определения конфликтов рабочий процесс `pg_dump` запрашивает разделяемую блокировку с опцией `NOWAIT`. Если рабочий процесс ее не получает, значит кто-то в это время установил монопольную блокировку, и продолжение сохранения невозможно, так что `pg_dump` завершает работу с ошибкой. Для создания целостной резервной копии, сервер БД должен поддерживать создание синхронизированных снимков, которые появились в PostgreSQL 9.2. С этой возможностью клиенты БД могут быть уверены, что видят один набор данных, даже используя несколько соединений. `pg_dump -j` использует несколько соединений: одно соединение для главного процесса и по одному на каждый рабочий. Без механизма синхронизированных снимков не гарантируется, что разные рабочие процессы видят одни и те же данных в разных соединениях, что может привести к не целостному состоянию резервной копии. При желании запуска параллельного создания резервной копии на серверах версии ниже 9.2 необходимо обеспечить неизменность содержимого БД от момента установки соединения главным процессом до завершения соединения последним рабочим. Наиболее простым способом является остановка всех модифицирующих процессов (DDL и DML) до начала создания резервной копии. При запуске `pg_dump -j` на серверах версии ниже 9.2 требуется указание опции `--no-synchronized-snapshots`.

Продолжение таблицы 121

Аргумент	Описание
-n schema --schema=schema	Выгружать только совпадающие с указанным шаблоном схемы, что включает выгрузку непосредственно схем и всех объектов, которые в них входят. Если опция не указана, из указанной БД выгружаются все несистемные схемы. Может быть указано несколько схем путем использования нескольких аргументов -n. Параметр интерпретируется как шаблон в соответствии с правилами для команды \d утилиты psql, поэтому несколько схем может быть указано с помощью использования специальных символов в шаблоне. При этом необходимо параметр заключать в кавычки для предотвращения его обработки командной оболочкой. Примечания: 1. При указании -n pg_dump пытается сохранять все объекты БД, от которых зависят выбранные схемы. В связи с этим не гарантируется, что результат выборочного резервирования схем будет успешно восстановлен в чистую БД. 2. Не относящиеся к схеме объекты, такие как большие объекты, не сохраняются при указании -n. Большие объекты можно добавить указанием аргумента --blobs.
-N schema --exclude-schema=schema	Не выгружать совпадающие с указанным шаблоном схемы. Параметр интерпретируется по тем же правилам, что и в -n. -N может быть указан несколько раз для исключения схем по разным шаблонам. При одновременном указании -n и -N резервирование осуществляется, как если бы было указано -n без -N. При указании -N без -n схемы исключаются из всего набора схем БД.
-o --oids	Включить OID в выгрузку как часть данных для каждой таблицы. Эту опцию необходимо использовать в случае, когда клиентское приложение тем или иным образом использует OID столбцов (например, для внешних ключей). В иных случаях опция использоваться не должна.
-O --no-owner	Не включать в резервную копию команд установки привилегий владения объектами, как в исходной БД. По умолчанию pg_dump использует команды ALTER OWNER или SET SESSION AUTHORIZATION для установки привилегий владения создаваемыми объектами. Эти команды приведут к ошибке в случае, когда восстановление осуществляется пользователем, не являющимся суперпользователем или владельцем всех содержащихся в скрипте объектов. Для того чтобы скрипт мог быть выполнен любым пользователем, указывается -O, при этом он становится владельцем всех восстанавливаемых объектов. Опция имеет смысл только для текстового формата. Для архивных форматов это может быть задано при последующем вызове pg_restore
-R --no-reconnect	Вышедшая из употребления опция, принимается в целях обратной совместимости.
-s --schema-only	Выгрузить только схему без данных. Опция противоположна --data-only. Опция похожа, но в силу исторических причин не идентична --section=pre-data и --section=post-data. Для исключения данных только некоторого набора таблиц следует использовать --exclude-table-data.

Продолжение таблицы 121

Аргумент	Описание
`-S username` `--superuser=username`	Указать имя суперпользователя для использования при выключении триггеров. Это используется, только если указана опция `--disable-triggers`. (Лучше выполнять восстановление под суперпользователем.)
`-t table` `--table=table`	Выгружать только совпадающие с указанным шаблоном таблицы (представления и последовательности). Может быть указано несколько таблиц путем использования нескольких аргументов `-t`. Параметр интерпретируется как шаблон в соответствии с правилами для команды `\d` утилиты `psql`, поэтому несколько таблиц может быть указано с использованием специальных символов в шаблоне. При этом необходимо параметр заключать в кавычки, для предотвращения его обработки командной оболочкой. Аргументы `-n` и `-N` при указании `-t` игнорируются, поскольку таблицы, выбираемые с помощью `-t`, выгружаются независимо от значения этих аргументов, а другие объекты не выгружаются. Примечания: 1. При указании `-t pg_dump` пытается сохранять все объекты БД, от которых зависят выбранные таблицы. В связи с этим, не гарантируется, что результат выборочного резервирования таблиц будет успешно восстановлен в чистую БД. 2. Поведение аргумента `-t` не является полностью обратно совместимым с версиями PostgreSQL, меньших 8.2. Ранее, при задании `-t tab` выбирались все таблицы с именем `tab`, сейчас же только та, которая доступна в пути поиска объектов по умолчанию. Для получения старого поведения необходимо указывать `-t '*.tab'`. Также можно записать, например `-t sch.tab` для выбора таблицы конкретной схемы, тогда как раньше для этого использовалось выражение `-n sch -t tab`.
`-T table` `--exclude-table=table`	Не выгружать совпадающие с указанным шаблоном таблицы. Параметр интерпретируется по тем же правилам, что и в `-t`. `-T` может быть указано несколько раз для исключения таблиц по разным шаблонам. При одновременном указании `-t` и `-T` резервирование осуществляется, как если бы было указано `-t` без `-T`. При указании `-T` без `-t` указанные таблицы исключаются из всего набора таблиц БД.
`-v` `--verbose`	Определяет режим детализированной информации. При этом `pg_dump` выводит комментарии объектов и время запуска/останова в выходной файл, а также сообщения о ходе процесса в стандартный поток ошибок.

`-x`
`--no-privileges`
`--no-acl`
	Предотвращать выгрузку прав доступа (команд `GRANT`, `REVOKE`).

`-Z` `--compress=0..9`	Задает уровень сжатия для форматов вывода. Ноль означает отсутствие сжатия. Для формата `"custom"` это значение задает уровень сжатия отдельных сегментов табличных данных, и по умолчанию установлен средний уровень сжатия. Для текстового формата установка ненулевого значения вызывает сжатие всего выходного файла, который целиком подается на вход `gzip`, но по умолчанию сжатие не осуществляется.

Продолжение таблицы 121

Аргумент	Описание
`--binary-upgrade`	Только для использования утилитами обновления. Использование аргумента в других целях не рекомендуется и не поддерживается.
`--column-inserts` `--attribute-inserts`	Выгрузить данные как набор команд `INSERT` с заданием названий столбцов (`INSERT INTO таблица (столбец, ...) VALUES ...`). Это сильно замедляет процесс восстановления и в основном используется для загрузки данных в БД под управлением иных (не PostgreSQL) СУБД. Когда генерируется отдельная команда для каждой строки данных, при возникновении ошибки теряется только одна строка, а не все содержимое таблицы.
`--disable-dollar-quoting`	Отключить обрамление тел функций символом `$`, использовать стандартный синтаксис SQL для обрамления.
`--disable-macs`	Отключить сохранение/восстановление мандатных меток. По умолчанию резервная копия создается с сохранением мандатных меток.
`--disable-triggers`	Опция относится только к созданию резервной копии, содержащей только данные, и указывает `pg_dump` включать в скрипт команды временного отключения триггеров таблиц, в которые загружаются данные. Это используется в том случае, когда таблицы содержат проверки целостности или триггеры, вызов которых нежелателен в процессе загрузки данных. Команды, включаемые `--disable-triggers`, должны выполняться суперпользователем. В связи с чем требуется задание суперпользователя опцией `-S` или запуском восстановления от имени суперпользователя. Опция имеет смысл только для текстового формата. Для архивных форматов это может быть задано при последующем вызове `pg_restore`.
`--exclude-table-data=table`	Не выгружать данных, совпадающих с указанным шаблоном таблиц. Параметр интерпретируется по тем же правилам, что и в `-t`. `--exclude-table-data` может быть указано несколько раз для исключения таблиц по разным шаблонам. Опция удобна, когда требуется сохранение определения таблицы без сохранения ее данных. Для исключения сохранения всех данных, следует использовать `--schema-only`.
`--if-exists`	Указание опции позволяет `pg_dump` использовать при восстановлении в командах условия (например, добавлять `IF EXISTS`) при очистке объектов базы данных. Эта опция не корректна, если `--clean` не указана. Примечание. Данный параметр может быть использован только в версии СУБД 9.6.
`--inserts`	Выгрузить данные как набор команд `INSERT` вместо `COPY`. Это сильно замедляет процесс восстановления и в основном используется для загрузки данных в БД под управлением иных (не PostgreSQL) СУБД. Когда генерируется отдельная команда для каждой строки данных, при возникновении ошибки теряется только одна строка, а не все содержимое таблицы. Необходимо отметить, что восстановление может вызвать ошибку в случае переопределения порядка столбцов. Для предотвращения этого безопаснее применять опцию `--column-inserts`.

Продолжение таблицы 121

Аргумент	Описание
`--lock-wait-timeout=timeout`	Не ждать бесконечное время установки разделяемых блокировок сохраняемых таблиц, а вместо этого завершаться ошибкой при невозможности установки блокировки таблицы в течение заданного тайм-аута. Тайм-аут может быть указан в любом формате, который воспринимает команда `SET statement_timeout`.
`--no-synchronized-snapshots`	Позволяет выполнить `pg_dump -j` на серверах версии ниже 9.2 (см. описание опции `-j`).
`--no-tablespaces`	Не сохранять присваивания табличных пространств, при этом все объекты создаются в табличном пространстве, установленном по умолчанию при восстановлении. Опция имеет смысл только для текстового формата. Для архивных форматов это может быть задано при последующем вызове `pg_restore`.
`--no-unlogged-table-data`	Не сохранять содержимое таблиц, созданных как `UNLOGGED`. Опция не влияет на сохранение определения таблиц, предотвращается только сохранение данных. При создании резервной копии на резервном сервере такие таблицы всегда исключаются.
`--quote-all-identifiers`	Принудительное заключение идентификаторов в кавычки. Может быть полезно при переносе резервной копии БД на более высокую версию, в которой могут присутствовать новые ключевые слова.
`--section=sectionname`	Сохранять только указанную секцию. Именем секции может быть `pre-data`, `data` или `post-data`. Опция может быть указана несколько раз для задания разных секций. По умолчанию сохраняются все секции. Секция `data` содержит непосредственно данные таблиц, содержимое больших объектов и значения последовательностей. Секция `post-data` включает определения индексов, триггеров, правил и ограничений (кроме ограничений `CHECK`). Секция `pre-data` содержит все остальные определения.
`--serializable-deferrable`	Использование сериализуемых транзакций для создания резервной копии для гарантии того, что используемый снимок не противоречив с последующим состоянием БД; выполняется ожиданием момента времени в потоке транзакций, при котором отсутствуют аномалии, для снижения риска завершения резервирования с ошибкой или отката других транзакций с ошибкой `serialization_failure`. Опция не приносит преимуществ по созданию резервной копии и введена только для аварийного восстановления. Может быть полезна при создания резервной копии в целях получения отчетов или иных применений только для чтения в то время, пока основная БД продолжает обновляться. Без этой опции резервная копия может отражать состояние, которое не согласуется с любым последовательным исполнением завершенных транзакций. Например, если используются методы пакетной обработки, пакет может отражаться как закрытый в дампе, без появления всех его элементов. Опция не изменяет поведение в случае отсутствия пишущих транзакций в момент запуска `pg_dump`. Если такие транзакции активны, запуск `pg_dump` будет задержан на неопределенное время. После запуска не оказывает `pg_dump` влияние на производительность.

Окончание таблицы 121

Аргумент	Описание
`--use-set-session-authorization`	Использовать команды `SET SESSION AUTHORIZATION` вместо команд `ALTER OWNER TO` для задания прав владения объектами. Это делает резервную копию более совместимой со стандартами, но требует определенного порядка объектов в резервной копии, что может не дать возможности корректного восстановления. В тоже время, восстановление с использованием команды `SET SESSION AUTHORIZATION` требует прав суперпользователя, тогда как `ALTER OWNER` требует меньших привилегий.
`--role=rolename`	Указывает роль, которая будет использоваться при создании резервной копии. Опция указывает `pg_dump` использовать команду `SET ROLE` после подключения к БД. Используется при нехватке необходимых для запуска утилиты привилегий у пользователя (заданного `-U`), но который может их получить, использовав указанную роль. Некоторые конфигурации имеют политику запрета подключения от имени суперпользователя, и использование указанной опции позволяет выполнить операцию без нарушения политики.

Утилита также принимает аргументы командной строки, используемые для установки соединения (см. 18.1).

Утилита `pg_dump` выполняет запросы `SELECT`. Если при запуске `pg_dump` возникают ошибки, следует убедится, что существует возможность выборки данных из БД с помощью, например `psql`. Также применяются все параметры установки соединения и переменные окружения, используемые `libpq`.

Действия, выполняемые `pg_dump` обычно учитываются сборщиком статистики. Если это нежелательно, необходимо установить параметр `track_counts` в `false` с помощью `PGOPTIONS` или команды `ALTER USER`.

Если кластер БД содержит какие-либо локальные дополнения в БД `template1`, необходимо быть внимательным при восстановлении в чистую БД, иначе можно получить ошибки при попытке повторного определения объектов. Для создания чистой БД без локальных добавлений ее необходимо создавать из шаблона БД `template0`, а не `template1`, например:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

При создании резервной копии, содержащей только данные, и применении опции `--disable-triggers` `pg_dump` вставляет команды для отключения триггеров на таблицах перед вставкой в них данных и команд их включения после вставки. При остановке процесса восстановления посередине системный каталог может оказаться в несогласованном состоянии.

Необходимо помнить, что архивы tar ограничены размером в 8 ГБ. (Это унаследованное ограничение формата tar-файлов.) В связи с этим, этот формат не может использоваться, если текстовое представление какой-нибудь таблицы превышает указанное ограничение.

Общий размер tar-архива или иных форматов вывода не ограничен, за исключением возможных ограничений со стороны ОС.

Созданный утилитой pg_dump архивный файл не содержит информации о статистике, которую использует оптимизатор запросов. Выполнять команду ANALYZE после восстановления из резервной копии для достижения лучшей производительности (см. 13.1.3). Архивный файл также не содержит команд ALTER DATABASE ... SET, эти установки архивируются утилитой pg_dumpall вместе с информацией о пользователях и других глобальных параметрах установки.

Поскольку pg_dump используется для переноса данных в новые версии PostgreSQL, то ожидается, что результат выполнения будет загружен в сервер PostgreSQL более новой версии, чем pg_dump. pg_dump также может делать резервные копии с серверов PostgreSQL версии ниже своей (поддерживаются версии от 7.0). Однако pg_dump не может делать резервные копии с серверов PostgreSQL, чьи версии новее его основной версии, при этом не производится даже попытки исполнения, т. к. высок риск получения испорченной резервной копии. Также не гарантируется восстановление резервной копии на сервере PostgreSQL более старой основной версии, даже если она была снята с сервера такой же версии. Загрузка резервной копии в сервер старой версии может требовать ручного редактирования для устранения непонятного для него синтаксиса.

Примеры:

1. Создание резервной копии БД mydb в виде SQL-скрипта:

```
$ pg_dump mydb > db.sql
```

2. Загрузка подобного скрипта в новую БД newdb:

```
$ psql -d newdb -f db.sql
```

3. Создание резервной копии БД mydb в формате "custom":

```
$ pg_dump -Fc mydb > db.dump
```

4. Создание резервной копии БД mydb в формате "directory":

```
$ pg_dump -Fd mydb -f dumpdir
```

5. Создание резервной копии БД mydb в формате "directory" параллельно 5-ю процессами:

```
$ pg_dump -Fd mydb -j 5 -f dumpdir
```

6. Загрузка подобного архивного файла в новую БД newdb:

```
$ pg_restore -d newdb db.dump
```

7. Создание резервной копии таблицы mytab:

```
$ pg_dump -t mytab mydb > db.sql
```

8. Создание резервной копии всех таблиц, начинающихся с emp в схеме detroit, за исключением таблицы employee_log:

```
$ pg_dump -t 'detroit.emp*' -T detroit.employee_log mydb > db.sql
```

9. Создание резервной копии всех схем, начинающихся с east или west и заканчивающихся на gsm, исключая все схемы, содержащие слово test:

```
$ pg_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb > db.sql
```

10. То же, используя регулярные выражения:

```
$ pg_dump -n '(east|west)*gsm' -N '*test*' mydb > db.sql
```

11. Создание резервной копии всех объектов БД, за исключением таблиц, начинающихся с ts_:

```
$ pg_dump -T 'ts_*' mydb > db.sql
```

Для задания имен в верхнем регистре или смешанном в -t и подобных опциях необходимо заключать параметры в двойные кавычки, иначе они будут приведены к нижнему регистру. Но двойные кавычки являются специальным символами в командной оболочке, так что они также должны заключаться в кавычки. Таким образом, создание резервной копии одной таблицы в смешанном написании может быть выполнено следующим образом:

```
$ pg_dump -t "\"MixedCaseName\"" mydb > mytab.sql
```

18.13. pg_dumpall - резервное копирование кластера

Для создания резервной копии кластера в виде скрипта используется утилита pg_dumpall.

Утилита сохраняет резервные копии всех БД кластера в один файл скрипта. Скрипт содержит SQL-команды и может быть подан в дальнейшем на вход утилиты psql для восстановления. Операция осуществляется последовательным вызовом утилиты pg_dump для каждой БД кластера. Кроме того, pg_dumpall сохраняет глобальные объекты, единые для всех БД. Эти объекты включают в себя информацию о пользователях и группах и такие свойства, как: права доступа, применяемые для всех БД в целом.

Поскольку pg_dumpall читает таблицы из всех БД, то наиболее вероятно, что потребуется осуществлять соединение с БД в качестве суперпользователя, чтобы получить полную резервную копию. Также привилегии суперпользователя потребуются для выполнения сохраненного скрипта, чтобы добавлять пользователей, группы и создавать БД.

SQL-скрипт выводится в стандартный поток вывода. Для перенаправления вывода в файл могут быть использованы операторы командной оболочки.

pg_dumpall требует установления нескольких соединений с сервером PostgreSQL (по одному на каждую БД). При использования парольного метода аутентификации пароль каждый раз будет запрошен заново. В подобном случае удобно иметь файл ~/.pgpass.

Синтаксис:

```
pg_dumpall [option...]
```

Аргументы командной строки приведены в таблице 122.

Таблица 122

Аргумент	Описание
-a --data-only	Выгрузить только данные, без схемы. Опция имеет смысл только для текстового формата. Для архивных форматов это может быть задано при последующем вызове pg_restore.
-c --clean	Очищать (удалять) объекты БД перед пересозданием. Опция имеет смысл только для текстового формата. Для архивных форматов это может быть задано при последующем вызове pg_restore.
-f filename --file=filename	Указывает имя выходного файла. Если не указано, используется стандартный поток вывода.
-g --globals-only	Сохранять только глобальные объекты (роли и табличные пространства) без БД.
-i --ignore-version	Устаревшая опция, игнорируется.
-o --oids	Включить OID в выгрузку как часть данных для каждой таблицы. Эту опцию необходимо использовать в случае, когда клиентское приложение тем или иным образом использует OID столбцов (например, для внешних ключей). В иных случаях опция использоваться не должна.
-O --no-owner	Не включать в резервную копию команд установки привилегий владения объектами, как в исходной БД. По умолчанию используются команды ALTER OWNER или SET SESSION AUTHORIZATION для установки привилегий владения создаваемыми объектами. Эти команды приведут к ошибке в случае, когда восстановление осуществляется пользователем, не являющимся суперпользователем или владельцем всех содержащихся в скрипте объектов. Для того чтобы скрипт мог быть выполнен любым пользователем указывается -O, при этом он становится владельцем всех восстанавливаемых объектов. Опция имеет смысл только для текстового формата. Для архивных форматов это может быть задано при последующем вызове pg_restore.
-r --roles-only	Сохранять только роли без БД и табличных пространств.
-s --schema-only	Выгрузить только схему без данных.
-S username --superuser=username	Указать имя суперпользователя для использования при выключении триггеров. Это используется, только если указана опция --disable-triggers. (Лучше выполнять восстановление под суперпользователем.).
-t --tablespaces-only	Сохранять только табличные пространства без БД и ролей.
-v --verbose	Определяет режим детализированной информации. При этом pg_dumpall выводит время запуска/останова в выходной файл, и сообщения о ходе процесса в стандартный поток ошибок. Также включает режим verbose у вложенных вызовов pg_dump.
-x --no-privileges --no-acl	Предотвращать выгрузку прав доступа (команд GRANT, REVOKE).
--binary-upgrade	Только для использования утилитами обновления. Использование аргумента в других целях не рекомендуется и не поддерживается.

Продолжение таблицы 122

Аргумент	Описание
--column-inserts --attribute-inserts	Выгрузить данные как набор команд INSERT с заданием названий столбцов (INSERT INTO таблица (столбец, ...) VALUES ...). Это сильно замедляет процесс восстановления и в основном используется для загрузки данных в БД под управлением иных (не PostgreSQL) СУБД. Когда генерируется отдельная команда для каждой строки данных, при возникновении ошибки теряется только одна строка, а не все содержимое таблицы.
--disable-dollar-quoting	Отключить обрамление тел функций символом $, использовать стандартный синтаксис SQL для обрамления.
--disable-macs	Отключить сохранение/восстановление мандатных меток. По умолчанию резервная копия создается с сохранением мандатных меток.
--disable-triggers	Опция относится только к созданию резервной копии, содержащей только данные, и указывает pg_dump включать в скрипт команды временного отключения триггеров таблиц, в которые загружаются данные. Это используется в том случае, когда таблицы содержат проверки целостности или триггеры, вызов которых нежелателен в процессе загрузки данных. Команды, включаемые --disable-triggers, должны выполняться суперпользователем. В связи с чем требуется задание суперпользователя опцией -S или запуском восстановления от имени суперпользователя. Опция имеет смысл только для текстового формата. Для архивных форматов это может быть задано при последующем вызове pg_restore.
--if-exists	Указание опции позволяет pg_dump использовать при восстановлении в командах условия (например, добавлять IF EXISTS) при очистке объектов базы данных. Эта опция не корректна, если --clean не указана. Примечание. Данный параметр может быть использован только в версии СУБД 9.6.
--inserts	Выгрузить данные как набор команд INSERT вместо COPY. Это сильно замедляет процесс восстановления и в основном используется для загрузки данных в БД под управлением иных (не PostgreSQL) СУБД. Когда генерируется отдельная команда для каждой строки данных, при возникновении ошибки теряется только одна строка, а не все содержимое таблицы. Необходимо отметить, что восстановление может вызвать ошибку в случае переопределения порядка столбцов. Для предотвращения этого безопаснее применять опцию --column-inserts.
--lock-wait-timeout=timeout	Не ждать бесконечное время установки разделяемых блокировок сохраняемых таблиц, а вместо этого завершаться ошибкой при невозможности установки блокировки таблицы в течение заданного тайм-аута. Тайм-аут может быть указан в любом формате, который воспринимает команда SET statement_timeout.
--no-tablespaces	Не сохранять присваивания табличных пространств, при этом все объекты создаются в табличном пространстве, установленном по умолчанию при восстановлении. Опция имеет смысл только для текстового формата. Для архивных форматов это может быть задано при последующем вызове pg_restore.

Окончание таблицы 122

Аргумент	Описание
`--no-unlogged-table-data`	Не сохранять содержимое таблиц, созданных как `UNLOGGED`. Опция не влияет на сохранение определения таблиц, предотвращается только сохранение данных. При создании резервной копии на резервном сервере такие таблицы всегда исключаются.
`--quote-all-identifiers`	Принудительное заключение идентификаторов в кавычки. Может быть полезно при переносе резервной копии БД на более высокую версию, в которой могут присутствовать новые ключевые слова.
`--use-set-session-authorization`	Использовать команду `SET SESSION AUTHORIZATION` вместо команды `ALTER OWNER TO` для задания прав владения объектами. Это делает резервную копию более совместимой со стандартами, но требует определенного порядка объектов в резервной копии, что может не дать возможности корректного восстановления. В тоже время, восстановление с использованием команды `SET SESSION AUTHORIZATION` требует прав суперпользователя, тогда как `ALTER OWNER` требует меньших привилегий.
`--role=rolename`	Указывает роль, которая будет использоваться при создании резервной копии. Опция указывает `pg_dump` использовать команду `SET ROLE` после подключения к БД. Используется при нехватке необходимых для запуска утилиты привилегий у пользователя (заданного `-U`), но который может их получить, использовав указанную роль. Некоторые конфигурации имеют политику запрета подключения от имени суперпользователя, и использование указанной опции позволяет выполнить операцию без нарушения политики.

Утилита также принимает аргументы командной строки, используемые для установки соединения (см. 18.1).

Поскольку `pg_dumpall` вызывает `pg_dump`, некоторые диагностические сообщения могут относится к `pg_dump`.

После восстановления рекомендуется выполнение команды `ANALYZE` для каждой БД для достижения лучшей производительности. Также возможно выполнение команды `vacuumdb -a -z` для проведения анализа всех БД.

`pg_dumpall` требует наличия всех каталогов, относящихся к табличным пространствам перед выполнением операции восстановления, иначе БД, расположенные не в стандартных каталогах не смогут быть восстановлены.

Примеры:

1. Создание резервной копии всех БД:

```
$ pg_dumpall > db.out
```

2. Восстановление сохраненных таким образом БД:

```
$ psql -f db.out postgres
```

Не имеет значения к какой БД было осуществлено соединение, т. к. созданный с помощью `pg_dumpall` скрипт содержит соответствующие команды для создания и соединения

для указанных БД.

18.14. pg_isready - проверка статуса соединения с сервером PostgreSQL

Для проверка статуса соединения с сервером PostgreSQL используется утилита pg_isready.

Примечание. Введена в версии PostgreSQL 9.6.

Результат возврата утилиты отражает результат проверки соединения.

Синтаксис:

```
pg_isready [option...]
```

Аргументы командной строки приведены в таблице 123.

Таблица 123

Аргумент	Описание
-d dbname --dbname dbname	Указывает БД для соединения. Если параметр содержит знак равенства = или начинается с корректного URI-префикса (postgresql:// или postgres://), он рассматривается как строка соединения.
-q --quiet	Не выводить сообщение о статусе.
-t seconds --timeout=seconds	Максимальное время в секундах ожидания установки соединения до возврата сообщения о недоступности сервера. Установка в 0 отключает ожидание. По умолчанию используется значение 3 секунды.

pg_isready возвращает 0 при нормальной установке соединения с сервером, 1, если сервер отверг соединение (например во время своего запуска), 2, если не было ответа на попытку доступа, и 3, если попытка установки соединения не выполнялась (например, из за неверных параметров запуска).

Утилита также принимает аргументы командной строки, используемые для установки соединения (см. 18.1).

Примечание. Не требуется обеспечивать корректное имя пользователя, пароль или имя базы данных для получения значение статуса сервера. Однако, если неправильные значения предоставляются, сервер будет пытаться по заведомо некорректным параметрам.

Примеры:

1. Обычное использование:

```
$ pg_isready
/tmp:5432 - accepting connections
$ echo $?
0
```

2. Запуск с параметрами соединения во время старта кластера:

```
$ pg_isready -h localhost -p 5433
localhost:5433 - rejecting connections
```

```
$ echo $?
1
```

3. Запуск с параметрами соединения к не отвечающему кластеру:

```
$ pg_isready -h someremotehost
someremotehost:5432 - no response
$ echo $?
2
```

18.15. pg_receivexlog - получение потока журнала транзакций с кластера PostgreSQL

Для получения потока журнала транзакций с кластера PostgreSQL используется утилита pg_basebackup.

Журнал транзакций получается с помощью протокола потоковой репликации и записывается в файлы локального каталога. Этот каталог может быть использован как архив журнала для восстановления с помощью непрерывного архивирования и восстановления (см. 14.3).

pg_receivexlog получает журнал транзакций в реальном времени по мере его генерации сервером без ожидания полного заполнения сегментов, как это выполняет archive_command. По это причине при использовании pg_receivexlog не требуется установка archive_timeout.

Журнал транзакций получается с помощью обычного соединения с PostgreSQL и использует протокол репликации. Соединение должно быть создано от имени суперпользователя или пользователя с привилегией REPLICATION (см. 10.2), при этом pg_hba.conf должен явно разрешать соединения для репликации. Сервер должен быть настроен с достаточными значениями max_wal_senders (как минимум одно соединения для целей резервирования).

При потере соединения, или, если оно не может быть установлено, при не фатальных ошибках pg_receivexlog бесконечно повторяет попытки установки соединения для наискорейшего восстановления потока данных. Для предотвращения этого используется параметр -n.

Синтаксис:

```
pg_receivexlog [option...]
```

Аргументы командной строки приведены в таблице 124.

Таблица 124

Аргумент	Описание
-D directory --pgdata=directory	Задает каталог, в который будет выполнятся вывод. Опция является обязательной.
-n --no-loop	Не повторять попытку установки соединения в случае ошибки, а завершаться с ошибкой.
-s interval --status-interval=interval	Указывает число секунд между приходом пакетов статуса на сервер. Это дает возможность простого мониторинга процессов с сервера. Значение параметра, равное нулю выключает обновления статуса полностью, хотя обновления будут приходить с сервера в обход задержки. Значением по умолчанию является 10 секунд. Примечание. Данный параметр может быть использован только в версии СУБД 9.6.
-S slotname --slot=slotname	Требует pg_receivexlog использовать существующий слот репликации (см. 15.2.6). Если опция используется, то pg_receivexlog будет сообщать об очистке позиции сервера, отмечаяя, когда каждый сегмент будет синхронизирован на диск, соответственно, сервер может удалить этот сегмент, если он не нужен. При использовании этого параметра необходимо удостовериться, что pg_receivexlog не может стать синхронным резервным сервером при некорректных установках synchronous_standby_names (см. 8.6.2). Примечание. Данный параметр может быть использован только в версии СУБД 9.6.
-d connstr --dbname=connstr	Задает строку соединения с сервером. Опция названа --dbname для единообразия с остальными клиентскими приложениями, но поскольку pg_receivexlog не устанавливает соединение с конкретной БД, имя БД в строке соединения может быть опущено.
-s interval --status-interval=interval	Задает время в секундах между отправкой информации о состоянии серверу. Это позволяет облегчить мониторинг о ходе процесса на сервере. Нулевое значение полностью отключает обновление статуса, хотя она и отправляется на запрос сервера в целях предотвращения разрыва соединения по таймауту. Значение по умолчанию 10 секунд.

При использовании pg_receivexlog вместо archive_command сервер продолжает циркуляцию файлов журнала транзакций, даже если резервная копия не была корректно сохранена, т.к. нет команды, возвращающей ошибку. Это может быть решено с помощью archive_command, возвращающей ошибку, если файл не был корректно архивирован, например:

```
archive_command = 'sleep 5 && test -f /mnt/server/archivedir/%f'
```

Начальный таймаут необходим, поскольку pg_receivexlog использует асинхронную репликацию, и может немного отставать от основного сервера.

Пример

Получение журнала транзакций с сервера mydbserver и сохранение в локальный каталог /usr/local/pgsql/archive:

```
$ pg_receivexlog -h mydbserver -D /usr/local/pgsql/archive
```

18.16. pg_recvlogical - контроль потоков логического декодирования

Примечание. Введена в версии PostgreSQL 9.6.

pg_recvlogical контролирует логическое декодирование слотов репликации и потоков данных с этих слотов.

Она создает подключение в режиме репликации, таким образом, оно подлежит таким же ограничениям как pg_receivexlog (см. 18.15). Дополнительно к ограничениям добавляются ограничения логической репликации.

Синтаксис:

```
pg_recvlogical [option...]
```

Аргументы командной строки приведены в таблице 125.

Таблица 125

Аргумент	Описание
--create-slot	Создает новый слот логической репликации, имя которого указано опцией --slot, используя плагин, указанный опцией --plugin, для базы данных, указанной опцией --dbname.
--drop-slot	Удаляет слот репликации, имя которого указано опцией --slot, потом завершает работу.
--start	Начинает изменения потока со слота логической репликации, имя которого указано опцией --slot, до выхода по сигналу. Если на серверной стороне поток завершится по сигналу выключения или отключения, повторяет действия при условии указания опции --no-loop. Формат потока определяется входным плагином, указанный при создании слота. Будет использоваться то же самое подключение, как и при создании слота. Опции --create-slot and --start могут быть указаны вместе. С опцией --drop-slot не могут быть указаны с каким-либо другой опцией.
-f filename --file=filename	Записывает принятые и декодированные данные транзакции в файл. Используйте символ - для перенаправление в стандартный поток вывода (stdout).
interval_seconds --fsync-interval= interval_seconds	Опеределяет насколько часто pg_recvlogical должен использовать вызовы fsync(), чтобы обеспечить гарантированную запись файла на диск. Сервер будет иногда спрашивать клиента для выполнения очистки и сообщить позицию очистки сервера. Указание значения 0 отключает вызовы fsync(), хотя отчетносты прогресса отправляется на сервер. В этом случае, данные могут быть потеряны в случае ошибки.
-I lsn --startpos=lsn	В режиме --start, репликация выполняется с указанного LSN. Игнорируется в других режимах.
-n --no-loop	Если подключение к серверу потеряно, не пытаться повторить подключение в цикле, просто завершить свою работу.
-o name[=value] --option=name[=value]	Устанавливает параметру name плагина вывода значение value, если оно указано.
P plugin --plugin=plugin	При создании слота указывает плагин логического декодирования. Эта опция не окажет эффекта, если слот уже создан.

Окончание таблицы 125

Аргумент	Описание
interval_seconds --status-interval= interval_seconds	Эта опция имеет тот же самый эффект, как и аналогичная в утилите pg_receivexlog (см. 18.15).
-S slot_name --slot=slot_name	В режиме --start использует существующий слот логической репликации с именем slot_name. В режиме --create-slot создает слот с указанным именем. В режиме --drop-slot удаляет слот с указанном именем.
-v --verbose	Включает расширенный режим.
d database --dbname=database	Имя базы данных, к которой необходимо подключиться. По умолчанию совпадает с именем пользователя.
-h hostname-or-ip --host=hostname-or-ip	Указывает имя хоста, к которому подключается сервер. Если значение начинается с /, оно используется как директория сокета Unix-домена. Значение по умолчанию берется из переменной окружения PGHOST, если установлена, иначе принимается соединение по сокету Unix-домена.
-p port --port=port	Указывает порт TCP или сокета локального Unix домена, на котором сервер слушает подключения. По умолчанию используется значение переменной окружения PGPORT, если установлена, или значение, установленное при сборке.
-U user --username=user	Имя пользователя, который подключается. По умолчанию соответсвует текущему имени пользователя операционной системы.
-w --no-password	Никогда не спрашивать пароль. Если сервер требует провести парольную аутентификацию и пароль не обнаружен в файле .pgpass, в подключении будет отказано. Эта опция полезна при использовании скриптов, где пользователю не требуется вводить пароль.
-W --password	Заставлять pg_recvlogical требовать пароль до того, как пользователь подключается к базе данных. Эта опция е важна, так как pg_recvlogical автоматически запросит пароль, если сервер требует аутентификацию по паролю. В некоторых случаях стоит использовать опцию W, чтобы избежать дополнительных попыток подключения.
-V --version	Вывести версию pg_recvlogical и завершить работу.
-? --help	Вывести помощью по pg_recvlogical, его аргументам командной строки и завершить работу.

18.17. pg_restore - восстановление резервной копии

Для восстановления архивов резервных копий БД используется утилита pg_restore.

Утилита pg_restore предназначена для восстановления БД из архивов, созданных утилитой pg_dump в одном из нетекстовых форматов. Она осуществляет команды, необходимые для воссоздания БД до состояния на момент времени создания резервной копии. Архивные файлы также позволяют выбирать с помощью утилиты pg_restore, что именно восстанавливать, и даже менять порядок восстанавливаемых элементов. Файлы архивов разработаны переносимыми между разными архитектурами.

pg_restore может функционировать в двух режимах. При указании БД архив вос-

станавливается непосредственно в нее. В другом случае, скрипт, содержащий необходимые для пересоздания БД SQL-команды, создается и выводится в файл или стандартный поток вывода. Результирующий скрипт эквивалентен формату текстового вывода утилиты pg_dump. Вследствие этого некоторые опции, управляющие выводом, аналогичны опциям pg_dump.

Очевидно, что pg_restore не может восстановить информацию, не содержащуюся в архиве. К примеру, если архив был создан с указанием «сохранять данные в виде команд INSERT», pg_restore не сможет загружать данные, используя команды COPY.

Синтаксис:

```
pg_restore [option...] [file]
```

Аргументы командной строки приведены в таблице 126.

Таблица 126

Аргумент	Описание
file	Указывает путь к файлу архива, который должен быть восстановлен. Если не указан, используется стандартный поток ввода.
-a --data-only	Восстанавливать только данные, без схемы.
-c --clean	Очищать (удалять) объекты БД перед пересозданием. (Если указана --if-exist, в процессе восстановления может генерироваться некоторые безвредные сообщения об ошибках, если какие-либо объекты не были представлены в базе данных назначения.)
-C --create	Создавать БД перед восстановлением в нее. При одновременном указании --clean, база данных удаляется и создается заново перед подключением к ней. При указании этой опции БД, указанная с помощью опции -d, используется только для выполнения команд DROP DATABASE и CREATE DATABASE. Данные восстанавливаются уже в созданную БД.
-d dbname --dbname=dbname	Подсоединиться к указанной БД и выполнить восстановление в нее.
-e --exit-on-error	Завершать работу, если обнаружена ошибка при посылке SQL-команды БД. По умолчанию работа продолжается, а к концу восстановления отображается общее количество ошибок.
-f filename --file=filename	Указывает имя выходного файла для скрипта или листинга при указании -l. Если не указано, используется стандартный поток вывода.
-F format --format=format	Указывает формат входного файла. Не является обязательным, т. к. pg_restore определяет формат архива автоматически. Может быть один из: c, custom: — формат "custom" pg_dump; d, directory: — формат "directory" pg_dump; t, tar: — tar-архив pg_dump.
-i --ignore-version	Устаревшая опция, игнорируется.
-I index --index=index	Восстановить определения только указанного индекса. Несколько индексов могут быть указаны с помощью нескольких опций -I

Продолжение таблицы 126

Аргумент	Описание
-j njobs --jobs=njobs	Указывает количество одновременно выполняемых работ для выполнения этапов работы pg_restore, таких как загрузка данных, создание индексов и создание ограничений. Опция может значительно уменьшить время восстановления большой БД на многопроцессорном сервере. Каждая работа представляет собой процесс или поток в зависимости от ОС и использует отдельное соединение с сервером. Оптимальное значение этого параметра зависит от аппаратной конфигурации сервера, клиента и сети. Влияющие факторы включают в себя количество ядер процессора и параметры дисковой системы. Рекомендуется начинать с количества, равного числу процессорных ядер сервера, но во многих случаях большее значение может привести к большей производительности. В тоже время очевидно, что очень большие значения могут привести к ухудшению производительности за счет перегрузки. Опция применима только для архивов в форматах "custom" и "directory". При этом входной файл должен быть регулярным (к примеру, не являться каналом). В режиме вывода скрипта, а не подключения к серверу БД, опция игнорируется. Также множество работ не может быть использовано при указании опции --single-transaction.
-l --list	Отобразить содержимое архива. Вывод операции может быть использован совместно с опцией -L для ограничения или изменения порядка восстанавливаемых элементов. Если используются опции -n или -t, они ограничивают выводимый список элементов.
-L list-file --use-list=list-file	Восстановить только указанные в списке элементы в том порядке, в котором они в нем указаны. Если используются опции -n или -t, они ограничивают список элементов. Обычно список элементов создается путем редактирования результата предыдущей операции -l. Строки могут быть перемещены или закомментированы символом «;» в начале строки.
-n namespace --schema=schema	Восстанавливать объекты, принадлежащие только указанной схеме. Может комбинироваться с опцией -t для восстановления отдельной таблицы.
-O --no-owner	Не включать в резервную копию команд установки привилегий владения объектами, как в исходной БД. По умолчанию pg_dump использует команды ALTER OWNER или SET SESSION AUTHORIZATION для установки привилегий владения создаваемыми объектами. Эти команды приведут к ошибке в случае, когда восстановление осуществляется пользователем, не являющимся суперпользователем или владельцем всех содержащихся в скрипте объектов. Для того чтобы скрипт мог быть выполнен любым пользователем указывается -O, при этом он становится владельцем всех восстанавливаемых объектов.
-P --function= function-name (argtype [,...])	Восстановить только указанную функцию. Несколько функций могут быть указаны с помощью нескольких опций -P. Имя и аргументы функции должны быть указаны так, как они представлены в списке содержимого архива.
-R --no-reconnect	Вышедшая из употребления опция принимается в целях обратной совместимости.

Продолжение таблицы 126

Аргумент	Описание
-s --schema-only	Выгрузить только схему без данных. Текущие значения последовательностей также не восстанавливаются. Несколько схем могут быть указаны с помощью нескольких опций -s (Не путать с опцией --schema, которая использует слово «schema» в ином значении.).
-S username --superuser=username	Указать имя суперпользователя для использования при выключении триггеров. Это используется, только если указана опция --disable-triggers.
-t table --table=table	Восстанавливать определение и/или данные только указанной таблицы. Несколько таблиц могут быть указаны с помощью нескольких опций -t.
-T trigger --trigger=trigger	Восстанавливать только указанный триггер. Несколько таблиц может быть указаны с помощью нескольких опций -T.
-v --verbose	Определяет режим детализированной информации.
-x --no-privileges -no-acl	Предотвращать восстановление прав доступа (команд GRANT, REVOKE).
-1 --single-transaction	Выполнять восстановление в одной транзакции (т. е. окружать выполняемые команды в блок BEGIN/COMMIT). Это гарантирует, что либо все команды выполнятся успешно, либо не будет осуществлено никаких изменений. Опция включает в себя --exit-on-error.
--disable-triggers	Опция относится только к созданию резервной копии, содержащей только данные, и указывает pg_restore выполнять команды временного отключения триггеров таблиц, в которые загружаются данные. Это используется в том случае, когда таблицы содержат проверки целостности или триггеры, вызов которых нежелателен в процессе загрузки данных. Команды, включаемые --disable-triggers, должны выполняться суперпользователем. В связи с чем требуется задание суперпользователя опцией -S или запуском восстановления от имени суперпользователя.
--if-exists	Указание опции позволяет pg_dump использовать при восстановлении в командах условия (например, добавлять IF EXISTS) при очистке объектов базы данных. Эта опция не корректна, если --clean не указана. Примечание. Данный параметр может быть использован только в версии СУБД 9.6.
--no-data-for-failed-tables	По умолчанию восстановление данных производится даже при ошибке команды создания таблицы (т. к. возможно она уже существовала к этому времени). При указании этой опции восстановление данных для подобной таблицы пропускается. Такое поведение полезно, если БД уже содержит эту информацию. Например, вспомогательные таблицы для расширений PostgreSQL, таких как PostGIS могут уже быть загружены; необходимо применять указанную опцию для предотвращения загрузки дублированных или устаревших данных. Опция полезна только при восстановления непосредственно в БД, а не для производства SQL-скрипта.

Окончание таблицы 126

Аргумент	Описание
`--no-tablespaces`	Не сохранять присваивания табличных пространств, при этом все объекты создаются в табличном пространстве, установленном по умолчанию при восстановлении.
`--section=sectionname`	Восстанавливать только указанную секцию. Именем секции может быть `pre-data`, `data` или `post-data`. Опция может быть указана несколько раз для задания разных секций. По умолчанию восстанавливаются все секции. Секция `data` содержит непосредственно данные таблиц, содержимое больших объектов и значения последовательностей. Секция `post-data` включает определения индексов, триггеров, правил и ограничений (кроме ограничений `CHECK`). Секция `pre-data` содержит все остальные определения.
`--use-set-session-authorization`	Использовать команду `SET SESSION AUTHORIZATION` вместо команд `ALTER OWNER TO` для задания прав владения объектами. Это делает резервную копию более совместимой со стандартами, но требует определенного порядка объектов в резервной копии, что может не дать возможности корректного восстановления.
`--role=rolename`	Указывает роль, которая будет использоваться при восстановлении резервной копии. Опция указывает `pg_restore` использовать команды `SET ROLE` после подключения к БД. Используется при нехватке необходимых для запуска утилиты привилегий у пользователя (заданного `-U`), но который может их получить, использовав указанную роль. Некоторые конфигурации имеют политику запрета подключения от имени суперпользователя, и использование указанной опции позволяет выполнить операцию без нарушения политики.

Утилита также принимает аргументы командной строки, используемые для установки соединения (см. 18.1).

Если конфигурация содержит какие-либо локальные дополнения в БД `template1`, необходимо быть внимательным при восстановлении в чистую БД, иначе можно получить ошибки при попытке повторного определения объектов. Для создания чистой БД без локальных добавлений ее необходимо создавать из шаблона БД `template0`, а не `template1`, например:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

Ограничения утилиты `pg_restore`:

1) при восстановлении данных в уже существующую таблицу и применении опции `--disable-triggers` `pg_restore` выполняет команды для отключения триггеров на таблицах перед вставкой в них данных и команд их включения после вставки. При остановке процесса восстановления посередине системный каталог может оказаться в несогласованном состоянии;

2) `pg_restore` не позволяет восстанавливать большие объекты по выбору, например только для отдельной таблицы. Если архив содержит большие объекты, только все они могут быть восстановлены или нет, при их исключении с помощью `-L`, `-t` или иных опций;

После восстановления рекомендуется выполнение команды ANALYZE для каждой таблицы для достижения лучшей производительности.

Примеры:

1. Создание резервной копии БД mydb в формате «custom»:

```
$ pg_dump -Fc mydb > db.dump
```

2. Удаление БД и воссоздание ее из резервной копии:

```
$ dropdb mydb
$ pg_restore -C -d postgres db.dump
```

БД, указанной в опции -d, может быть любая БД кластера. pg_restore использует ее только для выполнения команды CREATE DATABASE. С опцией -C данные всегда восстанавливаются в БД, указанную в резервной копии.

3. Загрузка резервной копии в новую БД newdb:

```
$ createdb -T template0 newdb
$ pg_restore -d newdb db.dump
```

Необходимо отметить, что опция -C не была использована, вместо этого осуществлялось подключение непосредственно к восстанавливаемой БД. Новая БД была создана из шаблона template0, а не template1, для обеспечения первоначальной чистоты базы.

4. Переопределение порядка элементов БД:

```
$ pg_restore -l db.dump > db.list
```

Результирующий файл листинга содержит заголовок и по одной строке на элемент, 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
```

Символ «;» в начале строки обозначает комментарий, число в начале строки — внутренний идентификатор, ассоциированный с элементом архива.

Строки в файле листинга могут быть закомментированы, удалены или перемещены, например:

```
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
```

Приведенный листинг может быть подан на вход утилиты `pg_restore` и определяет восстановление элементов 10-го и 6-го в указанном порядке:

```
$ pg_restore -L db.list db.dump
```

18.18. `psql` - интерактивный терминал

Для взаимодействия с PostgreSQL в интерактивном режиме используется утилита `psql`.

Утилита `psql` является интерактивным клиентом PostgreSQL и позволяет интерактивно набирать запросы, отправлять их серверу и получать результаты. Также ввод может осуществляться из файла. В дополнение утилита поддерживает метакоманды и некоторые возможности командной оболочки для облегчения создания скриптов и автоматизации широкого круга задач.

Синтаксис:

```
psql [option...] [dbname [username]]
```

Аргументы командной строки приведены в таблице 127.

Таблица 127

Аргумент	Описание
`-a` `--echo-all`	Выводить непустые входные строки в стандартный поток вывода в том виде, в каком они были получены. Это более удобно для обработки скриптов, чем при интерактивном режиме работы. Эквивалентно установке переменной `ECHO` в `all`.
`-A` `--no-align`	Переключиться в неформатированный вывод. (По умолчанию осуществляется форматированный вывод.)

Продолжение таблицы 127

Аргумент	Описание
`-c command` `--command command`	Выполнить одну командную строку и выйти. Это удобно для скриптов командной оболочки. `command` должно быть командной строкой, полностью понятной серверу (т. е. не должна содержать специфичных особенностей `psql`), или отдельной управляющей командой. Таким образом, не допускается смешивание SQL-команд и метакоманд `psql` при использовании указанной опции. Чтобы обойти это, существует возможность подачи строки на вход утилиты `psql` с помощью канала, например: `echo '\x \\ SELECT * FROM foo;' | psql` (`\\` являются разделителем метакоманд). Если командная строка содержит несколько SQL-команд, они обрабатываются в одной транзакции, так что требуется явное указание команд `BEGIN/COMMIT` для разбиения строки на несколько транзакций. Это отличается от поведения при подаче строки на стандартный вход `psql`.
`-d dbname` `--dbname dbname`	Указывает БД для соединения. Эквивалентно указанию БД в качестве первого неопционального аргумента командной строки при вызове утилиты. Если параметр содержит знак равенства `=` или начинается с корректного URI-префикса (`postgresql://` или `postgres://`), он рассматривается как строка соединения.
`-e` `--echo-queries`	Копировать посылаемые серверу SQL-команды в стандартный поток вывода. Эквивалентно установке переменной `ECHO` в `queries`.
`-E` `--echo-hidden`	Выводить запросы, сгенерированные командами `\d` или аналогичными. Может быть использовано для изучения внутренних операций `psql`. Эквивалентно установке переменной `ECHO_HIDDEN` в `on`.
`-f filename` `--file filename`	Использовать в качестве источника команд файл вместо интерактивного чтения команд. После выполнения содержимого `psql` завершает работу. Во многом это эквивалентно внутренней команде `\i`. Если параметр задан дефисом `-`, используется стандартный поток ввода. Использование этой опции несколько отличается от выполнения `psql < filename`. Оба варианта ведут себя так, как и ожидается, но использование `-f` предоставляет сообщения об ошибках с указанием номера строки. Также использование этой опции позволяет сократить временные затраты при запуске утилиты. С другой стороны, перенаправление ввода в командной оболочке гарантирует точно такой же результат, как если бы ввод осуществлялся вручную.
`-F separator` `--field-separator separator`	Использовать параметр в качестве разделителя при неформатированном выводе. Эквивалентно командам `\pset fieldsep` или `\f`.
`-H` `--html`	Включает табулированный вывод HTML. Эквивалентно командам `\pset format html` или `\H`.
`-l` `--list`	Вывести список доступных БД и выйти. Другие опции, кроме необходимых для установки соединения, игнорируются. Аналогично команде `\list`.
`-L filename` `--log-file filename`	Записывать все запросы в указанный файл в дополнение к обычному выводу.
`-n` `--no-readline`	Не использовать для редактирования строк возможности `readline` и не использовать историю выполненных команд. Может быть удобно для выключения автодополнения при выполнении операций копирования/вставки.

Окончание таблицы 127

Аргумент	Описание
`-o filename` `--output filename`	Записывать все результаты выполнения запросов в указанный файл. Эквивалентно команде `\o`.
`-P assignment` `--pset assignment`	Позволяет указать опции печати в стиле `\pset` в командной строке. Необходимо разделять имя и значение знаком равенства = вместо пробела. Так, при желании вывода в формате LaTeX необходимо писать `-P format=latex`.
`-q` `--quiet`	Указывает утилите `psql` не выводить сообщений в процессе работы. По умолчанию выводятся приглашения к вводу и множество информационных сообщений. Удобно при использовании совместно с опцией `-c`. Установить переменную `QUIET` в `on` для достижения того же эффекта.
`-R separator` `--record-separator separator`	Использовать указанный параметр в качестве разделителя полей при неформатированном выводе. Эквивалентно команде `\pset recordsep`.
`-s` `--single-step`	Запуск в пошаговом режиме. Означает, что пользователь будет спрошен перед выполнением каждой посылаемой серверу команды, с возможностью прервать исполнение. Используется для отладки скриптов.
`-S` `--single-line`	Запуск в построчном режиме, когда символ новой строки в SQL-команде действует как символ `;`.
`-t` `--tuples-only`	Выключает печать наименований столбцов и счетчика строк при выводе. Эквивалентно команде `\t`.
`-T table_options` `--table-attr table_options`	Указывать опции, помещаемые внутрь тэга HTML (описание команды `\pset`).

`-v assignment`
`--set assignment`
`--variable assignment`
	Присвоение значения переменной, аналогично внутренней команде `\set`. Необходимо отметить, что имя и значения переменной в командной строке должны быть разделены знаком равенства =. Для удаления переменной необходимо указать только ее имя без знака равенства. Для объявления переменной требуется указать знак равенства без указания значения. Присвоение переменных осуществляется на раннем этапе запуска, так что переменные, предназначенные для внутреннего использования, могут быть переопределены позднее.

`-x` `--expanded`	Включает режим расширенного форматирования таблиц. Эквивалентно команде `\x`.
`-X` `--no-psqlrc`	Не читать файл установок (как общесистемный, так и пользовательский файл `~/.psqlrc`).
`-z` `--field-separator-zero`	Устанавливает в качестве разделителя полей при неформатированном выводе нулевой байт.
`-0` `--record-separator-zero`	Устанавливает в качестве разделителя строк при неформатированном выводе нулевой байт. Может быть удобно для взаимодействия, например с `xargs -0`.
`-1` `--single-transaction`	При выполнении утилитой `psql` скрипта совместно с опцией `-f` добавление этой опции окружает скрипт командами `BEGIN/COMMIT` для выполнения его в одной транзакции. Это гарантирует, что либо все команды были завершены успешно, либо никакие изменения не были применены. Если скрипт внутри себя использует команды `BEGIN`, `COMMIT` или `ROLLBACK`, опция не позволит достигнуть желаемого эффекта. Также, если скрипт содержит команды, которые не могут быть исполнены внутри блока транзакции, указание этой опции может вызвать завершение команды (или даже всей транзакции) с ошибкой.

Утилита psql возвращает 0 в командную оболочку при нормальном завершении работы, 1 — при возникновении фатальной ошибки (недостаток памяти, файл не найден), 2 — соединение с сервером разорвалось и сессия потеряла ретроактивность, 3 — ошибка обнаружена в скрипте и установлена переменная ON_ERROR_STOP.

18.18.1. Соединение с базой данных

Утилита psql является обыкновенным клиентским приложением PostgreSQL. Для установки соединения требуется указание БД, имя и номер порта сервера и имя пользователя, под которым устанавливается соединение. Существует возможность указать эти параметры с помощью аргументов командной строки -d, -h, -p и -U, соответственно. Если аргумент не соответствует ни одной из опций, он воспринимается как имя БД (или имя пользователя, если БД уже было получено). Не все из перечисленных опций необходимы; существуют значения по умолчанию. При отсутствии указания имени сервера psql осуществляет соединение с помощью Unix-сокета с сервером на локальной ЭВМ, или по TCP/IP с локальной ЭВМ, не имеющей Unix-сокетов. Значение по умолчанию для номера порта задается при компиляции. Поскольку сервер использует такое же значение, во многих случаях указания номера порта не требуется. Имя пользователя по умолчанию соответствует имени пользователя операционной системы, и также определяется БД по умолчанию. Нельзя осуществить соединения с любой БД, используя произвольное имя пользователя. Администратор должен проинформировать о предоставленных правах доступа.

Если значения по умолчанию не верны, существует возможность их переопределения установкой переменных окружения PGDATABASE, PGHOST, PGPORT и/или PGUSER в соответствующие значения. Также удобно использовать файл ~/.pgpass для устранения необходимости регулярного ввода пароля.

Альтернативным путем задания параметров соединения является строка соединения, используемая вместо имени БД. Этот механизм предоставляет широкие возможности по управлению установкой соединения. Например:

```
$ psql "service=myservice sslmode=require"
$ psql postgresql://dbmaster:5433/mydb?sslmode=require
```

Таким же образом возможно использование LDAP для определения параметров соединения.

При невозможности установки соединения в силу тех или иных причин (например, недостаток прав доступа, сервер не запущен, и т. п.) утилита psql возвращает ошибку и завершает работу.

Если и входного или и выходной поток является терминалом, psql устанавливает кодировку клиента в "auto", для автоматического определения подходящей кодировки клиента из его локальных настроек (переменной окружения LC_CTYPE в Unix-системах).

Если это не работает нужным образом, кодировка клиента может быть переопределена с помощью переменной окружения PGCLIENTENCODING.

18.18.2. Ввод SQL-команд

При нормальном функционировании psql выводит приглашение с именем БД, с которой в настоящее время установлено соединение, за которым следует =>. Например:

```
$ psql testdb
psql (9.x.x)
Наберите "help" для справки.
testdb=>
```

После приглашения пользователь имеет возможность ввода SQL-команд. Обычно введенный запрос отсылается серверу после ввода завершающего символа «;». Перевод строки не завершает команду. Таким образом, команда может быть записана в несколько строк для лучшего восприятия. Если команда была отослана серверу и выполнена без ошибок, на экран выводится результат ее выполнения.

Также всякий раз при выполнении команды psql опрашивает сервер на наличие асинхронных событий, сгенерированных с помощью LISTEN и NOTIFY.

Если блоки комментариев в стиле языка C, переданные серверу на обработку, будут удалены, то стандартные SQL коментарии будут удалены клиентом psql.

18.18.3. Метакоманды

В случае ввода строки, начинающейся с не заключенного в кавычки символа \, она воспринимается как метакоманда и обрабатывается непосредственно утилитой psql. Подобные команды делают утилиту более удобной для администрирования и создания скриптов.

Формат команд psql состоит из символа \, за которым следует командное слово с последующими аргументами. Аргументы отделяются от командного слова и друг друга произвольным количеством пробельных символов.

Для включения пробельного символа в аргумент его необходимо заключать в одинарные кавычки. Для включения символа одинарной кавычки в подобный аргумент необходимо использовать два символа одинарной кавычки.

Кроме того, все, что заключено в одинарные кавычки, проверяется на подстановки в стиле C для \n (новая строка), \t (табуляция), \b (забой), \r (возврат каретки), \f (прогон страницы), \digits (восьмеричное значение) и \xdigits (шестнадцатеричное значение). В обрамленных одинарными кавычках строках символ \ экранирует следующий символ.

Аргументы, заключенные в апострофы (`), воспринимаются как командная строка и передаются командной оболочке. Результат выполнения команды (исключая возможные переводы строки вначале) берется в качестве аргумента. Описанные выше управляющие

последовательности также могут применяться в заключенных в апострофы аргументах.

Если не заключенный в кавычки аргумент начинается с двоеточия (:), он воспринимается как переменная psql и в качестве аргумента вместо имени используется ее значение.

Некоторые команды принимают SQL-идентификаторы (такие как имя таблицы) в качестве аргумента. Такие аргументы следуют правилам синтаксиса SQL: не заключенные в кавычки символы переводятся в нижний регистр, тогда как заключение в двойные кавычки предотвращает изменение регистра символов и позволяет использовать внутри идентификатора пробельные символы. Внутри двойных кавычек парные двойные кавычки заменяются на одну двойную кавычку в результирующем имени. Например, FOO"BAR"BAZ интерпретируется как fooBARbaz, а "A weird"" name" становится A weird" name.

Разбор аргументов заканчивается при обнаружении следующего не заключенного в кавычки символа \, что воспринимается как начало следующей метакоманды. Специальная последовательность \\ (два символа) отмечает завершение аргументов и дальше разбирается SQL-команда, если она указана. Таким образом SQL- и psql-команды могут быть смешаны в одной строке. Но в любом случае аргументы метакоманд не могут продолжаться после перевода строки.

Основные метакоманды приведены в таблице 128.

Таблица 128

Аргумент	Описание
\a	Если формат вывода текущей таблицы является не выровненный, он переключается в выровненный, и наоборот. Команда поддерживается в целях обратной совместимости. См. описание \pset для более общего и универсального случая.
\c (или \connect) [dbname [username] [host] [port]]	Устанавливает новое соединение с сервером PostgreSQL. Если соединение было установлено успешно, предыдущее соединение закрывается. Если какой-нибудь из аргументов опущен или указан как «-», используется значение от предыдущего соединения. В случае отсутствия предыдущего соединения используются значения по умолчанию. В случае если соединение не было установлено (неправильное имя пользователя, ошибка доступа и т. п.), предыдущее соединение сохраняется, если psql находился в интерактивном режиме. При выполнении неинтерактивного скрипта обработка немедленно прерывается с ошибкой. Это отличие в поведении обусловлено удобством пользователя для повторного ввода, с одной стороны, и обеспечением механизма защиты от случайного выполнения скрипта с неисправной БД, с другой стороны.
\cd [directory]	Меняет текущий рабочий каталог в directory. При указании без аргумента устанавливает в качестве рабочего каталога домашний каталог пользователя. Примечание. Для получения своего рабочего каталога возможно использование команды \!pwd.

Продолжение таблицы 128

Аргумент	Описание
`\C [ title ]`	Устанавливает заголовок для всех выводимых в результате выполнения запроса таблиц или отменяет подобные заголовки. Эквивалентно команде `\pset title title`. (Имя команды произошло от «caption» и использовалась изначально для установки заголовка HTML-таблицы.)
`\copy {table [ ( column_-list ) ] | ( query ) }{from | to }{'filename' | program 'command' | stdin | stdout | pstdin | pstdout }[ with ] ( option [, ...] ) ]`	Осуществляет копирование данных на стороне клиента. Операция выполняется как SQL-комнада `COPY`, но вместо чтения или записи сервером определенного файла `psql` читает или пишет файл и перенаправляет вывод между сервером и локальной ФС. Это означает, что применяются привилегии локального пользователя, а не сервера, и привилегии SQL суперпользователя не требуются. Если программа `program`, указанная команда `command` исполняется утилитой `psql` и данные, которые оправляются или получаются командой, перенаправляются серверу и клиенту СУБД. Это означает, что применяются привилегии локального пользователя, а не сервера, а также не требуются привилегии SQL суперпользователя. `\copy ... from stdin | to stdout` осуществляет чтение / запись в командные потоки ввода/вывода, соответственно. Все читаемые из источника строки передаются команде до обнаружения `\.` или конца потока (EOF). Вывод команды направляется в командный поток вывода. Для чтения/записи из стандартных потоков ввода/вывода утилиты `psql` используются `pstdin` или `pstdout`. Эта опция полезна при необходимости вывода таблиц в файл со скриптом. Синтаксис команды схож с SQL командой `COPY`, и параметр `option` должен задавать одну из опций команды `COPY`. Специальные правила разбора не применимы для команды `\copy`. В частности, не применяются правила подстановки переменных и управляющих последовательностей. Примечание. Данная операция не так эффективна как исходная команда `COPY`, поскольку данные между клиентом и сервером передаются через соединение. Для больших объемов данных предпочтительнее использование SQL-команды.
`\conninfo`	Выводит информацию о текущем соединении с БД.
`\copyright`	Отображение информации об авторских правах и условий распространения PostgreSQL.

Продолжение таблицы 128

Аргумент	Описание
`\d[S+] [ pattern ]`	Для каждого отношения (таблицы, вида, индекса или последовательности), совпадающего с шаблоном, отображаются столбцы, их типы и табличные пространства (если они не стандартны) и специальные атрибуты, такие как `NOT NULL` или значения по умолчанию. Ассоциированные индексы, ограничения и триггеры так же отображаются, как и определение вида для видов. Для внешних таблиц выводится ассоциированный с ними внешний сервер. Для некоторых типов отношений команда отображает дополнительную информацию для каждого из столбцов: значение для последовательностей, индексное выражение для индексов, опции обработчика внешних данных для внешних таблиц. Форма команды `\d+` идентична, за исключением количества выводимой информации: дополнительно отображаются комментарии столбцов таблицы, наличие OID, определение представлений при параметре `replica identity`, установленному не по умолчанию. По умолчанию отображаются только пользовательские объекты; для распространения шаблона и на системные объекты применяется модификатор `S`. Примечание. Если `\d` используется без указания шаблона, это эквивалентно команде `\dtvsE`, отображающей все таблицы, виды, последовательности и внешние таблицы.
`\da[S] [ pattern ]`	Отображает все доступные агрегатные функции с указанием типа результата и операндов. При указании шаблона отображаются только совпадающие с ним. По умолчанию отображаются только пользовательские объекты; для распространения шаблона и на системные объекты применяется модификатор `S`.
`\db[+] [ pattern ]`	Отображает все доступные табличные пространства. При указании шаблона отображаются только совпадающие с ним. При указании в команде «+» объекты выводятся с правами доступа.
`\dc[S+] [ pattern ]`	Отображает все доступные преобразования символов между кодировками. При указании шаблона отображаются только совпадающие с ним. По умолчанию отображаются только пользовательские объекты; для распространения шаблона и на системные объекты применяется модификатор `S`. При указании в команде «+» объекты выводятся с описаниями.
`\dC[+] [ pattern ]`	Отображает все доступные приведения типов. При указании шаблона отображаются только те, чьи исходные или целевые типы совпадают с ним. При указании в команде «+» объекты выводятся с описаниями.
`\dd[S] [ pattern ]`	Отображает описания объектов типов `constraint`, `operator class`, `operator family`, `rule` и `trigger`. Описания объектов других типов могут быть просмотрены соответствующими командами для этих типов. Отображает описания объектов, совпадающих с шаблоном, или всех объектов, если он не указан. В любом случае отображаются только имеющие описание объекты. По умолчанию отображаются только пользовательские объекты; для распространения шаблона и на системные объекты применяется модификатор `S`. Описания объектов могут быть созданы SQL-командой `COMMENT`.

Продолжение таблицы 128

Аргумент	Описание
`\ddp [ pattern ]`	Отображает настройки доступа по умолчанию. Выводится информация для каждой роли или схемы (если применимо), для которой настройки доступа были изменены от заданных при сборке. Если шаблон не задан выводится информация только по ролям и схема, точно совпадающих с заданным именем. Для задания настроек доступа по умолчанию используется команда `ALTER DEFAULT PRIVILEGES`.
`\dD[S+] [ pattern ]`	Отображает все доступные домены. При указании шаблона отображаются только совпадающие с ним. По умолчанию отображаются только пользовательские объекты; для распространения шаблона и на системные объекты применяется модификатор `S`. При указании в команде «+» объекты выводятся с описаниями и правами доступа.

`\dE[S+] [ pattern ]`
`\di[S+] [ pattern ]`
`\dm[S+] [ pattern ]`
`\ds[S+] [ pattern ]`
`\dt[S+] [ pattern ]`
`\dv[S+] [ pattern ]`
	В этой группе команд символы `E`, `i`, `m`, `s`, `t` и `v` обозначают внешние таблицы (External tables), индексы (index), материализованные представления (materialized view), последовательности (sequence), таблицы (table) и виды (view), соответственно. Возможно указание любых из этих символов в любой последовательности для получения списка соответствующих объектов. Например, `\dit` отображает индексы и таблицы. При добавлении символа «+» каждый объект выводится с приведением занимаемого места на диске и описанием (при его наличии). По умолчанию отображаются только пользовательские объекты; для распространения шаблона и на системные объекты применяется модификатор `S`.

`\des[+] [ pattern ]`	Отображает все внешние серверы. При указании шаблона отображаются только совпадающие с ним. При использовании формы `\des+` для каждого выводятся описания, включая списки контроля доступа, типы, версии, параметры и описание.
`\deu[+] [ pattern ]`	Выводит все отображения пользователей («внешних пользователей»). При указании шаблона отображаются только совпадающие с ним. При использовании формы `\deu+` для каждого пользователя выводится дополнительная информация. **ВНИМАНИЕ!** `\deu+` может также отображать имя и пароль внешнего пользователя, так что использование этой формы требует осторожности.
`\dew[+] [ pattern ]`	Отображает обработчики внешних данных. При указании шаблона отображаются только совпадающие с ним. При использовании формы `\dew+` для каждого выводятся списки контроля доступа, параметры и описание.
`\df[antwS+] [ pattern ]`	Отображает все доступные, функции с указанием их аргументов, типов результата и типа самой функции: 'agg' (агрегат), 'normal', 'trigger' и 'window'. Для вывода функций конкретного типа используются соответствующие буквы `a`, `n`, `t` или `w`. При указании шаблона отображаются только совпадающие с ним. При использовании формы `\df+` для каждой функции выводится дополнительная информация, включая тип изменчивости (volatility), процедурный язык, исходный код и описание. По умолчанию отображаются только пользовательские объекты; для распространения шаблона и на системные объекты применяется модификатор `S`. Для поиска функции, принимающей аргументы или возвращающей результат конкретного типа, необходимо использовать возможности функции поиска средства просмотра вывода `\df`.

Продолжение таблицы 128

Аргумент	Описание
`\dF[+] [ pattern ]`	Отображает доступные конфигурации текстового поиска. При указании шаблона отображаются только совпадающие с ним. При использовании формы `\dF+` для каждой выводится полное описание, включающее используемый синтаксический анализатор и словарный список для каждого типа токена.
`\dFd[+] [ pattern ]`	Отображает доступные словари текстового поиска. При указании шаблона отображаются только совпадающие с ним. При использовании формы `\dFd+` для каждого выводится дополнительная информация, включающая используемый шаблон тестового поиска и значения параметров.
`\dFp[+] [ pattern ]`	Отображает доступные синтаксические анализаторы текстового поиска. При указании шаблона отображаются только совпадающие с ним. При использовании формы `\dFp+` для каждого выводится полное описание, включающее используемые функции и список распознаваемых токенов.
`\dFt[+] [ pattern ]`	Отображает доступные шаблоны текстового поиска. При указании аргумента отображаются только совпадающие с ним. При использовании формы `\dFt+` для каждого выводится полное описание, включающее используемые функции.
`\dg[+] [ pattern ]`	Отображает роли. При указании аргумента отображаются только совпадающие с ним (эта команда более эффективна чем `\du`). При указании шаблона отображаются только совпадающие с ним. При указании в команде «+» объекты выводятся с описаниями.
`\dl`	Псевдоним для команды `\lo_list`, выводящей список больших объектов.
`\dL[S+] [ pattern ]`	Отображает список процедурных языков. При указании шаблона отображаются только совпадающие с ним. По умолчанию отображаются только пользовательские объекты; для распространения шаблона и на системные объекты применяется модификатор `S`. При указании в команде «+» для каждого процедурного языка дополнительно выводится обработчик, функция проверки, права доступа и признак системного языка.
`\dn[S+] [ pattern ]`	Отображает доступные схемы. При указании шаблона отображаются только совпадающие с ним. По умолчанию отображаются только пользовательские объекты; для распространения шаблона и на системные объекты применяется модификатор `S`. При добавлении символа «+» каждая схема выводится с правами доступа и описанием (при его наличии).
`\do[S] [ pattern ]`	Отображает доступные операторы с указанием операндов и типов результата. При указании шаблона отображаются только совпадающие с ним. По умолчанию отображаются только пользовательские объекты; для распространения шаблона и на системные объекты применяется модификатор `S`.
`\dO[S+] [ pattern ]`	Отображает способы сортировки (collation). При указании шаблона отображаются только совпадающие с ним. По умолчанию отображаются только пользовательские объекты; для распространения шаблона и на системные объекты применяется модификатор `S`. При указании в команде «+» для каждого способа сортировки выводится описание. Отображаются только способы сортировки, применимые к кодировке БД.

Продолжение таблицы 128

Аргумент	Описание
`\dp [ pattern ]`	Отображает доступные таблицы, виды и последовательности с правами доступа. При указании шаблона отображаются только совпадающие с ним. Права доступа назначаются или отбираются командами `GRANT` и `REVOKE`.
`\drds [ role-pattern [ database-pattern ] ]`	Отображает заданные конфигурационные настройки. Настройки могут быть заданы для роли, БД или для сочетания обоих. Шаблоны `role-pattern` и `database-pattern` задают попадающие в список роли и БД соответственно. Если шаблоны не указаны или указан символ `*`, выводятся все настройки. Для изменения специальных конфигурационных настроек для роли или БД используются команды `ALTER ROLE` и `ALTER DATABASE`, соответственно.
`\dT[S+] [ pattern ]`	Отображает доступные типы данных. При указании шаблона отображаются только совпадающие с ним. При добавлении символа «+» каждый тип выводится с внутренним именем и размером, и списком допустимых значений для перечислений. По умолчанию отображаются только пользовательские объекты; для распространения шаблона и на системные объекты применяется модификатор `S`.
`\du[+] [ pattern ]`	Отображает роли. При указании шаблона отображаются только совпадающие с ним. При указании в команде «+» объекты выводятся с описаниями.
`\dx[+] [ pattern ]`	Отображает установленные расширения (extension). При указании шаблона отображаются только совпадающие с ним. При указании в команде «+» для каждого расширения выводятся принадлежащие ему объекты.
`\dy[+] [ pattern ]`	Отображает событийные триггеры (event trigger). При указании шаблона отображаются только совпадающие с ним. При указании в команде «+» для каждого объекта выводится описание.
`\e (или \edit) [ filename ] [ line_number ]`	При указании файла вызывается редактор для его редактирования; после редактирования его содержимое помещается в буфер запросов. При отсутствии аргумента текущее содержимое буфера запросов копируется во временный файл, который редактируется аналогичным образом. Новый буфер запросов заново разбирается в соответствии с правилами `psql`, трактующих весь буфер как одну строку. (Таким образом нельзя сделать скрипт. Для этого может быть использовано `\i`.) Это означает, что, если запрос завершается (а часто и содержит) символ «;», он будет немедленно выполнен. До этого запрос будет просто находиться в буфере. При задании `line_number` курсор позиционируется в соответствующую строку файла или буфера. Если команде задан единственный числовой аргумент, он рассматривается как номер строки, а не как имя файла. Примечание. Используемый редактор определяется стандартными для используемой командной оболочки переменными окружения.

Продолжение таблицы 128

Аргумент	Описание

`\echo text [ ... ]`	Осуществляет печать аргументов в стандартный поток вывода, разделенных пробельными символами и завершающимися новой строкой.
	Это может быть полезно для представления информации на выходе скрипта в более удобном виде. Например:
	`=> \echo 'date'`
	`Tue Oct 26 21:40:57 CEST 1999`
	Если первый аргумент является не заключенным в кавычки `-n`, строка после вывода не переводится.
	При использовании команды `\o` для перенаправления вывода необходимо использовать вместо этой команду `\qecho`.

`\ef [ function_description ] [ line_number ]`	Команда извлекает и редактирует определение указанной функции в форме команды `CREATE OR REPLACE FUNCTION`. Редактирование осуществляется аналогично команде `\edit`. После выхода из редактирования обновленная команда помещается в буфер запросов; после этого необходимо ввести символ «;» или выполнить `\g` для отправки ее, `\r` — для отмены. Функция может быть указана именем или именем с аргументами, например `foo(integer, text)`. Аргументы должны быть указаны, если существует несколько функций с таким именем. Если функция не указана, для редактирования предоставляется пустой шаблон `CREATE FUNCTION`. При задании `line_number` курсор позиционируется в соответствующую строку файла или буфера. Если команде задан единственный числовой аргумент, он рассматривается как номер строки, а не как имя файла. Примечание. Используемый редактор определяется стандартными для используемой командной оболочки переменными окружения.
`\encoding [ encoding ]`	Устанавливает кодировку символов клиента. Без указания аргумента отображает текущую кодировку.
`\f [ string ]`	Устанавливает разделить столбцов при невыровненном выводе. По умолчанию используется вертикальная черта (|). См. описание команды `\pset` для более общего и универсального способа задания параметров вывода.
`\g [ { filename | |command } ]`	Посылает содержимое буфера запросов серверу и опционально сохраняет результат в файл или канал для вывода в отдельно выполняющуюся команду командной оболочки. Команда `\g` виртуально эквивалентна символу «;». Команда `\g` с аргументом является «мгновенной» альтернативой команде `\o`.

Продолжение таблицы 128

Аргумент	Описание

`\gset [ prefix ]`	Посылает содержимое буфера запросов серверу и сохраняет результат в указанной переменной `psql` (см. 18.18.4.1). Выполняемый запрос должен возвращать только одну строку результата. Каждый столбец возвращаемого результата помещается в отдельную переменную с именем, совпадающим с именем столбца. Например:
	`=> SELECT 'hello' AS var1, 10 AS var2`
	`-> \gset`
	`=> \echo :var1 :var2`
	`hello 10`
	При указании префикса `prefix`, указанная строка добавляется к имени столбца для формирования имени переменной:
	`=> SELECT 'hello' AS var1, 10 AS var2`
	`-> \gset result_`
	`=> \echo :result_var1 :result_var2`
	`hello 10`
	Если значение столбца `NULL`, соответствующая переменная будет не установлена.
	Если запрос выполняется с ошибкой или возвращает более одной строки, значения переменных не изменяются.

`\h (или \help) [ command ]`	Предоставляет справку по синтаксису указанной SQL-команды. Если команда не указана, выводится список команд, по которым существует справка. Если в качестве команды указан символ `*`, выводится справка по всем SQL-командам. Для простоты ввода состоящие из нескольких слов команды могут не заключаться в кавычки, т. е. можно написать `\help alter table`.
`\H`	Включение вывода в виде HTML. Если вывод уже в HTML, осуществляется переключение обратно к формату выровненного текста по умолчанию.
`\i filename` или `\include filename`	Читает из указанного файла и выполняет его содержимое так, как если бы оно было набрано вручную. Примечание. Чтобы видеть на экране содержимое включаемого файла, необходимо установить значение переменной `ECHO` в значение `all`.
`\ir filename` или `\include_relative filename`	Команда `ir` аналогична `i`, но отличается по поведению для относительных путей. В интерактивном режиме различие между командами отсутствует, но при исполнении скрипта команда считает имена относительно каталога исходного скрипта, а не относительно текущего рабочего каталога.
`\l[+] or \list[+] [ pattern ]`	Перечисляет БД на сервере и выводит для них имена, владельцев, кодировки и права доступа. При указании шаблона отображаются только совпадающие с ним. При указании символа «+» дополнительно выводятся размеры, табличные пространства по умолчанию и описания. (Информация о размере доступна только для БД, к которым пользователь может установить соединение.)

Продолжение таблицы 128

Аргумент	Описание
`\lo_export loid filename`	Читает из БД большой объект с идентификатором `loid` и записывает его в файл `filename`. Это несколько отличается от серверной функции `lo_export`, которая использует привилегии пользователя, от имени которого запущен сервер, для доступа к ФС на сервере. Для получения идентификатора большого объекта используется команда `\lo_list`.

`\lo_import filename [ comment ]`	Сохраняет содержимое указанного файла в большом объекте PostgreSQL. Дополнительно может указываться описание объекта. Например:
	`foo=> \lo_import '/home/peter/pictures/photo.xcf'`
	`    'a picture of me'`
	`lo_import 152801`
	Ответ от сервера содержит идентификатор загруженного объекта 152801, который может быть использован для доступа в дальнейшем. Идентификаторы и описания объектов могут быть получены с помощью команды `\lo_list`.
	Это несколько отличается от серверной функции `lo_import`, поскольку выполняется от имени локального пользователя на локальной ФС, а не от имени серверного пользователя на серверной ФС.

`\lo_list`	Отображает список больших объектов PostgreSQL, сохраненных в БД, с их описаниями.
`\lo_unlink loid`	Удаляет большой объект с указанным идентификатором из БД. Для получения идентификатора большого объекта используется команда `\lo_list`.
`\o [ {filename | |command } ]`	Сохраняет результаты последующих запросов в файл или канал для выполнения внешней обработки в командной оболочке Unix. Если аргумент не указан, вывод результатов запросов возвращается в стандартный поток вывода. Результаты запросов включают в себя все таблицы, ответы и сообщения, получаемые с сервера БД, а также результаты выполнения метакоманд (таких как `\d`), опрашивающих БД, но исключая сообщения об ошибках. Для оформления текстового вывода среди результатов запроса используется команда `\qecho`.
`\p` или `\print`	Печать текущего содержимого буфера запросов в стандартный поток вывода.
`\password [ username ]`	Смена пароля указанного пользователя (по умолчанию — текущего). Команда запрашивает новый пароль, шифрует его и посылает серверу командой `ALTER ROLE`. Это гарантирует, что новый пароль не появится в раскрытом текстовом виде в истории команд, журналах сервера или где еще.
`\prompt [ text ] name`	Запрашивает у пользователя значение заданной переменной. Дополнительно может быть указан текст подсказки. (Многословные подсказки должны быть заключены в одинарные кавычки.) По умолчанию `\prompt` использует для ввода/вывода терминал. Однако при указании опции `-f` в командной строке `\prompt` использует стандартные потоки ввода/вывода.

Продолжение таблицы 128

Аргумент	Описание
`\pset option [ value ]`	Команда устанавливает параметры, влияющие на вывод результатов запросов. `option` указывает, какой именно параметр устанавливается. От него зависит и смысл значения. Для некоторых параметров опускание значения приводит к переключению параметра или отмене его определения. Если не указано специальное поведение при опускании значения, отображается текущее состояние параметра. `\pset` без аргументов выводит текущий статус всех настроек печати. Существуют следующие параметры выравнивания: - `border` — второй аргумент должен быть числом. Чем больше число, тем больше рамки и линии у таблиц, но это зависит еще от конкретного формата. В формате HTML значение транслируется непосредственно в атрибут `border=`, в других случаях применяются только значения `0` (отсутствие рамки), `1` (внутренние разделяющие линии) и `2` (рамка таблицы). `latex` и `latex-longtable` поддерживают значение `3`, при котором добавляется разделитель между строк; - `columns` — задает ширину для формата `wrapped` и ширину для определения того, требуется ли средство прокрутки вывода. Если параметр установлен в `0` (по умолчанию), режим влияет только на вывод на экран и учитывает значение переменной окружения `COLUMNS` или выявленную ширину экрана. Если параметр `\pset columns` установлен не в нулевое значение, весь вывод сворачивается, включая вывод в файл и канал; - `expanded` (или `x`) — указание второго аргумента, который может принимать значения `on` или `off`, что включает или выключает расширенный режим, и `auto`. Если второй аргумент не указан, осуществляется переключение между обычным режимом и расширенным. При включении расширенного режима результаты запросов отображаются в двух столбцах с указанием имени столбца слева и выводом данных справа. Режим полезен, когда данные не помещаются на экране в обычном «горизонтальном» режиме. Значение `auto` действует только для режимов `aligned` и `wrapped`, в остальных это равноценно `off`; - `fieldsep` — задает разделитель столбцов для формата вывода `unaligned`. Таким способом можно получить, к примеру, разделенный табуляцией или запятой вывод, предпочтительный для других программ. Для установки в качестве разделителя табуляции необходимо задать `\pset fieldsep '\t'`. Разделителем по умолчанию является символ `|` (вертикальная черта); - `fieldsep_zero` — задает разделитель столбцов для формата вывода `unaligned` в нулевой символ; - `footer` — указание второго аргумента, который может иметь значения `on` или `off`, что включает или выключает отображение нижнего колонтитула таблицы по умолчанию (`x` строк). При отсутствии второго аргумента осуществляется переключение между этими режимами;

Продолжение таблицы 128

Аргумент	Описание
	- `format` — устанавливает один из заданных форматов вывода, таких как `unaligned`, `aligned`, `wrapped`, `HTML`, `latex`, `latex-longtable` или `troff-ms`. Также принимаются уникальные сокращения. (Достаточно одной буквы.) `unaligned` выводит все столбцы строки в одну линию, разделенную текущим разделителем столбцов. Это предполагает создание результата, пригодного для дальнейшего чтения другими программами (разделение табуляцией, запятой); `aligned` является стандартным, удобным для восприятия, форматированным тестовым выводом и используется по умолчанию. `wrapped` похож на формат `aligned`, но сворачивает вывод до заданной ширины. Ширина определяется описанным ранее параметром `columns`. Следует отметить, что `psql` не сворачивает заголовки столбцов. Поэтому формат ведет себя так же, как `aligned`, если общая ширина необходимая для заголовков столбцов превышает заданную. `HTML`, `latex`, `latex-longtable` и `troff-ms` выводят таблицы в виде, предполагающем их последующие использование в документах соответствующих языков разметки. Результаты не являются законченными документами! (Это может быть не так страшно для `HTML`, но для `latex` обязательна обработка для получения законченного документа. Для `latex-longtable` также необходимы LaTeX пакеты `longtable` и `booktabs`); - `linestyle` — задает стиль отрисовки линий в `ascii`, `old-ascii` или `unicode`. Та же принимаются уникальные сокращения. (Достаточно одной буквы.) По умолчанию используется `ascii`. Действует только для режимов `aligned` и `wrapped`. `ascii` использует обычные ASCII-символы. Перевод строки в данных отображается символом + с правым выравниванием. При переносе строк в случае выравнивания, отображается символ (`.`) с правым выравниванием в первой строке и левым в последующих. `old-ascii` использует обычные ASCII-символы в стиле, принятом в версиях PostgreSQL 8.4 и ранее. Перевод строки в данных отображается символом `:` с левым выравниванием. При переносе строк в случае выравнивания, отображается символ (`;`) с левым выравниванием. `unicode` использует Unicode-символы для отрисовки рамок. Перевод строки в данных отображается символом «возврат каретки» с правым выравниванием. При переносе строк в случае выравнивания, отображается многоточие с правым выравниванием в первой строке и левым в последующих; - `null` — второй аргумент является строкой, которая выводится в том случае, если данные в столбце имеют неопределенной значение. По умолчанию не выводится ничего, что может вызывать ошибку восприятия, например, пустых строк. В этом случае можно выбрать вариант `\pset null '(null)'`; - `numericlocale` — указание второго аргумента, который может иметь значения `on` или `off`, что включает или выключает отображение зависящих от локализации символов разделения групп разрядов чисел слева от запятой. При отсутствии второго аргумента осуществляется переключение между этими режимами;

Продолжение таблицы 128

Аргумент	Описание
	- pager — управляет использованием средств прокрутки для просмотра результатов запроса и справки программы psql. Если установлена переменная окружения PAGER, результаты передаются с помощью конвейера указанной программе. Другими словами, используется специфичная для платформы программа (подобная more). Если параметр не установлен, средство прокрутки не используется. При установленном применяется только там, где это применимо, т. е. при выводе на терминал, когда результат не вмещается на экран. \pset pager переключает между режимами on и off. Также может быть установлено значение always, при котором средство прокрутки используется всегда; - recordsep — задает разделитель строк при использовании формата вывода unaligned. Значением по умолчанию является новая строка; - recordsep_zero — задает разделитель строк при использовании формата вывода unaligned в нулевой символ; - tableattr (или T)[text] — позволяет указать некоторые атрибуты, помещаемые в тэг table в режиме «HTML». К примеру это могут быть атрибуты cellpadding или bgcolor. Можно не задавать таким образом рамку таблицы, т. к. это делается командой \pset border. В формате latex-longtable задает пропорциональную ширину для столбцов с левым выравниванием. Значения указываются списком, например '0.2 0.2 0.6'. Столбцы, не заданные в списке, используют последнее заданное значение; - title [text] — задает заголовок для вывода последующих таблиц. Может быть использовано для получения описательных меток в выводе. При отсутствии аргумента заголовок отключается; - tuples_only (or t) — указание второго аргумента, который может иметь значения on или off, что включает или выключает режим «только данные». При отсутствии второго аргумента осуществляется переключение между этими режимами. При полном отображении выводится дополнительная информация, такая как заголовки столбцов, таблиц и возможные колонтитулы. В режиме «только данные» отображаются только данные. Существует набор различных сокращенный команд для \pset — см. \a, \C, \H, \t, \T и \x.
\q или \quit	Завершение работы утилиты psql.
\qecho text [...]	Команда идентична команде \echo, за исключением того, что осуществляет вывод в канал вывода запросов, определенный командой \o.
\r или \reset	Очищает буфер запросов.
\s [filename]	Отображает или сохраняет в файл историю команд. Если имя файла не указано, история выводится в стандартный поток вывода. Команда доступна, только если утилита psql сконфигурирована для использования библиотеки GNU Readline.

Продолжение таблицы 128

Аргумент	Описание
\set [name [value [...]]]	Устанавливает внутренней переменной указанное значение, или если указано несколько, то их конкатенацию. Если не указан второй аргумент, переменная объявляется без значения. Для удаления переменной используется команда \unset. Команда \set без аргументов отображает имена и значения переменных, установленных к этому моменту в psql. Имя переменной может содержать символы, цифры и знак подчеркивания. Имена переменных чувствительны к регистру символов. Возможна установка любых переменных, некоторые из них psql трактует специальным образом. Примечание. Данная команда полностью отлична от SQL-команды SET.
\setenv [name [value]]	Устанавливает новое значение указанной переменной окружения. Если значение не задано, переменная окружения удаляется. Например: testdb=> \setenv PAGER less testdb=> \setenv LESS -imx4F
\sf[+] function_description	Отображает определение именованной функции в форме команды CREATE OR REPLACE FUNCTION. Описание выводится в текущий канал вывода, заданный параметром \o. Функция может быть задана как только именем, так и именем с аргументами, например foo(integer, text). Если существует несколько функций с одинаковыми именами, должны быть указаны типы аргументов. При указании в команде символа «+», выводятся номера строк, начиная с 1.
\t	Переключает режим вывода имен столбцов и нижнего колонтитула таблиц. Команда эквивалентна \pset tuples_only.
\T table_options	Позволяет задать атрибуты, помещаемые в тэг table в режиме вывода HTML. Команда эквивалентна \pset tableattr table_options.
\timing [on | off]	Без параметра переключает отображение времени выполнения запросов в миллисекундах. При задании аргумента параметр устанавливается в заданное значение.
\unset name	Отменяет установку (удаляет) указанную переменную psql.
\w { filename [| command] } или \write { filename [| command] }	Выводит содержимое буфера запросов в файл или канал конвейера Unix.
\watch [seconds]	Повторение текущего буфера запроса (аналогично \g) до прерывания или ошибки. Ожидает указанное количество (по умолчанию 2) между повторами.
\x [on | off | auto]	Переключает расширенный режим вывода. Эквивалентно команде \pset expanded.
\z [pattern]	Отображает список доступных таблиц, видов и последовательностей с правами доступа. При указании шаблона выводятся только совпадающие с ним. Для предоставления и отбора прав доступа используются команды GRANT и REVOKE. Является псевдонимом для \dp (display privileges).

Окончание таблицы 128

Аргумент	Описание
\! [command]	Выполняет отдельный сеанс командной оболочки Unix или команду Unix. Аргументы никак не интерпретируются, а передаются непосредственно командной оболочке.
\?	Отображает справку по метакомандам.

18.18.3.1. Шаблоны

Различные команды \d принимают в качестве аргумента шаблон, определяющий имена отображаемых объектов. В простейшем случае шаблон представляет собой точное имя объекта. Символы шаблона приводятся к нижнему регистру, как и SQL-имена; к примеру, \dt FOO отобразит таблицы с именем foo. Как и в SQL заключение в двойные кавычки предотвращает изменение регистра символов. При необходимости использования двойной кавычки в шаблоне она записывается дважды с заключением всего шаблона в двойные кавычки; что соответствует правилам, применяемым к SQL-идентификаторам. Например, \dt "FOO""BAR" отобразит таблицу FOO"BAR (а не foo"bar). В отличие от стандартных правил, для SQL-имен возможно заключение в кавычки только части шаблона, например \dt FOO"FOO"BAR отобразит таблицу fooFOObar.

Во всех случаях, когда параметр шаблона полностью опущен, команды \d выводят все объекты, видимые по текущему пути поиска в схемах, что эквивалентно использованию шаблона *. Для получения всех объектов в БД необходимо указать *.*.

Символ * в шаблоне обозначает любую последовательность символов (включая их отсутствие), а символ ? обозначает одиночный символ. (Такая нотация совместима с шаблонами имен файлов в Unix.) Например, \dt int* отображает все таблицы, чьи имена начинаются на int. При этом внутри двойных кавычек символы * и ? теряют свое специальное значение и воспринимаются буквально.

Шаблон, содержащий точку, интерпретируется как имя схемы с последующим именем объекта. Например, \dt foo*.*bar* отображает все таблицы, чьи имена содержат bar и принадлежат схемам с именами, начинающимися на foo. При отсутствии точки шаблон сравнивается только с объектами, доступными по текущему пути поиска объектов. Точка также, находясь внутри двойных кавычек, теряет свое специальное значение и воспринимается буквально.

Опытные пользователи могут использовать нотации регулярных выражений, такие как классы символов, к примеру [0–9] означает любую цифру. При этом действуют все специальные символы регулярных выражений (см. 6.7.3), за исключением точки, которая воспринимается в качестве разделителя. Символ * трактуется как нотация .* регулярных выражений, ? транслируется в ., а $ воспринимается буквально. При необходимости существует возможность эмуляции указанных шаблонов заданием ? для ., (R+|) для R* или

(R|) для R?. $ не требуется в регулярных выражениях, т. к. шаблон должен означать имя целиком, в отличие от обычной интерпретации регулярных выражений ($ автоматически добавляется к шаблону). Написание * в начале и/или в конце позволяет задать плавающий шаблон. Необходимо отметить, что находясь внутри двойных кавычек, все специальные символы регулярных выражений теряют свое специальное значение и воспринимаются буквально. Также специальные символы регулярных выражений воспринимаются буквально в шаблонах имен операторов (т. е. аргументе команды \do).

18.18.4. Расширенные возможности

18.18.4.1. Переменные

Утилита psql предлагает подстановку значений переменных, сходную с принятой в командной оболочке Unix. Переменные представляют собой простые пары «имя–значение», где «значение» может быть любой строкой любой длины. Имена внутренних переменных psql могут содержать символы, числа и знак подчеркивания в любом сочетании.

Для установки значения переменной используется метакоманда \set:

```
testdb=> \set foo bar
```

Будет установлено значение переменной foo в bar. Для получения значения переменной перед именем переменной ставится двоеточие и указывается аргумент любой метакоманды:

```
testdb=> \echo :foo
bar
```

При вызове \set без второго аргумента, переменная создается с пустой строкой в качестве значения. Для удаления переменной используется команда \unset.

Примечание. Аргументы команды \set обрабатываются по тем же правилам, как и у других команд. Это позволяет создавать ссылки типа \set :foo 'something' и получать «ссылку на» или «переменные переменные» в стиле Perl или PHP, соответственно. С другой стороны, \set bar :foo является хорошим способом копирования значения переменной.

Некоторые из переменных трактуются специальным образом. Они обозначают некоторые параметры, которые могут быть изменены в процессе работы путем установки значений переменных, или отражают некоторое состояние приложения. Для удобства все специальные переменные состоят из прописных букв (чисел и знака подчеркивания). В целях совместимости не рекомендуется использовать имена, совпадающие с именами специальных переменных, в собственных целях.

Список специальных переменных приведен в таблице 129.

Таблица 129

Переменная	Описание
AUTOCOMMIT	Во включенном состоянии on (по умолчанию) каждая SQL-команда автоматически подтверждается после успешного выполнения. Для отсрочки подтверждения в этом режиме требуется явное задание команд BEGIN или START TRANSACTION. В выключенном состоянии off SQL-команды не подтверждаются до тех пор, пока не будут явно указаны команды COMMIT или END. Этот режим осуществляет неявное выполнение команды BEGIN только перед теми командами, которые не находятся в блоке транзакции и перед которыми не были выполнены вручную команда BEGIN или иная команда управления транзакциями, но не перед теми, которые не могут выполняться внутри транзакции (например, VACUUM). Примечания: 1. В режиме отсутствия автоматического подтверждения транзакций любая ошибочная транзакция может быть прервана командами ABORT или ROLLBACK. При выходе из сессии без подтверждения транзакции неподтвержденные изменения будут потеряны. 2. Режим автоматического подтверждения транзакций является традиционным поведением для PostgreSQL, режим отсутствия автоматического подтверждения транзакций — для спецификации SQL. Значение этого параметра может быть задано в общесистемном файле psqlrc или в пользовательском файле ~/.psqlrc.
COMP_KEYWORD_CASE	Определяет регистр символов для завершения ключевых слов SQL. При установке в lower или upper слово будет завершаться в нижнем или верхнем регистре, соответственно. При значении preserve-lower или preserve-upper (по умолчанию), слово будет завершатся в том регистре, в каком уже вводится, а слова, которые еще не начинали вводить, будут в нижнем или верхнем регистре, соответственно.
DBNAME	Имя БД, с которой установлено соединение. Значение устанавливается каждый раз при установке соединения с БД (включая запуск программы) и не может быть сброшено.
ECHO	При установке в all все строки, вводимые с клавиатуры или из скрипта, выводятся в стандартный поток вывода перед их анализом или выполнением. Для получения такого поведения программы при запуске используется аргумент командной строки -a. При установке в queries psql выводит только посылаемые серверу запросы. Аргумент командной строки для этого — -e.
ECHO_HIDDEN	При установке этой переменной при выполнении метакомандами запросов к БД они отображаются перед выполнением. Таким образом предоставляется возможность изучить особенности PostgreSQL и добавить похожую функциональность в собственные программы. (Для получения подобного поведения при запуске используется аргумент командной строки -E.) При установке параметра в noexec запросы только отображаются, но не отсылаются и не выполняются на сервере.
ENCODING	Текущая клиентская кодировка символов.
FETCH_COUNT	Если переменная установлена в целое значение больше нуля, результаты запросов SELECT выбираются и отображаются группами по указанному количеству строк вместо получения всего набора данных результата перед отображением. Таким образом, используется ограниченное количество памяти, независимо от количества строк во всем результирующем наборе данных. При использовании этой функции обычно применяются значения от 100 до 1000. Необходимо также помнить, что при ее использовании запрос может завершиться неудачно уже после отображения некоторого количества строк. Указанный параметр может быть использован при любом формате вывода, формат по умолчанию — aligned — может выглядеть плохо, т. к. каждая группа из FETCH_COUNT строк форматируется отдельно, что приводит к различию ширины столбцов в разных группах строк. Остальные форматы работают лучше.

Продолжение таблицы 129

Переменная	Описание
HISTCONTROL	При установке этой переменной в ignorespace строки, начинающиеся с пробельных символов, в историю выполненных команд не попадают. При установке в ignoredups в историю выполненных команд не попадают повторяющиеся команды. Значение ignoreboth комбинирует рассмотренные два варианта. Если значение переменной не установлено или установлено в любое другое значение, все строки в интерактивном режиме сохраняются в истории выполненных команд.
HISTFILE	Имя файла, используемое для хранения истории выполненных команд. Значение по умолчанию — ~/.psql_history. Например, задание: \set HISTFILE ~/.psql_history- :DBNAME в файле ~/.psqlrc приводит к поддержке ведения истории выполненных команд раздельно для каждой БД.
HISTSIZE	Количество команд, сохраняемых в истории выполненных команд. Значение по умолчанию — 500.
HOST	Имя сервера БД, с которым установлено соединение. Значение устанавливается каждый раз при установке соединения с БД (включая запуск программы) и не может быть сброшено.
IGNOREEOF	Если не установлено, посылка символа EOF (обычно **<Ctrl+D>**) в интерактивной сессии psql приводит к завершению работы программы. При установке в числовое значение указанное количество символов EOF игнорируется перед завершением работы. При установке не в числовое значение применяется значение по умолчанию — 10.
LASTOID	Значение последнего полученного идентификатора OID, возвращенного командами INSERT или \lo_insert. Гарантируется корректное значение переменной до вывода результата следующей SQL-команды.
ON_ERROR_ROLLBACK	При установке в on, в случае генерации ошибки внутри блока транзакции, она игнорируется, и транзакция продолжается. В интерактивном режиме подобные ошибки игнорируются только при ручном вводе, но не при чтении файлов скритов. При значении off (по умолчанию) выражение в блоке транзакций, вызвавшее ошибку, прерывает выполнение всей транзакции. Режим on обеспечивается неявной установкой точки сохранения (SAVEPOINT) перед выполнением каждой команды в блоке транзакции и откат к ней при возникновении ошибки.
ON_ERROR_STOP	По умолчанию, в случае возникновения ошибки при выполнении не интерактивного скрипта, такой как неверно написанная SQL- или метакоманда, обработка не останавливается. При установке этой переменной обработка скрипта прерывается. Если скрипт был вызван из другого скрипта, он так же прерывается. При выполнении внешнего скрипта не в интерактивном режиме, а с помощью аргумента командной строки -f, утилита psql возвращает код ошибки 3, для отличия от завершения по причине возникновения фатальной ошибки (код ошибки 1).
PORT	Порт сервера БД, с которым установлено соединение. Значение устанавливается каждый раз при установке соединения с БД (включая запуск программы) и не может быть сброшено.
PROMPT1 PROMPT2 PROMPT3	Определяют вид приглашения в psql.
QUIET	Установка этой переменной в on эквивалентна аргументу командной строки -q.
SINGLELINE	Установка этой переменной в on эквивалентна аргументу командной строки -S.
SINGLESTEP	Установка этой переменной в on эквивалентна аргументу командной строки -s.
USER	Пользователь БД, от имени которого установлено соединение. Значение устанавливается каждый раз при установке соединения с БД (включая запуск программы) и не может быть сброшено.

Окончание таблицы 129

Переменная	Описание
VERBOSITY	Может быть установлена в значения default, verbose и terse для управления детализацией сообщений об ошибках.

18.18.4.2. Подстановки в SQL

Еще одна из возможностей использования переменных psql заключается в подстановке их в SQL-выражения, таким же образом, как и в метакоманды. Кроме того, psql обеспечивает корректное обрамление кавычками значений, используемых в качестве литералов и идентификаторов SQL. Синтаксис подстановки значения без заключения в кавычки заключается в предварении имени переменной двоеточием (:), например:

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

Будет осуществлен запрос из таблицы my_table. Значение переменной подставляется буквально, поэтому может содержать даже непарные кавычки или метакоманды. Необходимо быть уверенным, что подстановка имеет смысл.

Если значение используется в качестве литерала или идентификатора SQL, более безопасным является заключение его в кавычки. При использовании значения переменной в качестве литерала SQL имя переменной после двоеточия обрамляется одинарными кавычками. При использовании значения переменной в качестве идентификатора SQL имя переменной после двоеточия обрамляется двойными кавычками. Такие конструкции корректно обрабатывают кавычки и прочие специальные символы внутри значений переменных. Предыдущий пример может быть более безопасно записан следующим образом:

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :"foo";
```

Внутри заключенных в кавычки элементов SQL подстановка не осуществляется. Следовательно, конструкция типа ':foo' не обеспечивает получение из значения переменной заключенного в кавычки литерала (и не безопасно даже если сработает, поскольку в этом случае включенные в значения кавычки обрабатываются некорректно).

Одним из возможных способов использования этого механизма является копирование содержимого файла в столбец таблицы. Сначала файл загружается в переменную, затем используется как было описано:

```
testdb=> \set content `cat my_file.txt`
testdb=> INSERT INTO my_table VALUES (:'content');
```

Следует отметить, что данный подход не сработает, если файл my_file.txt содержит NUL байты. psql не поддерживает использование NUL байт в значениях переменных.

Поскольку двоеточие может легально появляться в командах SQL, явная попытка подстановки (например, :name, :'name' или :"name") не будет выполнена, если значе-

ние переменной не установлено. В любом случае существует возможность экранировать двоеточие символом \ для предотвращения подстановки.

Синтаксис двоеточия для переменных является стандартом SQL для встраиваемых языков запросов, таких как ECPG. Синтаксис двоеточия для срезов массивов и приведений типов является расширением PostgreSQL и может приводить к конфликту. Синтаксис двоеточия с кавычками для значения переменных, при использовании их в качестве литералов и идентификаторов SQL, является расширением psql.

18.18.4.3. Приглашения

Приглашение psql может быть настроено в зависимости от предпочтения. Три переменные PROMPT1, PROMPT2 и PROMPT3 содержат строки и специальные управляющие последовательности, описывающие представление приглашения. PROMPT1 — обычное приглашение, которое выводится при запросе новой команды. PROMPT2 — приглашение, выводимое при ожидании продолжения ввода команды, если команда не была завершена символом «;» или не были закрыты кавычки. PROMPT3 выводится при запуске команды SQL COPY для ввода данных с помощью терминала.

Содержимое выбранного приглашения выводится буквально, за исключением обнаруженного знака %. В зависимости от следующего символа, вместо него подставляется некоторый текст. Определенные в psql подстановки приведены в таблице 130.

Таблица 130

Подстановка	Описание
%M	Полное имя сервера БД (включая имя домена), или [local], если соединение установлено с помощью Unix-сокетов, или [local:/dir/name] при нестандартном расположении Unix-сокетов.
%m	Имя сервера БД, обрезанное по первой точке, или [local], если соединение установлено с помощью Unix-сокетов.
%>	Порт, по которому сервер БД принимает соединения.
%n	Имя пользователя установленной с БД сессии. (Значение может меняться в течение сессии в результате выполнения команды SET SESSION AUTHORIZATION.)
%/	Имя текущей БД.
%~	Аналогично %/, но в случае соединения с БД по умолчанию выводится тильда ( ).
%#	Если пользователь сессии является суперпользователем, выводится #, иначе — >. (Значение может меняться в течение сессии в результате выполнения команды SET SESSION AUTHORIZATION.)
%R	В PROMPT1 в нормальном состоянии выводится =, ^ — в однострочном режиме, и !, если не установлено соединение с БД (что может случиться при неудачной попытке выполнения команды \connect). В PROMPT2 заменяется на -, *, одинарную кавычку, двойную кавычку или символ $, в зависимости от того, что ожидает psql в дальнейшем, при незавершенной команде, внутри блока комментариев /* ... */ или внутри незакрытого экранирования кавычками или символом $. В PROMPT3 ничего не подставляется.
%x	Состояние транзакции: пустая строка вне блока транзакции, * — внутри блока транзакции, ! — внутри блока транзакции с ошибкой, ? — состояние транзакции не определено (например, при отсутствии соединения).

Окончание таблицы 130

Подстановка	Описание
%digits	Отображается символ, заданный в восьмеричном виде.
%:name:	Значение указанной переменной psql (см. 18.18.4.1).
%'command'	Результат выполнения команды, аналогичный обычной подстановке при использовании апострофов.
%[... %]	Приглашения могут содержать управляющие символы (которые, например, меняют цвет, цвет фона, стиль выводимого текста) или менять заголовок окна терминала. Для корректной работы возможностей редактирования библиотеки Readline указанные непечатаемые управляющие символы должны определяться как невидимые путем окружения их %[и %]. Допустимо их множественное использование в приглашении. Например: testdb=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%]%# ' выводит жирное (1;) желтым по черному (33;40) приглашение на цветных терминалах, совместимых с VT100.

Для использования символа % в приглашении он должен быть удвоен %%. По умолчанию используется '%/%R%# ' для PROMPT1 и PROMPT2, '>> ' — для PROMPT3.

18.18.4.4. Редактирование командной строки

Утилита psql поддерживает использование библиотеки Readline для удобного редактирования и поиска в командной строке. История команд автоматически сохраняется в момент завершения работы и восстанавливается при запуски программы. Также поддерживается автозавершение по табуляции, хотя механизм автозавершения не претендует на роль синтаксического анализатора SQL. Если по некоторым причинам автозавершение нежелательно, необходимо в файл .inputrc в домашней директории добавить:

```
$if psql
set disable-completion on
$endif
```

Примечание. Это особенность библиотеки Readline.

18.18.5. Переменные окружения

Переменные окружения приведены в таблице 131.

Таблица 131

Переменная	Описание
COLUMNS	Если \pset columns равно нулю, задает ширину для формата «wrapped» и ширину для определения того, требуется ли средство прокрутки вывода.
PAGER	Если результат выполнения запроса не помещается на экране, он передается указанной программе. Обычные значения — more или less. Установка по умолчанию зависит от платформы. Использование средства прокрутки вывода может быть запрещено с помощью команды \pset.
PGDATABASE PGHOST PGPORT PGUSER	Параметры соединения по умолчанию (см. 18.1).

Окончание таблицы 131

Переменная	Описание
PSQL_EDITOR EDITOR VISUAL	Редактор для использования командами \e и \ef. Переменные просматриваются в указанном порядке, используется первая установленная.
PSQL_EDITOR_LINENUMBER_ARG	Задает аргумент, передаваемый внешнему редактору, для указания стартового номера строки, в случае использования параметров \e и \ef с указанием номера строки. Для редакторов типа Emacs и vi используется знак +. При необходимости наличия пробела между именем опции и номером строки, в значение переменной необходимо включать пробельный символ. PSQL_EDITOR_LINENUMBER_ARG='+' PSQL_EDITOR_LINENUMBER_ARG='--line ' По умолчанию в Unix используется + (согласно редактору по умолчанию vi).
PSQL_HISTORY	Альтернативное расположение файла истории команд. Применяется расширение (~).
PSQLRC	Альтернативное расположение пользовательского файла .psqlrc. Применяется расширение (~).
SHELL	Команда, выполняемая для \!.
TMPDIR	Каталог для хранения временных файлов. По умолчанию — /tmp.

Утилита psql, как и другие утилиты PostgreSQL, использует переменные окружения, поддерживаемые библиотекой libpq.

18.18.6. Файлы

Если не указаны опции -X или -c, в процессе запуска psql до приема команд пытается прочитать и выполнить команды из общесистемного файла psqlrc и пользовательского файла ~/.psqlrc (см. .../share/psqlrc.sample для примера общесистемного файла) после подключения к БД. Он может быть использован для настройки параметров клиента или сервера (используя команды set или SET).

Общесистемный файл psqlrc расположен в системном каталоге конфигурации, который может быть определен с помощью pg_config --sysconfdir. Указанный каталог может быть задан явным образом с помощью переменной окружения PGSYSCONFDIR.

Пользовательский файл .psqlrc расположен в домашнем каталоге пользователя.

Как общесистемный файл psqlrc, так и пользовательский файл ~/.psqlrc могут быть специализированы путем указания версии PostgreSQL, например ~/.psqlrc-9.1.x. При использовании предпочтение отдается специализированным файлам.

История команд сохраняется в файле ~/.psql_history.

Корректная работа psql гарантируется только с серверами той же версии, что и утилита. Это не означает, что при других комбинациях всегда возникают ошибки, но могут возникать те или иные проблемы. Метакоманды, в частности, могут выполняться неверно с сервером более новой версии, чем psql. В тоже время, метакоманды семейства \d

корректно работают с серверами, начиная с версии 7.4, но не обязательно с серверами новее, чем `psql`.

Примеры:

1. Разбиение команды при вводе в несколько строк (обратите внимание на изменение приглашения при этом):

```
testdb=> CREATE TABLE my_table (
testdb(> first integer not null default 0,
testdb(> second text)
testdb-> ;
CREATE TABLE
```

2. Просмотр определения таблицы:

```
testdb=> \d my_table
          Таблица "my_table"
   Атрибут  |   Тип   |     Модификатор
 -----------+---------+--------------------
  first     | integer | not null default 0
  second    | text    |
```

3. Изменение приглашения:

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

4. Просмотр содержимого таблицы:

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 строк)
```

5. Изменение стиля отображения таблиц командой `\pset`:

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
```

```
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 строк)
peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 строк)
peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep ","
Field separator is ",".
peter@localhost testdb=> \pset tuples_only
Showing only tuples.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4
```

6. Альтернативное использование коротких команд:

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
```

```
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

18.19. reindexdb - пересоздание индексов в базе данных

Для пересоздания индексов в БД PostgreSQL используется утилита `reindexdb`.

Утилита является оболочкой для вызова SQL-команды `REINDEX`, при этом нет разницы в производительности при использовании утилиты или при ином доступе к серверу.

Синтаксис:

```
reindexdb [option...] [dbname]
reindexdb [option...] --all|-a
reindexdb [option...] --system|-s [dbname]
```

Аргументы командной строки приведены в таблице 132.

Таблица 132

Аргумент	Описание
`dbname`	Указывает имя используемой БД. По умолчанию в качестве имени БД используется имя пользователя, заданное при установке соединения.
`-a` `--all`	Пересоздать индексы во всех БД.
`[-d] dbname` `[--dbname=] dbname`	Указывает имя БД для пересоздания индексов. Если аргумент не указан, и не указаны аргументы (`-a, --all`), имя БД берется из переменной окружения `PGDATABASE`. Если она не установлена, в качестве имени БД используется имя пользователя, заданное при установке соединения.
`-e` `--echo`	Показывать команды, которые формируются и отправляются серверу.
`-i index` `--index=index`	Пересоздать только указанный индекс.
`-q` `--quiet`	Не выводить сообщений в процессе работы.
`-s` `--system`	Пересоздать индексы системного каталога.
`-t table` `--table=table`	Пересоздать индексы только указанной таблицы.

Утилита также принимает аргументы командной строки, используемые для установки соединения (см. 18.1).

Утилита `reindexdb` может требовать установления соединений с сервером PostgreSQL несколько раз, каждый раз запрашивая пароль. В подобном случае удобно

иметь файл ~/.pgpass.

Примеры:

1. Пересоздание индексов в БД test:

```
$ reindexdb test
```

2. Пересоздание индекса в таблице foo и индекса bar в БД abcd:

```
$ reindexdb --table foo --index bar abcd
```

18.20. vacuumdb - чистка и анализ БД

Для сборки «мусора» используется утилита vacuumdb. Утилита также собирает статистику для оптимизатора запросов PostgreSQL.

Утилита является оболочкой для вызова SQL-команды VACUUM, при этом нет разницы в производительности при использовании утилиты или при ином доступе к серверу.

Синтаксис:

```
vacuumdb [option...] [ --table | -t table [( column [,...] )] ] ... [dbname]
vacuumdb [option...] --all | -a
```

Аргументы командной строки приведены в таблице 133.

Таблица 133

Аргумент	Описание
dbname	Указывает имя используемой БД. По умолчанию в качестве имени БД используется имя пользователя, заданное при установке соединения.
-a --all	Произвести сборку «мусора» во всех БД.
[-d] dbname [--dbname=] dbname	Указывает имя БД для сборки «мусора». Если аргумент не указан, и не указаны аргументы (-a, --all), имя БД берется из переменной окружения PGDATABASE. Если она не установлена, в качестве имени БД используется имя пользователя, заданное при установке соединения.
-e --echo	Показывать команды, которые формируются и отправляются серверу.
-f --full	Произвести глобальную сборку «мусора».
-F --freeze	Заморозить информацию о транзакциях в строках.
-q --quiet	Не выводить сообщений в процессе работы.
-s --system	Пересоздать индексы системного каталога.
-t table [(column [,...])] --table=table [(column [,...])]	Произвести сборку «мусора» или собрать статистику только указанной таблицы. Имена столбцов могут быть указаны только при использовании совместно с опцией --analyze. При указании столбцов может понадобиться экранировать скобки (см. примеры, приведенные ниже).

Окончание таблицы 133

Аргумент	Описание
-v --verbose	Выводить дополнительную информацию в процессе работы.
-z --analyze	Обновить статистику для оптимизатора.
-Z --analyze-only	Вычислить только статистику для использования оптимизатора (без вакуума). Примечание. Данная опция может быть использована только в версии 9.6.
--analyze-in-stages	Вычислить только статистику для использования оптимизатором (без вакуума), наподобие --analyze-only. Запускает несколько (на текущий момент три) этапов анализа с различными конфигурационными настройками для получения статистики. Эта опция полезна для анализа базы данных, которая была недавно восстановлена из дампа или с помощью pg_upgrade. vacuumdb с этой опцией будет пытаться создавать статистику как можно быстрее для ее использования, а также для получения полной статистики на последующих этапах. Примечание. Данная опция может быть использована только в версии 9.6.
-Z --analyze-only	Только обновить статистику для оптимизатора.
--maintenance-db=dbname	Задает имя БД, к которой необходимо подключится для определения баз данных, в которых должна быть выполнена сборка «мусора». Если не указано, используется postgres, а если она отсутствует, то template1.

Утилита также принимает аргументы командной строки, используемые для установки соединения (см. 18.1).

Утилита vacuumdb может требовать установления соединений с сервером PostgreSQL несколько раз, каждый раз запрашивая пароль. В подобном случае удобно иметь файл ~/.pgpass.

Примеры:

1. Сборка «мусора» в БД test:

```
$ vacuumdb test
```

2. Сборка «мусора» и проведение анализа для оптимизатора в БД bigdb:

```
$ vacuumdb --analyze bigdb
```

3. Сборка «мусора» в отдельной таблице foo в БД xyzzy и анализ одного ее столбца bar для оптимизатора:

```
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzy
```

19. СЕРВЕРНЫЕ ПРИЛОЖЕНИЯ POSTGRESQL

В разделе содержится справочная информация о серверных приложениях и утилитах PostgreSQL. Описываемые команды могут быть запущены только на хосте, где функционирует сервер БД. Остальные утилиты приведены в разделе 18.

19.1. `initdb` - создание кластера БД

Для создания нового кластера БД PostgreSQL на сервере используется утилита `initdb`.

Утилита `initdb` предназначена для создания нового кластера БД PostgreSQL. Кластер БД представляет собой набор БД, управляемых одним запущенным экземпляром сервера PostgreSQL.

Создание кластера БД состоит из создания каталогов для хранения данных БД, создания разделяемых таблиц системного каталога (таблиц, относящихся ко всему кластеру БД, а не к конкретной БД), и создания БД `template1` и `postgres`. При создании в дальнейшем новых БД в них копируется содержимое БД `template1`. (Таким образом, все, что установлено в БД `template1`, автоматически будет скопировано в каждую создаваемую в дальнейшем БД. БД `postgres` является БД по умолчанию для использования пользователями, утилитами и сторонними приложениями.

Хотя `initdb` и предпринимает попытки создать указанный каталог данных, может не хватить привилегий, если родительский каталог принадлежит суперпользователю. В этом случае пустой каталог создается от имени суперпользователя, после чего права на его владение передаются соответствующему пользователю БД, после чего командой `su` осуществляется смена пользователя на пользователя БД для запуска `initdb`.

Утилита `initdb` должна запускаться пользователем, владеющим серверным процессом, поскольку сервер должен иметь доступ к создаваемым файлам и каталогам. В связи с тем, что сервер не может быть запущен от имени суперпользователя, утилита `initdb` так же не может быть им запущена. (Фактически будет получен отказ при запуске.)

Утилита `initdb` инициализирует значения по умолчанию для локали и кодировки символов в кластере. Кодировка символов, порядок сортировки (`LC_COLLATE`) и классы наборов символов (`LC_CTYPE`, верхний регистр, нижний регистр, цифры и т. п.) могут быть установлены отдельно для каждой БД при их создании. Утилита определяет эти параметры в БД `template1` и использует их по умолчанию для всех остальных БД.

Для изменения значений по умолчанию для порядка сортировки классов наборов символов используются опции `--lc-collate` и `--lc-ctype`. Порядки сортировки, отличающиеся от C или POSIX, вызывают потерю производительности. В связи с чем, важно выбирать правильную локаль при запуске утилиты `initdb`.

Остальные категории локали могут быть изменены позже при работе сервера. Также возможно использование `--locale` для установки значений по умолчанию для всех категорий локали, включая порядок сортировки и классы наборов символов. Все значения локали сервера (`lc_*`) могут быть отображены с помощью `SHOW ALL`.

Для изменения кодировки по умолчанию используется опция `--encoding`.

Синтаксис:

```
initdb [option...] [--pgdata | -D] directory
```

Аргументы командной строки приведены в таблице 134.

Таблица 134

Аргумент	Описание
`-A authmethod` `--auth=authmethod`	Задает метод аутентификации для локальных соединений, используемый в `pg_hba.conf`. Не стоит использовать метод `trust` (используется по умолчанию для простоты установки), если в системе существуют недоверенные локальные пользователи.
`--auth-host=authmethod`	Задает метод аутентификации для локальных соединений с помощью TCP/IP, используемый в `pg_hba.conf`.
`--auth-local=authmethod`	Задает метод аутентификации для локальных соединений с доменных сокетов UNIX, используемый в `pg_hba.conf`.
`-D directory` `--pgdata=directory`	Задает каталог для создаваемого кластера БД. Это единственная необходимая утилите информация, но можно избежать ее указания установкой переменной окружения `PGDATA`, что может быть удобным, т. к. сервер БД использует в дальнейшем эту переменную для определения каталога кластера.
`-E encoding` `--encoding=encoding`	Задает кодировку шаблонной БД. Она же будет использоваться по умолчанию для кодировок всех создаваемых впоследствии БД. Значение по умолчанию наследуется из локали или устанавливается в `SQL_ASCII`, если локаль отсутствует. Поддерживаемые сервером PostgreSQL кодировки описаны в 12.3.
`-k`, `--data-checksums`	Использовать контрольные суммы на страницах данных для определения ошибок ввода-вывода. Включение контрольных сумм может оказать ощутимое снижение производительности. Эта опция может быть установлена только при иниицализации и не может изменена в дальнейшем. Если установлена, контрольные суммы вычисляются для всех объектов во всех базазах данных. Примечание. Доступно только в СУБД версии 9.6.
`-k` `--data-checksums`	Использование контрольных сумм в страницах данных для облегчения обнаружения разрушения данных подсистемой ввода-вывода. Включение использования контрольных сумм может незначительно снизить производительность. Параметр может быть задан только при инициализации кластера и не может быть в последствии изменен. Если включено, контрольные суммы вычисляются для всех объектов во всех БД.
`--locale=locale`	Задает локаль по умолчанию для кластера БД. Если не указано, локаль наследуется из окружения, в котором запускается `initdb`. Поддержка локализации описана в 12.1.

Окончание таблицы 134

Аргумент	Описание
--lc-collate=locale --lc-ctype=locale --lc-messages=locale --lc-monetary=locale --lc-numeric=locale --lc-time=locale	Аналогично --locale, но задает значение конкретной категории локали.
--no-locale	Эквивалентно --locale=C
-N --nosync	По умолчанию initdb ожидает завершения записи всех файлов на диск. Включение этой опции указывает initdb завершаться без ожидания, что быстрее, но может привести к тому, что следующий за этим сбой ОС может оставить каталог данных разрушенным. Опция удобна для тестирования, но не не должна применяются при реальном использовании.
--pwfile=filename	Считать пароль для нового администратора из файла. В качестве пароля используется первая строка файла.
-S --sync-only	Ожидать завершения записи всех файлов на диск перед выходом. Не отличается от нормального поведения initdb.
-T CFG --text-search-config=CFG	Задает конфигурацию текстового поиска по умолчанию.
-U username --username=username	Задает имя администратора БД. По умолчанию используется эффективное имя пользователя, запустившего утилиту initdb.
-W --pwprompt	Запросить пароль для нового администратора. Если не планируется использование парольной аутентификации, это не является важным. В противном случае использование парольной аутентификации невозможно без задания пароля.
-X directory --xlogdir=directory	Задает расположение каталога журнала транзакций.
Редко используемые опции	
-d --debug	При работе выводить сообщения bootstrap клиента и некоторые другие диагностические сообщения. bootstrap клиент является программой, которую вызывает утилита initdb для создания таблиц системного каталога. Опция генерирует очень большое количество диагностических сообщений.
-L directory	Задает путь поиска входных файлов для инициализации кластера БД. Обычно этого не требуется. При необходимости будет выдан запрос для его явного указания.
-n --noclean	По умолчанию при обнаружении ошибки, не дающей полностью завершить создание кластера БД, утилита initdb удаляет все созданные к этому времени файлы. Опция позволяет запретить очистку и удобна для отладки.

Примечание. initdb также может быть выполнена с помощью pg_ctl initdb.

19.2. pg_controldata - отображение информации о кластере БД

Для отображения информации о кластере БД используется утилита pg_controldata.

pg_controldata выводит информацию, инициализированную при создании кластера утилитой initdb, такую как версия каталога данных. Так же отображается информация

о журнале транзакций и обработке точек сохранения, т. к. она относится ко всему кластеру БД, а не задается отдельно для каждой БД.

Утилита может быть запущена только тем пользователем, который инициализировал кластер, поскольку при этом требуется доступ к каталогу данных. Каталог данных может быть задан аргументом командной строки или переменной окружения `PGDATA`. Утилита поддерживает опции `-V` и `--version`, которые определяют вывод версии `pg_controldata`.

19.3. `pg_ctl` - управление сервером

Для управления сервером БД PostgreSQL используется утилита `pg_ctl`.

`pg_ctl` является утилитой для инициализации кластера, запуска, остановки, перезапуска, перезагрузки конфигурационных файлов и информирования о состоянии сервера PostgreSQL. Хотя сервер может быть запущен вручную, `pg_ctl` включает такие задачи, как: перенаправление вывода журнала и корректное отключение сервера от терминала и группы процессов. Так же предоставляются удобные опции по управляемому завершению работы.

Режимы инициализации `init` или `initdb` создают новый кластер PostgreSQL. Кластер БД представляет собой набор БД, управляемых одним запущенным экземпляром сервера PostgreSQL. Этот режим вызывает команду `initdb`. Дополнительные сведения приведены в 19.1.

В режиме запуска `start` запускается новый сервер. При этом сервер запускается в фоне, и стандартный поток ввода перенаправляется в `/dev/null`. Стандартные потоки вывода и ошибок объединяются и выводятся в журнал сервера (при использовании опции `-f`) или перенаправляются на стандартный поток вывода утилиты `pg_ctl` (не на стандартный поток ошибок). Если файл журнала не выбран, поток вывода утилиты должен быть перенаправлен в файл или передан по конвейеру другому процессу типа программы ротации журналов `rotatelogs`; в противном случае сервер `postgres` будет осуществлять вывод на контролирующий терминал (из фона) и не покинет группу процессов оболочки. Поведение по умолчанию может быть изменено опцией `-l` для перенаправления потока вывода сервера в файл. Рекомендуется использовать опцию `-l` и перенаправление.

В режиме останова `stop` останавливается сервер, запущенный в указанном каталоге. С помощью опции `-m` могут быть выбраны три различных режима останова: «Smart» (умный) — ожидает завершения процессов сохранения резервных копий и отключения всех клиентов. Указанный режим используется по умолчанию. «Fast» (быстрый) — не ждет отключения клиентов и прерывает процессы сохранения резервных копий. Все активные транзакции откатываются, клиенты принудительно отключаются, и сервер останавливается. «Immediate» (немедленный) — прерывает выполнение всех серверных процессов без выполнения корректного завершения работы, что приводит к выполнению восстановления при

перезапуске.

Режим перезапуска `restart` эффективно выполняет остановку сервера с последующим запуском. Это позволяет изменять аргументы командной строки `postgres`. Утилита `pg_ctl` сохраняет и использует для последующего запуска аргументы командной строки, с которыми сервер был запущен изначально.

Режим перезагрузки конфигурации `reload` посылает процессу сервера `postgres` сигнал `SIGHUP`, что приводит к повторному чтению конфигурационных файлов (`postgresql.conf`, `pg_hba.conf` и т. п.). Это позволяет менять параметры конфигурационных файлов, не требующих полного перезапуска сервера.

Режим получения статуса `status` позволяет проверить, запущен ли сервер в указанном каталоге данных. В этом случае выводится PID процесса сервера и аргументы командной строки, с которыми он был запущен. Если сервер не запущен возвращается код возврата `3`.

Примечание. В СУБД версии 9.6 дополнительно может быть возвращен код возврата `4`, если каталога базы данных не указано.

В режиме `promote` резервному серверу, запущенному в указанном каталоге данных, указывается завершить процедуру восстановления и перейти в режим нормального функционирования для выполнения операций чтения-записи.

Режим отправки сигналов `kill` позволяет послать сигнал указанному серверу. Поддерживаемые сигналы отображаются опцией `--help`.

Синтаксис:

```
pg_ctl init[db] [-s] [-D datadir] [-o initdb-options]
pg_ctl start [-w] [-t seconds] [-s] [-D datadir] [-l filename] [-o options]
  [-p path] [-c]
pg_ctl stop [-W] [-t seconds] [-s] [-D datadir]
  [-m s[mart] | f[ast] | i[mmediate] ]
pg_ctl restart [-w] [-t seconds] [-s] [-D datadir] [-c]
  [-m s[mart] | f[ast] | i[mmediate] ] [-o options]
pg_ctl reload [-s] [-D datadir]
pg_ctl status [-D datadir]
pg_ctl promote [-s] [-D datadir]
pg_ctl kill signal_name process_id
```

Аргументы командной строки приведены в таблице 135.

Таблица 135

Аргумент	Описание
-c --core-file	Совершать попытку создания дамп-файлов при фатальных сбоях сервера на платформах, где это возможно, путем поднятия накладываемых ими ограничений на программные ресурсы. Это может быть полезно для отладки или диагностики проблем, поскольку позволяет изучить стек вызовов, полученный из завершившегося неудачей серверного процесса.
-D datadir --pgdata=datadir	Указывает расположение каталога данных. Если не указано, используется переменная окружения PGDATA.
-l filename --log=filename	Записывать (или добавлять) записи журнала сервера в заданный файл. Если файл не существует, он создается. Маска доступа устанавливается в 077 так, чтобы по умолчанию доступ к файлу другими пользователями был исключен.
-m mode --mode=mode	Задает режим остановки. Режим может быть smart, fast, immediate или первой буквой от них. Если не указано, используется smart.
-o options	Задает опции командной строки, передаваемые команде postgres (исполняемому файлу сервера PostgreSQL). Опции обычно окружаются одинарными или двойными кавычками для гарантии их групповой передачи.
-o initdb-options	Задает опции командной строки, передаваемые команде initdb. Опции обычно окружаются одинарными или двойными кавычками для гарантии их групповой передачи.
-p path	Указывает путь к исполняемому модулю postgres. По умолчанию исполняемый модуль берется по тому же пути, что и сама утилита pg_ctl, или при неудаче — жестко заданный каталог. Нет необходимости использовать эту опцию, если не выполнялось нестандартных действий и не была получена ошибка нахождения исполняемого модуля postgres. В режиме initdb указывает путь к утилите initdb.
-s --silent	Сообщать только об ошибках, исключая информационные сообщения.
-t seconds --timeout=seconds	Задает в секундах время ожидания старта или завершения работы сервера. По умолчанию используется 60 сек.
-V --version	показать версию и выйти.
-w	Ожидать завершения старта или остановки работы сервера. Ожидание является опцией по умолчанию для остановки, но не для старта. При ожидании старта pg_ctl выполняет попытки установить соединение с сервером. При ожидании остановки pg_ctl ожидает удаления сервером его PID файла. После ожидания утилита возвращает код ошибки, основанный на успешности операций запуска или останова.
-W	Не ждать завершения операции. Значение по умолчанию для запуска и перезапуска.

Файлы каталога данных:

- postmaster.pid — используется утилитой pg_ctl для определения, запущен ли в настоящее время сервер в данном каталоге данных или нет;

- postmaster.opts — утилита pg_ctl (в режиме перезапуска) передает его содержимое в качестве аргументов командной строки команде postgres, если они не перегружены опцией -o. Содержимое файла также отображается в режиме вывода статуса сервера;

Примеры:

1. Запуск сервера:

```
$ pg_ctl start
```

2. Запуск сервера с ожиданием успешности операции:

```
$ pg_ctl -w start
```

3. Запуск сервера, использующего порт `5433` и запускаемого без `fsync`:

```
$ pg_ctl -o "-F -p 5433" start
```

4. Остановка сервера:

```
$ pg_ctl stop
```

5. Использование опции `-m` позволяет управлять тем, как будет завершаться работа сервера:

```
$ pg_ctl stop -m fast
```

6. Перезапуск сервера:

```
$ pg_ctl restart
```

7. Перезапуск сервера с ожиданием останова и запуска:

```
$ pg_ctl -w restart
```

8. Перезапуск сервера, используя порт `5433`, и отключения `fsync` после перезапуска:

```
$ pg_ctl -o "-F -p 5433" restart
```

9. Получение информации о статусе сервера

```
$ pg_ctl status
pg_ctl: Сервер запущен (pid: 13718)
Командная строка:
/usr/local/pgsql/bin/postgres '-D' '/usr/local/pgsql/data' '-p' '5433'
  '-B' '128'
```

Командная строка будет вызвана в режиме перезапуска.

19.4. pg_resetxlog - удаление журнала транзакций

Для сброса (очистки) журнала опережающей записи транзакций (WAL) и управляющей информации кластера PostgreSQL используется утилита pg_resetxlog.

Утилита pg_resetxlog очищает журнал WAL и опционально сбрасывает некоторую другую информацию, расположенную в файле pg_control. Такая операция иногда необходима в случае, когда эти файлы оказываются повреждены, и должна быть выполнена в последнюю очередь, если сервер не может запуститься из-за этих повреждений.

После запуска этой команды станет возможен запуск сервера, но необходимо иметь в виду, что БД могут находиться в несогласованном состоянии в связи с частично завершенными транзакциями. Необходимо незамедлительно создать резервную копию кластера, запустить initdb и загрузить ее. После загрузки данных требуется осуществить проверку целостности и исправить при необходимости ошибки.

Утилита может быть запущена только пользователем, который устанавливал сервер, т. к. требуется доступ на чтение/запись к каталогу данных. В целях безопасности требуется явное задание каталога данных в командной строке. Утилита pg_resetxlog не использует переменную окружения PGDATA.

Синтаксис:

```
pg_resetxlog [-f] [-n] [-o oid] [-x xid] [-e xid_epoch] [-m mxid,mxid]
  [-O mxoff] [-l xlogfile] datadir
```

Аргументы командной строки приведены в таблице 136.

Таблица 136

Аргумент	Описание
datadir	Каталог данных для проведения операции. Требуется обязательное указание, переменная окружения PGDATA не используется.
-e xid_epoch	Задает epoch ID следующей транзакции.
-f	Выполнить операцию, несмотря на невозможность определения верных данных для pg_control.
-l xlogfile	Задает минимальное начальное положение сегмента WAL для нового журнала транзакций.
-m mxid,mxid	Задает ID следующей мультитранзакции.
-n	Не выполнять операцию, а показать извлеченные контрольные значения (для тестирования).
-o oid	Задает следующий OID.
-O mxoff	Задает смещение следующей мультитранзакции.
-x xid	Задание ID следующей транзакции.
-V --version	Показать версию и выйти.

Если pg_resetxlog сообщает о невозможности определения верных данных для

pg_control, существует возможность, несмотря на это, выполнить операцию с указанием опции -f. В этом случае вместо недостающих данных подставляются правдоподобные значения. Ожидается, что большинство полей совпадут, но может потребоваться ручная установка таких полей, как: «next OID», «next transaction ID», «next transaction ID epoch», «next multitransaction ID», «next multitransaction offset», «WAL starting address» (стартовый адрес журнала транзакций). Указанные поля могут быть установлены с помощью опций: -o, -x, -e, -m, -O и -l. Если не существует возможности определить корректные значения для всех перечисленных полей, опция -f все же может быть использована, но восстанавливаемая БД должна исправляться более серьезно, чем обычно: незамедлительное создание резервной копии и пересоздание должны осуществляться в обязательном порядке. Перед выполнением резервной копии не должны выполняться никакие операции по модификации данных, т. к. любая операция может только усугубить разрушение.

- Безопасное значение для поля «next transaction ID» (-x) может быть определено путем поиска имени файла с наибольшим номером в каталоге pg_clog каталога данных, добавлением к полученному значению единицы и умножению затем на 1048576. Имя файла записано в шестнадцатеричном виде. Значение опции указывать также в шестнадцатеричном виде. Например, если 0011 — это наибольшее значение имени файла в pg_clog, -x 0x1200000 будет верным (пять нулей в конце обеспечивают правильный множитель).
- Безопасное значение для поля «next multitransaction ID» (-m) может быть определено путем поиска имени файла с наибольшим номером в каталоге pg_multixact/offsets каталога данных, добавлением к полученному значению единицы и умножению затем на 65536. Имя файла записано в шестнадцатеричном виде. Значение опции указывать также в шестнадцатеричном виде с добавлением четырех нулей.
- Безопасное значение для поля «next multitransaction offset» (-O) может быть определено путем поиска имени файла с наибольшим номером в каталоге pg_multixact/members каталога данных, добавлением к полученному значению единицы и умножению затем на 52352. Имя файла записано в шестнадцатеричном виде. Значение опции указывать также в шестнадцатеричном виде с добавлением четырех нулей.
- Безопасное значение для поля «WAL starting address» (-l) должно быть больше любого существующего имени файла сегмента WAL в каталоге pg_xlog каталога данных. Имена также заданы в шестнадцатеричном виде.Первая часть «timeline ID» и должна быть оставлена такой же. Например, если 000000010000003200000004A является максимальным именем файла в pg_xlog,

верным будет `-l 000000010000003200000004B` или большее.

`pg_resetxlog` самостоятельно исследует файлы в каталоге `pg_xlog` и выбирает в качестве значения по умолчанию для опции `-l` значение, превышающее максимальное имя найденных файлов. Таким образом, ручное задание опции `-l` требуется только тогда, когда известно, что файлы сегментов журнала WAL не находятся в настоящее время в `pg_xlog`, т. к. располагаются во внешнем архиве, или было потеряно все содержимое каталога `pg_xlog`.

- Определение корректного значения поля «next OID», превышающего на единицу соответствующее значение в БД, не является критичным.

- Значение поля «transaction ID epoch» в БД не хранится, за исключением устанавливаемого `pg_resetxlog`, так что подойдет любое значение. Установка этого поля может потребоваться для обеспечения корректной работы систем репликации, в этом случае, подходящее значение может быть получено из состояния нижестоящей реплицированной БД.

Опция `-n` (не выполнять операций) указывает `pg_resetxlog` отобразить значения, воссозданные из `pg_control`, и выйти без модификации чего-либо. Как правило, это используется для отладки, но удобно для пробного запуска ‘`pg_resetxlog`.

Опция `-n` (не выполнять операций) указывает `pg_resetxlog` отображать значения, восстановленные из `pg_control`, и затем выйти без изменения чего-либо. В основном это инструмент отладки, но может быть полезна для проверки отсутствия ошибок, прежде чем запускать `pg_resetxlog` на выполнение каких-либо работ.

Рассмотренная команда не должна использоваться при запущенном сервере. Утилита `pg_resetxlog` не запускается при обнаружении файла блокировки сервера в каталоге данных. При сбое сервера файл блокировки может остаться, в этом случае существует возможность его удалить для обеспечения запуска утилиты. Но перед этим необходимо убедиться в отсутствии работающих в это время серверных процессов.

19.5. `postgres` - сервер БД

Исполняемым модулем сервера БД PostgreSQL является `postgres`.

Для получения доступа к БД клиентское приложение устанавливает соединение (по сети или локально) к запущенному экземпляру `postgres`. После чего `postgres` запускает отдельный процесс для обработки поступившего соединения.

Один экземпляр `postgres` всегда управляет только одним кластером БД, который является набором БД, расположенных в одном общем месте ФС (в «области данных»). В системе может быть запущено более одного экземпляра `postgres`, если они используют разные области данных и порты для приема соединений. При запуске сервер `postgres`

должен иметь информацию о месте расположения области данных. Расположение может быть указано опцией -D или переменной окружения PGDATA; значения по умолчанию не существует. Обычно -D или PGDATA указывают непосредственно на каталог данных, созданный с помощью initdb.

По умолчанию процесс postgres запускается в интерактивном режиме и выводит сообщения журнала в стандартный поток ошибок. На практике приложение postgres должно быть запущено в фоновом режиме, возможно в момент загрузки системы.

Команда postgres также может быть запущена в однопользовательском режиме. Основное использование этого режима осуществляется в процессе создания каталога данных утилитой initdb. Иногда используется для отладки или восстановления после сбоя (но запуск сервера в однопользовательском режиме не является подходящим для отладки, т. к. при этом не выполняется реального межпроцессного взаимодействия и не возникает блокировок). При запуске в однопользовательском режиме пользователь имеет возможность вводить запросы и получать результаты выполнения на экране, но это форма более полезна для разработчиков, чем для конечных пользователей. В однопользовательском режиме пользователь сессии должен быть установлен в пользователя с идентификатором 1, при этом ему предоставляются исключительные права суперпользователя. Подобный пользователь не обязательно должен существовать, в связи с чем, однопользовательский режим может быть использован для ручного восстановления некоторых видов случайного повреждения системного каталога.

Синтаксис:

```
postgres [option...]
```

Аргументы командной строки приведены в таблице 137.

Таблица 137

Аргумент	Описание
-A 0|1	Разрешает использовать проверки времени исполнения, являющиеся средством отладки для поиска программных ошибок. Опция доступна, только если была разрешена при сборке PostgreSQL. В этом случае значением по умолчанию является on (разрешена).
-B nbuffers	Устанавливает число разделяемых серверными процессами буферов. Значение по умолчанию выбирается автоматически утилитой initdb. Задание этой опции эквивалентно конфигурационному параметру shared_buffers.
-c name=value	Устанавливает значение именованного параметра выполнения (run-time). Большинство опций командной строки являются сокращенной формой подобной установки. -c может встречаться несколько раз для установки множества параметров. Поддерживаемые сервером PostgreSQL параметры описаны в 8.

`-C name`	Выводит значение указанного параметра и завершает работу. (Дополнительная информация приведена выше при описании опции `-c`. Опция может быть использована на запущенном сервере, и возвращает значения из `postgresql.conf`, модифицированное каким-либо параметром использованном при вызове. Это не отражает параметров, используемых при старте кластера. Опция предназначена для взаимодействия других программ (например, `pg_ctl` с экземпляром сервера для получения значений параметров. Пользовательские приложения должны использовать вместо этого `SHOW` или представление `pg_settings`.
`-d debug-level`	Устанавливает уровень отладки. Чем больше заданный уровень, тем большее количество отладочной информации выводится в журнал сервера. Допустимые значения от 1 до 5. Так же возможно задание параметра опцией `-d 0` для заданной сессии, что предотвращает распространение значения уровня отладки родительского процесса `postgres` на эту сессию.
`-D datadir`	Задает расположение каталога данных или конфигурационных файлов в ФС.
`-e`	Устанавливает формат дат по умолчанию для ввода в европейский формат (ДМГ). Так же это может привести к выводу дня перед месяцем в некоторых форматах вывода дат.
`-F`	Отключает вызовы `fsync` для увеличения производительности, но это повышает риск повреждения данных при сбоях системы. Указание этой опции эквивалентно запрету конфигурационного параметра `fsync`.
`-h hostname`	Задает имя узла сети или IP-адрес, с которых `postgres` принимает соединения TCP/IP от клиентских приложений. Значение может быть списком адресов, разделенных запятой или `*`, что означает прием соединения со всех доступных интерфейсов. Пустое значение указывает не принимать соединения по TCP/IP, в этом случае для соединения с сервером могут использоваться только сокеты Unix. По умолчанию используется значение `localhost`. Указание опции эквивалентно установке конфигурационного параметра `listen_addresses`.
`-i`	Позволяет удаленным клиентам устанавливать соединения с помощью TCP/IP. Без указания этой опции принимаются только локальные соединения. Опция эквивалентна установке параметра `listen_addresses` в `*` в конфигурационном файле `postgresql.conf` или с помощью `-h`. Опция является нежелательной, поскольку не предоставляет всей функциональности `listen_addresses`.
`-k directory`	Указывает каталог сокета домена Unix, по которому `postgres` ожидает соединения от клиентских приложений. По умолчанию — `/tmp`, но может быть переопределено во время сборки.
`-l`	Разрешает безопасные подключения с помощью SSL. PostgreSQL должен быть собран с поддержкой SSL.
`-N max-connections`	Устанавливает максимальное количество клиентских соединений, принимаемых сервером. Значение по умолчанию этого параметра выбирается автоматически утилитой `initdb`. Задание этой опции эквивалентно конфигурационному параметру `max_connections`.
`-o extra-options`	Указанные аргументы командной строки передаются всем порождаемым `postgres` процессам. Если строка содержит пробелы, она должна быть заключена в кавычки. Опция является устаревшей; все аргументы командной строки для серверных процессов могут быть указаны непосредственно в командной строке `postgres`.
`-p port`	Задает номер порта TCP/IP или расширение сокета домена Unix, по которым сервер `postgres` принимает соединения от клиентских приложений. Значение по умолчанию указывается в переменной окружения `PGPORT`, или если она не установлена, значение по умолчанию, заданное во время сборки (обычно — `5432`). При указании значения, отличного от используемого по умолчанию, все клиентские приложения должны использовать такое же значение, используя аргументы командной строки или переменную `PGPORT`.

Окончание таблицы 137

Аргумент	Описание
`-s`	Отображать временную информацию и иную статистику после выполнения каждой команды. Это полезно при определении производительности системы и для настройки количества буферов.
`-S work-mem`	Задает объем памяти (в килобайтах) для внутренних операций сортировки и хэширования перед использованием временных файлов на диске. Описание конфигурационного параметра `work_mem` приведено в 8.4.1.
`--name=value`	Устанавливает значение именованного параметра выполнения (run-time); сокращенная форма `-c`.
`--describe-config`	Создать дамп внутренних переменных сервера, описаний и значений по умолчанию в табулированном формате команды `COPY`.

Большинство опций может быть указано в конфигурационном файле. Некоторые (безопасные) опции также могут быть установлены подключившимся клиентом с помощью соответствующего приложения для применения их только в текущей сессии. Например, если задано значение переменной окружения `PGOPTIONS`, клиентские приложения, использующие `libpq`, могут послать эту строку сервера, которая будет интерпретирована им как аргументы командной строки.

Приведенные в таблице 138 опции используются преимущественно в целях отладки и в некоторых случаях могут помочь при восстановление БД с небольшими повреждениями и предназначены для системных разработчиков PostgreSQL.

Таблица 138

Аргумент	Описание
`-f { s | i | o | b | t | n | m | h }`	Запрещает использование указанных методов доступа и объединения: `s` и `i` запрещают последовательный и индексный доступ, соответственно, `o`, `b` и `t` запрещают индексные доступы `index-only`, доступы с помощью карт и TID соответственно, тогда как `n`, `m` и `h` запрещают вложенные циклы (nested-loop), объединение слиянием (merge join) и хэшированием (hash join), соответственно. Ни последовательный метод доступа, ни вложенные циклы не могут быть запрещены полностью; опции `-fs` и `-fn` указывают оптимизатору не использовать эти типы планов, если существует любая другая альтернатива.
`-n`	Используется для отладки в случае некорректного завершения серверных процессов. Происходит уведомление других серверных процессов о необходимости завершения их работы, после чего осуществляется переинициализация разделяемой памяти и семафоров. Это связано с возможностью разрушения некоторых разделяемых состояний при завершении серверного процесса, в котором произошел сбой. Опция указывает не производить переининциализацию разделяемых структур данных, что позволяет опытному системному программисту исследовать состояние разделяемой памяти и семафоров.
`-O`	Разрешает изменение структуры системных таблиц. Используется утилитой `initdb`.
`-P`	Игнорировать системные индексы при чтении системных таблиц (при этом обновление индексов при модификации таблиц сохраняется). Полезна при восстановлении из разрушенных системных индексов.

Окончание таблицы 138

Аргумент	Описание
`-t pa|pl|ex`	Вывести временную статистику для каждого запроса раздельно для каждого из основных системный модулей (`pa[rser] | pl[anner] | e[xecutor]`). Опция не может использоваться совместно с опцией `-s`.
`-T`	Используется для отладки в случае некорректного завершения серверных процессов. Происходит уведомление других серверных процессов о необходимости завершения их работы, после чего осуществляется переинициализация разделяемой памяти и семафоров. Это связано с возможностью разрушения некоторых разделяемых состояний при завершении серверного процесса, в котором произошел сбой. Опция указывает процессу `postgres` останавливать, но не завершать другие процессы посылкой сигнала `SIGSTOP`, что позволяет опытному системному программисту вручную получить дамп всех серверных процессов.
`-v protocol`	Задает версию клиент-серверного протокола для обычных соединений. Опция только для внутреннего использования.
`-W seconds`	Задает временную задержку в секундах при запуске нового серверного процесса до проведения процедура аутентификации. Вводится для создания возможности подсоединиться к серверному процессу средствами отладки.
Опции для однопользовательского режима	
`--single`	Задание однопользовательского режима (должно быть первым аргументом).
`database`	Задает имя БД для доступа. Должно быть последним аргументом в командной строке. Если не указано, по умолчанию используется имя пользователя.
`-E`	Выводить запрос перед выполнением.
`-j`	Не использовать конец строки как интерактивный разделитель запросов.
`-r filename`	Перенаправлять журнал сервера в указанный файл.

Переменные окружения:

- `PGCLIENTENCODING` — кодировка символов, используемая клиентами по умолчанию. (Клиенты в индивидуальном порядке могут переопределять этот параметр.) Так же значение может быть установлено в конфигурационном файле;
- `PGDATA` — расположение каталога данных по умолчанию;
- `PGDATESTYLE` — значение по умолчанию для параметра времени выполнения `DateStyle` (использование этой переменной окружения не рекомендуется);
- `PGPORT` — порт по умолчанию (предпочтительна установка в конфигурационном файле);
- `TZ` — временная зона сервера.

19.5.1. Сообщения об ошибках

Сообщения об ошибках, упоминающие `semget` или `shmget`, вероятно показывают необходимость конфигурирования ядра для обеспечения требуемых объемов разделяемой памяти и семафоров. Существует возможность отложить реконфигурирование ядра путем уменьшения параметра `shared_buffers` для снижения потребности PostgreSQL в разделяемой памяти и/или уменьшения параметра `max_connections` для снижения потребности в семафорах.

Сообщение об ошибке, предполагающее наличие уже запущенного сервера, должно быть проверено в зависимости от системы, например следующими командами:

```
$ ps ax | grep postgres
```

или

```
$ ps -ef | grep postgres
```

В случае уверенности в том, что не существует конфликтующих работающих серверов, упоминаемый в сообщении файл блокировки может быть удален, после чего может быть повторена попытка запуска.

Сообщение об ошибке, указывающее на невозможность использования порта, говорит о том, что порт уже занят некоторым не-Postgres процессом. Так же возможно получение подобного сообщения при немедленном запуске `postgres` после остановки на том же порту; в этом случае требуется подождать в течение нескольких секунд перед повторной попыткой, пока ОС не закроет порт. Существует возможность получения подобного сообщения, если указывается порт, зарезервированный ОС. Например, многие версии Unix рассматривают порты с номерами до 1024, как «доверенные» и предоставляют к ним доступ только суперпользователю.

19.5.2. Дополнительная информация

Для безопасного и удобного запуска и останова сервера `postgres` может быть использована утилита `pg_ctl` (см. 19.3).

По мере возможности не стоит выполнять `SIGKILL` для прерывания основного процесса `postgres`. Это предотвращает освобождение системных ресурсов (разделяемой памяти, семафоров и т. п.) и они остаются после завершения, что может привести к проблемам при последующем запуске сервера `postgres`.

Для корректного завершения работы сервера могут использоваться следующие сигналы: `SIGTERM`, `SIGINT` или `SIGQUIT`. Первый ожидает завершения всех клиентских соединений перед выходом, второй принудительно отключает всех клиентов, третий вызывает немедленный выход без корректного завершения, что приводит к выполнению процедур восстановления при следующем запуске.

Сигнал `SIGHUP` вызывает перезагрузку сервером конфигурационных файлов. Так же возможна посылка сигнала конкретному серверному процессу.

Для прерывания выполнения запроса обрабатывающему серверному процессу посылается сигнал `SIGINT`.

Сервер `postgres` использует сигнал `SIGTERM` для сообщения подчиненным серверным процессам о необходимости корректного завершения работы и сигнал `SIGQUIT` — для завершения работы без выполнения корректных процедур. Указанные сигналы не должны использоваться пользователем. Так же можно послать сигнал `SIGKILL` серверу `postgres`,

основной серверный процесс рассматривает его как сбой и принудительно завершает все подчиненные процессы как часть процедуры восстановления после сбоев.

Для запуска сервера в однопользовательском режиме используется команда следующего вида:

```
postgres --single -D /usr/local/pgsql/data other-options my_database
```

Необходимо указывать корректное значение пути к каталогу данных опцией `-D` или установкой переменной окружения `PGDATA` и имени конкретной БД, с которой требуется работать.

Как правило, сервер в однопользовательском режиме рассматривает новую строку как завершение команды точкой с запятой, в отличие от `psql`. Для продолжения команды на следующих строках необходимо указывать символ `\` перед новой строкой, за исключением последней.

Но при указании опции командной строки `-j` новая строка не рассматривается как завершение команды. В этом случае сервер читает стандартный поток ввода до маркера конца файла (EOF), после чего сервер рассматривает вход как одну команду. Символы `\` в этом случае не несут специального значения.

Для завершения сессии требуется ввод EOF (обычно **<Ctrl+D>**). При использовании `-j` для выхода требуется два последовательных EOF.

Сервер в однопользовательском режиме не предоставляет особых средств редактирования в строке (например, истории команд).

Примеры:

1. Запуск `postgres` в фоновом режиме, используя значения по умолчанию:

```
$ nohup postgres >logfile 2>&1 </dev/null &
```

2. Запуск на указанном порту:

```
$ postgres -p 1234
```

3. Для соединения с таким сервером с помощью `psql`:

```
$ psql -p 1234
```

или установкой переменной окружения `PGPORT`:

```
$ export PGPORT=1234
$ psql
```

4. Установка именованных параметров времени исполнения:

```
$ postgres -c work_mem=1234
$ postgres --work-mem=1234
```

Обе формы перегружают возможные установленные значения work_mem в конфигурационном файле postgresql.conf. Подчеркивания в именах параметров в командной строке могут быть указаны как символами подчеркивания, так и дефисами. Предпочтительной является установка параметров в конфигурационном файле postgresql.conf вместо установки их в командной строке.

19.6. postmaster

Не рекомендуемым названием исполняемого модуля сервера БД является postmaster. Реализован как ссылка на исполняемый модуль postgres.

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

БД — база данных
ЕПП — единое пространство пользователей
КСЗ — комплекс средств защиты
НСД — несанкционированный доступ
ОС — операционная система
СУБД — система управления базами данных
ФС — файлЕовая система

ALD — Astra Linux Directory (единое пространство пользователей)
HBA — Host-based Authentication (аутентификация на основе адресов узлов сети)
IP — Internet Protocol (протокол Интернет)
PAM — Pluggable Authentication Modules (подгружаемые аутентификационные модули)
SQL — Structured Query Language (язык структурированных запросов)
SSL — Secure Sockets Layer (протокол защищенных сокетов)
TCP — Transmission Control Protocol (протокол передачи данных)
UID — User Identifier (идентификатор пользователя)