ООО «Газинформсервис»

УТВЕРЖДЕНО
643.72410666.00067-01 96 01-ЛУ

ЗАЩИЩЕННАЯ СИСТЕМА УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ «JATOBA»

Руководство пользователя

643.72410666.00067-01 96 01

Листов 120

2019

Изм.	Подп.	Дата

Литера

СОДЕРЖАНИЕ

1. НАЗНАЧЕНИЕ И УСЛОВИЯ ПРИМЕНЕНИЯ СУБД «JATOBA»

1.1. Назначение программы

СУБД «Jatoba» предназначена для управления базами данных в значимых объектах критической информационной инфраструктуры 2 категории[1)], в государственных информационных системах 2 класса защищенности[2)], в автоматизированных системах управления производственными и технологическими процессами 2 класса защищенности[3)], в информационных системах персональных данных при необходимости обеспечения 2 уровня защищенности персональных данных[4)], на базе электронно-вычислительных машин (ЭВМ) под управлением информационных систем (ОС):

а) Windows Server 2016;

б) Astra Linux Special Edition (Smolensk) v. 1.6;

в) РЕД ОС 7.2 МУРОМ.

1.2. Функции СУБД «Jatoba»

СУБД «Jatoba» реализует следующие функциональные возможности:

а) управление данными во внешней памяти;

б) управление данными в оперативной памяти;

в) выполнение запросов (DML/DDL);

г) управление транзакциями;

[1)] Устанавливается в соответствии со статьей 7 Федерального закона от 26 июля 2017 г. № 187-ФЗ «О безопасности критической информационной инфраструктуры Российской Федерации» (Собрание законодательства Российской Федерации, 2017, №31, ст. 4736) и Правилами категорирования объектов критической информационной инфраструктуры Российской Федерации, а также перечнем показателей критериев значимости объектов критической информационной инфраструктуры Российской Федерации, утвержденными постановлением Правительства Российской Федерации от 8 февраля 2018 г. № 127 (Собрание законодательства Российской Федерации, 2018, № 8, ст. 1204).

[2)] Устанавливается в соответствии с Требованиями о защите информации, не составляющей государственную тайну, содержащейся в государственных информационных системах, утвержденными приказом ФСТЭК России от 11 февраля 2013 г. № 17 (зарегистрирован Минюстом России 31 мая 2013 г., регистрационный № 28608) (с изменениями, внесенными приказом ФСТЭК России от 15 февраля 2017 г. № 27 (зарегистрирован Минюстом России 14 марта 2017 г., регистрационный №45933; Официальный интернет- портал правовой информации http://www.pravo.gov.ru, 15 марта 2017 г.).

[3)] Устанавливается в соответствии с Требованиями к обеспечению защиты информации в автоматизированных системах управления производственными и технологическими процессами на критически важных объектах, потенциально опасных объектах, а также объектах, представляющих повышенную опасность для жизни и здоровья людей и для окружающей природной среды, утвержденными приказом ФСТЭК России от 14 марта 2014 г. №31 (зарегистрирован Минюстом России 30 июня 2014 г., регистрационный №32919) (с изменениями, внесенными приказом ФСТЭК России от 23 марта 2017 г. №49 (зарегистрирован Минюстом России 30 июня 2017 г., регистрационный №32919; Официальный интернет-портал правовой информации http://www.pravo.gov.ru, 26 апреля 2017 г.) и приказом ФСТЭК России от 9 августа 2018 г. № 138 (зарегистрирован Минюстом России 5 сентября 2018 г., регистрационный № 52071; Официальный интернет- портал правовой информации http://www.pravo.gov.ru, 6 сентября 2018 г.).

[4)] Устанавливается в соответствии с Требованиями к защите персональных данных при их обработке в информационных системах персональных данных, утвержденными постановлением Правительства Российской Федерации от 1 ноября 2012 г. № 1119 (Собрание законодательства Российской Федерации, 2012, № 45, ст. 6257).

д) журнализация изменений, резервное копирование и восстановление базы данных после сбоев, репликация.

СУБД «Jatoba» в дополнение к стандартным возможностям управления базами данных, реализует следующие функции:

а) хранение пространственных, географических и геометрических данных, поддержка запросов к ним и управление ими;

б) синтаксическая совместимость с распространенными PL/SQL Oracle;

в) расширенные возможности секционирования больших таблиц;

г) протоколирование, анализ и контроль выполнения команд манипулирования данными (DML/DDL);

д) сбор журналов аудита всех операций и загрузка конфигураций в СУБД;

е) работа в составе отказоустойчивого кластера с механизмом переключения нагрузки на основной узел кластера;

ж) защита от несанкционированного изменения конфигурационных файлов;

з) единый пользовательский интерфейс для управления конфигурациями компонентов и просмотра их состояния СУБД.

1.3. Требования к среде функционирования СУБД «Jatoba»

СУБД «Jatoba» устанавливается на ЭВМ с процессорами, имеющими архитектуру x86, x86-64 и AMD64 удовлетворяющие следующим аппаратным требованиям, указанным в таблице 1.1.

Таблица 1.1 –Программные и аппаратные требования к средствам вычислительной техники, на которых фукционируют клиентская и серверная часть СУБД

Параметр	Характеристика
Требования к аппаратному обеспечению сервера СУБД	
ОЗУ	Не менее 2 Гб
Свободный объем жесткого диска:	Минимальный объем от 40 Гб Рекомендуемый объем от 50 Тб
Устройства видео вывода	Монитор и видеоадаптер с поддержкой VGA и разрешением 800x600 или выше
Тип процессора и минимальная тактовая частота процессора	64-разрядный процессор Intel или AMD 3 ГГц или больше
Минимальное количество ядер	4
Устройства ввода-вывода	Стандартные 105-клавишная клавиатура и манипулятор “мышь” с USB-интерфейсами
Адаптер Ethernet	100 Мбит/с
Требования к аппаратному обеспечению АРМ управления	
ОЗУ	Не менее 4 Гб
Свободный объем жесткого диск	От 3 Гб

Параметр	Характеристика
Устройства видео вывода	Монитор и видеоадаптер с поддержкой VGA и разрешением 800x600 или выше
Тип процессора и минимальная тактовая частота процессора	64-разрядный процессор Intel или AMD Рекомендуемая частота: 2.4 ГГц или больше
Устройства ввода-вывода	Стандартные 105-клавишная клавиатура и манипулятор “мышь” с USB-интерфейсами
Адаптер Ethernet	100 Мбит/с
Требования к программному обеспечению сервера	
Операционная система	• Windows Server 2016 (с установленной системной библиотекой Visual C++ 2017 версии 14.16.27012.6); • Astra Linux Special Edition (Smolensk) v. 1.6; • РЕД ОС 7.2 МУРОМ.
Требования к программному обеспечению АРМ управления	
Операционная система	• Windows 10 Pro (32-разрядная/64-разрядная) • Astra Linux Special Edition (Smolensk) v. 1.6; • РЕД ОС 7.2 МУРОМ.

2. ПОДГОТОВКА К РАБОТЕ

2.1. Состав СУБД «Jatoba»

В состав СУБД «Jatoba» входят:

а) ядро СУБД;

б) дополнительные программные средства:

- Oracle_fdw – модуль интеграции с СУБД Oracle Database;
- Orafce – модуль миграции с СУБД Oracle Database;
- pgVariable – модуль использования переменных сеанса связи с СУБД
- PostGIS – модуль управления геоданными в СУБД;
- TimescaleDB – модуль работы с большими таблицами и данными с привязкой ко времени;
- pgAudit – модуль протоколирования выполнения команд манипулирования данными (Data Definition Language, язык описания данных (DML)/ Data Manipulation Language, язык манипулирования данными (DDL));
- plsPgSql – модуль сокрытия исходных текстов, процедур и функций в СУБД;
- pgsql_http – модуль отправки запросов по HTTP из хранимых процедур;
- plperl – модуль реализации хранимых процедур на языке Perl;
- plpython – модуль реализации хранимых процедур на языке Python 2;
- plpython3 – модуль реализации хранимых процедур на языке Python 3;
- SecurityProfile – модуль управления парольными политиками пользователей СУБД;

в) модуль пользовательского интерфейса.

2.2. Порядок подключение к СУБД «Jatoba»

2.2.1. Порядок подключения к СУБД «Jatoba» под управлением ОС Windows Server

Для подключения к СУБД «Jatoba» необходимо выполнить следующие действия:

а) в ОС WindowsServer запустить командую строку «cmd»;

б) в открывшемся окне командной строки перейти в каталог с местонахождением psql, при помощи команды:

cd "C:\Program Files\GIS\Jatoba\1.1\bin\"

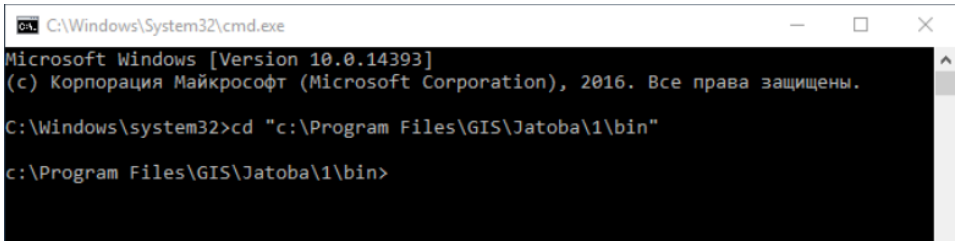

Рисунок 2.1 – Переход в каталог

в) ввести команду для подключения к СУБД:

psql -U <учетная запись пользователя в СУБД> -d<наименование базы данных к которой нужно произвести подключения>

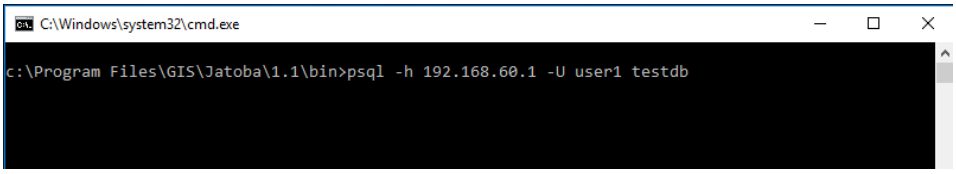

Рисунок 2.2 – Указание адреса подключения, пользователя и БД

г) система предложит ввести пароль учетной записи пользователя, который был введен в действие «в)»;

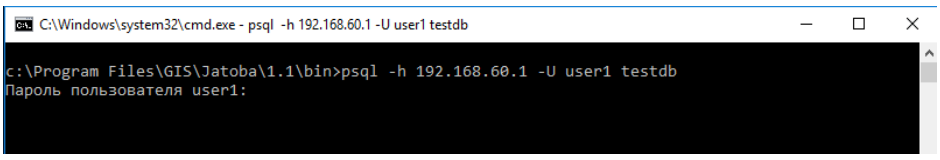

Рисунок 2.3 – Ввод пароля

д) после ввода пароля необходимо нажать на клавишу «ENTER».

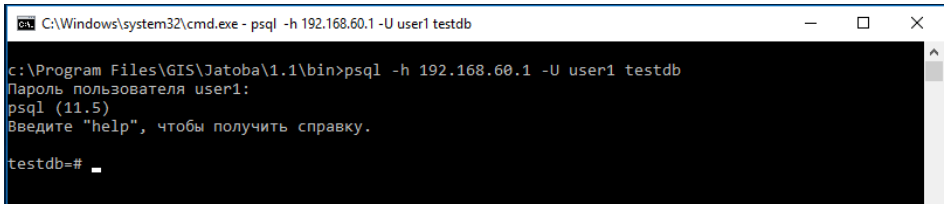

Рисунок 2.4 – Подключение к БД

2.2.2. Порядок запуска СУБД «Jatoba» под управлением ОС семейства Unix

Для подключения к СУБД «Jatoba» необходимо выполнить следующие действия:

а) в ОС Astra Linux запустить «Терминал Fly»[5)], в ОС РЕД ОС 7.2 МУРОМ запустить «Терминал»;

б) в открывшемся окне «Терминал Fly» выполнить команду по подключению к СУБД «Jatoba»:

psql -U <учетная запись пользователя в СУБД> -d <наименование базы данных к которой нужно произвести подключения>

в) система предложит ввести пароль учетной записи пользователя, который был введен в действие «б)»;

г) после ввода пароля необходимо нажать на клавишу «ENTER».

[5)]Для запуска «Терминал Fly» необходимо нажать на кнопку  → «Системные», а затем из выпадающего списка выбрать пункт «Терминал Fly».

3. ОПИСАНИЕ ОПЕРАЦИЙ СУБД «JATOBA»

Пользователь работает с СУБД «Jatoba» при помощи структурированного языка запросов (далее − Structured Query Language (SQL)).

Заполнение базы данных и обращение к ней с помощью запросов выполняется при помощи определенной структуры, а именно синтаксисом SQL.

3.1. Состав синтаксиса SQL

3.1.1. Лексическая структура

Код на языке SQL состоит из последовательности команд (команда завершается знаком точка с запятой «;», либо окончание входного потока). Команда состоит из последовательности компонентов, при этом от синтаксиса зависят какие компоненты можно использовать в этих командах.

Компоненты обычно разделяются пробелами (пробел, табуляция, новая строка), но это необязательно, если нет двусмысленности (что обычно имеет место только в том случае, если специальный символ соседствует с каким-либо другим типом компонента).

Компонент может быть: ключевым словом, идентификатором, идентификатором в кавычках, литералом (или константой) или символом отдельной команды, комментариями, операторы:

а) ключевое слово представляет собой фиксированное значение в языке SQL. Внизу представлен пример:

```
SELECT * FROM test_table;
UPDATE test_table SET B = 5;
INSERT INTO test_table  VALUES (5, 'test');
```

где «*test_table*» наименование таблицы.

На примере представлена последовательность, состоящая из трех отдельных команд, по одной команде на строку.

Синтаксис SQL не является очень строгим относительно компонентов, идентифицирующих команды, а также операндов и параметров. Первые несколько компонентов обычно являются именем команды, как в примере выше, select, update и insert. Но команда update всегда требует наличия компонента SET, который должен находиться в определенной позиции, insert всегда требует наличия компонента values.

В данном примере ключевым словом в данном примере это SELECT, UPDATE и INSERT.

Ключевые слова должны начинаться с буквы a-z, но также буквы с диакритическими знаками и нелатинскими буквами, или подчеркивания

(_). Последующими символами в ключевом слове могут быть буквы, подчеркивания, цифры (0-9) или знаки доллара ($).

б) идентификатор предназначен для идентификации таблиц, столбцов и других объектов базы данных. На примере выше идентификатором является «*test_table*».

Идентификаторы SQL должны начинаться с буквы a-z, но также буквы с диакритическими знаками и нелатинскими буквами, или подчеркивания (_). Последующими символами в идентификаторе могут быть буквы, подчеркивания, цифры (0-9) или знаки доллара ($).

в) литералы (или константы). Константы можно также записывать, указывая типы явно, что позволяет представить их более точно и обработать более эффективно. Константы представлены в подпункте 4.1.2.1.

г) специальные символы:

- знак (*$*), за которым следуют цифры.

  Используется для представления позиционного параметра в теле определения функции или подготовленного оператора. В других контекстах знак доллара может быть частью идентификатора или строковой константы в кавычках.

- круглые скобки (*()*).

  Имеют обычное значение для группирования выражений и обеспечения приоритета. В некоторых случаях скобки требуются как часть фиксированного синтаксиса конкретной команды SQL.

- скобки (*[]*) - используются для выбора элементов массива.

- запятые (,).

  Используются в некоторых синтаксических конструкциях для разделения элементов списка.

- точка с запятой (;) завершает команду SQL.

  Он не может появляться где-либо внутри команды, кроме как внутри строковой константы или идентификатора в кавычках.

- двоеточие (:) используется для выбора «кусочков» из массивов.

- звездочка (*) используется в некоторых контекстах для обозначения всех полей строки таблицы или составного значения. Он также имеет особое значение при использовании в качестве аргумента агрегатной функции, а именно, что агрегат не требует какого-либо явного параметра.

- точка (.) Используется в числовых константах, а также для отделения имён схем, таблиц и столбцов.

д) комментарии. Комментарии представляют собой последовательность символов. Комментарии в СУБД «Jatoba» обозначаются следующим образом:

- если комментарий состоит из одной строки, то тогда необходимо вначале данной строки указать двойную черту, как показано на примере ниже:

```
-- комментарий для одной строки
```

- если комментарий состоит из нескольких строк, то тогда необходимо вначале первой строки указать символы /*, а в конце комментарии символ */, как показано на примере ниже:

```
/* комментарии из
нескольких строк */
```

е) операторы. Под именем оператора понимаются последовательность символов (63 по умолчанию) из следующего списка:

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

Ограничение:

- так как сочетание символов: двоеная черта (--), звёздочка с косой чертой ((*/) или (/*)) не могут присутствовать в имени оператора, так как они обозначаются в системе, как комментарии.
- имена операторов не могут закачиваться на плюс (+), или минус (-), если имя не содержит хотя бы один из этих символов:

  ~ ! @ # % ^ & | ` ?
- при работе с нестандартными именами операторов необходимо разделять соседние операторы пробелами, чтобы избежать двусмысленности. Например, если определен левый унарный оператор с именем @, нельзя писать X * @ Y, а нужно написать X * @Y, чтобы СУБД «Jatoba» читал его как два имени оператора, а не как один.

Приоритет операторов представлен в таблице 3.1.

Таблица 3.1 – Приоритет операторов

Оператор/элемент	Очередность читаемости операторов	Описание
.	слева	разделитель имени таблицы / столбца
::	слева	приведение типов в стиле
[ ]	слева	выбор элемента массива
+ -	справа	унарный плюс, унарный минус
^	слева	определения

Оператор/элемент	Очередность читаемости операторов	Описание
* / %	слева	умножение, деление, по модулю
+ -	слева	сложение, вычитание
любой другой оператор	слева	все остальные собственные и пользовательские операторы
BETWEEN IN LIKE ILIKE SIMILAR		ограничение диапазона, набор членов, сопоставление строк
<> = <= >= <>		операторы сравнения
IS ISNULL NOTNULL		IS TRUE, IS FALSE, IS NULL, IS DISTINCT FROM, etc
NOT	с право	логическое отрицание
AND	слева	логическое соединение
OR	слева	логическая дизъюнкция

3.1.2. Выражения значений

Выражения значений используются в различных контекстах, таких как список целей команды SELECT, в качестве новых значений столбцов в INSERT или UPDATE или в условиях поиска в ряде команд. Чтобы отличить результат выражения значения от результата выражения таблицы (которое является таблицей) данные выражение называются скалярными выражениями или просто выражениями. Синтаксис выражения позволяет вычислять значения из примитивных частей, используя арифметические, логические, множественные и другие операции.

Виды выражение значения:

а) константа или литеральное значение;

б) ссылка на столбец;

в) ссылка на позиционный параметр в теле определения функции или подготовленного оператора;

г) индексное выражение;

д) выражение выбора поля;

е) вызов оператора;

ж) вызов функции;

з) агрегированное выражение;

и) вызов оконной функции;

к) приведение типа;

л) выражение сравнения;

м) скалярный подзапрос;

н) конструктор массива;

о) конструктор строк;

п) другое выражение значения в скобках (используется для группировки подвыражений и переопределения приоритета).

3.1.2.1. Константа или литеральное значение

Константы бывают:

а) строковые константы - произвольная последовательность символов, ограниченная одинарными кавычками ('), например, 'строка'.

Две строковые константы, которые разделены только пробелами хотя бы с одной новой строкой, объединяются и эффективно обрабатываются так, как если бы строка была записана как одна константа.

Внизу представлен пример:

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

эквивалентно:

```
SELECT 'foobar';
```

Но если ввести так:

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

система данный ввод будет считать ошибкой синтаксиса.

б) строковые константы с экранированными последовательностями в стиле языка С.

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 таблице 3.2.

Таблица 3.2 – 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 синтаксис экранирования Unicode.

в) строковые константы с экранированным Unicode.

СУБД «Jatoba» 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 и& (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 случае установки этого конфигурационного параметра в значение off, рассматриваемый синтаксис будет отклонен с сообщением об ошибке.

г) 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, СУБД «Jatoba» 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 СУДБ «Jatoba» 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 $Тад$Содержимое строки$Тад$ - это правильно, а $ТЛО$Содержимое строки$Тад$ - 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.

Заключенными 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.

д) битовые константы.

Битовые константы выглядят как обычные строковые константы с символом в (в нижнем или верхнем регистре), который идет сразу перед открывающей кавычкой (без пробелов), например, B'lOOl'. В битой константе допускаются только символы О и 1. Кроме того, битовые константы могут быть заданы в шестнадцатеричной нотации, используя лидирующий символ х (в верхнем или нижнем регистре), например X'lFF'. Такая нотация эквивалентна битовой константе с четырьмя двоичными разрядами для каждого шестнадцатеричного разряда. Обе формы битых констант могут занимать несколько строк таким же образом, как и обычные строковые константы. Экранирование символом $ не может быть использовано в битовых константах.

е) 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 (е), 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.

Внизу представлены правильные числовые константы:

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 -- TEST СТИЛЬ*

ж) произвольная константа.

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.

CAST(), :: и синтаксис в стиле функции могут также быть использованы для задания преобразования типов произвольных выражений в момент их вычисления. 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 СУБД разрешает его для всех типов.

3.1.2.2. Ссылка на столбец

Ссылка на столбец может иметь вид:

соотношение.имя_столбца

где correlation (соотношение) - это имя таблицы (возможно, дополненной именем схемы) или псевдоним для таблицы, определенной с помощью предложения FROM. Имя соотношения и отдельная точка могут быть опущены, если имя столбца является уникальным для всех таблиц.

3.1.2.3. Ссылка на позиционный параметр

Ссылка на позиционный параметр используется для указания значения, которое подается извне в оператор 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, как:

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

В данном примере $1 ссылается на значение первого аргумента функции в момент, когда вызывается данная функция.

3.1.2.4. Индексное выражение

Если выражение возвращает значение составного типа (тип строки), то конкретное поле строки можно извлечь, для этого необходимо написать:

expression[subsoript]

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

expression[lower_subscript:upper_subscript]

В данном примере 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.

3.1.2.5. Выбор поля

Если выражение возвращает значение составного типа (тип строки), то конкретное поле строки можно извлечь, для этого необходимо написать:

выражение.имя_поля

Обычно выражение *выражение.имя_поля* должно быть в круглых скобках, но круглые скобки могут быть опущены, когда выражение является ссылкой на таблицу или позиционным параметром.

3.1.2.6. Вызов оператора

Существуют все три синтаксиса для вызова оператора:

выражение оператор выражение (двоичный инфиксный оператор)
выражение оператора (оператор унарного префикса)
оператор выражения (унарный постфиксный оператор)

где оператор следует синтаксическим правилам (смотри пункт «е» пункта 4.1.1) или является одним из ключевых слов AND, OR и NOT или полным именем оператора в форме:

OPERATOR(схема.имя_оператора)

3.1.2.7. Вызов функции

Синтаксис для вызова функции - это имя функции (возможно, дополненное именем схемы), за которым следует список аргументов, заключенный в скобки:

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

Внизу представлен пример, где функция вычислит квадратный корень из 16:

sqrt(16)

Аргументы при необходимости могут иметь прикрепленные имена. Функция, которая принимает один аргумент составного типа, может вызываться с использованием синтаксиса выбора поля, и наоборот, выбор поля может быть написан в функциональном стиле. То есть обозначения *col (table)* и *table.col* являются взаимозаменяемыми.

3.1.2.8. Агрегированное выражение

Агрегатное выражение представляет собой применение агрегатной функции, вызываемую для строк, выбранных в запросе. Агрегатная функция получает несколько значений, а выдает только одно значение, такое как сумма или среднее значение. Агрегатное выражение имеет следующий синтаксис:

а) 1 форма:

имя_агрегатной_функции (выражение [, ...] [пункт_order_by]) [FILTER (WHERE пункт_фильтр)]

Агрегатного выражения вызывает агрегат один раз для каждой входной строки.

б) 2 форма:

имя_агрегатной_функции (ALL выражение [, ...] [пункт_order_by]) [FILTER (WHERE пункт_фильтр)]

Синтаксис является такой же, как и в первой форме, за исключением того, что all является значением по умолчанию.

в) 3 форма:

имя_агрегатной_функции (DISTINCT выражение [, ...] [пункт_order_by]) [FILTER (WHERE пункт_фильтр)]

Для каждого уникального значения (или уникального набора значений, для нескольких выражений) вызывается агрегат, найденного во входном наборе строк.

г) 4 форма:

*имя_агрегатной_функции (*) [FILTER (WHERE пункт_фильтр)]*

Вызывает агрегат один раз для каждой входной строки, так как конкретное входное значение не указано.

д) 5 форма:

имя_агрегатной_функции ([выражение [, ...]]) WITHIN GROUP (пункт_order_by) [FILTER (WHERE пункт_фильтр)]

Большинство агрегатных функций игнорируют нулевые входные данные, поэтому строки, в которых одно или несколько выражений дают нулевое значение, отбрасываются. Это можно считать истинным, если не указано иное, для всех встроенных агрегатов.

Например, count (*) дает общее количество входных строк; count (f1) возвращает количество входных строк, в которых f1 не равно нулю, поскольку count игнорирует нули; и count (отличный от f1) дает количество различных ненулевых значений f1.

Обычно входные строки подаются в агрегатную функцию в неопределенном порядке. Во многих случаях это не имеет значения; например, min выдает один и тот же результат независимо от того, в каком порядке он принимает входные данные. Однако некоторые агрегатные функции (например, array_agg и string_agg) выдают результаты, которые зависят от упорядочения входных строк. При использовании такого агрегата необязательный *пункт_фильтр* может использоваться для указания желаемого порядка. У *пункт_фильтр* тот же синтаксис, что и для предложения ORDER BY уровня запроса, за исключением того, что его выражения всегда являются просто выражениями и не могут быть именами или числами выходных столбцов.

3.1.2.9. Вызов оконной функции

Вызов оконной функции представляет собой применение агрегатоподобной функции над некоторой частью строк, выбранных 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 имеет доступ ко всем строкам, которые будут частью группы текущей строки в соответствии со спецификацией группировки (список PARTITION BY) вызова оконной функции. Синтаксис вызова оконной функции один из следующих:

имя_функции ([выражение [,выражение ...]]) [FILTER (WHERE пункт_фильтр)] OVER имя_окна
имя_функции ([выражение [,выражение ...]]) [FILTER (WHERE пункт_фильтр)] OVER (описание_окна)
*имя_функции (*) [FILTER (WHEREпункт_фильтр)] OVERимя_окна*
*имя_функции (*) [FILTER (WHERE пункт_фильтр)] OVER (описание_окна)*

где инструкция «*описание_окна*» имеет следующий синтаксис:

[имя_существующего окна]
[PARTITION BY выражение [, ...]]
[ORDERBYвыражение [ASC | DESC | USINGоператор] [NULLS { FIRST | LAST }] [, ...]]
[пункт_рамки]

Выражение «*пункт_рамки*» является необязательным, и имеет следующую инструкцию:

{ RANGE | ROWS | GROUPS } начало_рамки [исключение_рамки]
{ RANGE | ROWS | GROUPS } BETWEENначало_рамкиANDконец_рамки [исключение_рамки]

где выражение *начало_рамки* и *конец_рамки* могут содержать следующие значения:

UNBOUNDED PRECEDING
смещение PRECEDING
CURRENT ROW
смещениеFOLLOWING
UNBOUNDEDFOLLOWING

а выражение *исключение_рамки может содержать следующие значения:*

EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDENOOTHERS

Здесь выражение представляет собой любое выражение, которое само по себе не содержит вызова оконной функции. Имя_окна является ссылкой на именованную спецификацию окна, определенную в запросе выражением WINDOW. Последняя форма синтаксиса следует тем же правилам, что и изменение существующего имени окна внутри предложения WINDOW.

Предложение PARTITION BY группирует строки запроса в разделы, которые обрабатываются отдельно оконной функцией. PARTITION BY работает аналогично предложению GROUP BY уровня запроса, за исключением того, что его выражения всегда являются просто выражениями и не могут быть именами или числами выходных столбцов. Без PARTITION BY все строки, созданные запросом, обрабатываются как один раздел. Предложение ORDER BY определяет порядок, в котором строки раздела обрабатываются оконной функцией. Он работает аналогично предложению ORDER BY уровня запроса, но также не может использовать имена или номера выходных столбцов. Без ORDER BY строки обрабатываются в произвольном порядке.

Выражение пункт_рамки определяет набор строк, составляющих оконную раму, которая является подмножеством текущего раздела для тех оконных функций, которые действуют на кадр вместо всего раздела. Набор строк в кадре может варьироваться в зависимости от того, какая строка является текущей строкой. Кадр может быть указан в режиме RANGE, ROWS или GROUPS; в каждом случае он запускается от параметров начало_рамки и конец_рамки. Если конец_рамки опущен, конец по умолчанию равен CURRENT ROW.

Выражение начало_рамки UNBOUNDED PRECEDING означает, что кадр начинается с первой строки раздела и, аналогично, конец_рамки UNBOUNDED FOLLOWING означает, что кадр заканчивается последней строкой раздела. В режиме RANGE или GROUPS, начало_рамки CURRENT ROW означает, что кадр начинается с первой строки однорангового соединения текущей строки (строки, которую предложение ORDER BY окна сортирует как эквивалентную текущей строке), в то время как конец_рамки CURRENT ROW означает, что кадр заканчивается с последней равноправной строкой текущего ряда. В режиме ROWS CURRENT ROW просто означает текущую строку.

В варианте *смещение* PRECEDING и FOLLOWING в качестве смещения должно быть выражение, не содержащим никаких переменных, агрегатных функций или оконных функций. Значение смещения зависит от режима кадра:

а) в режиме ROWS *смещение* должно давать ненулевое, неотрицательное целое число, а параметр означает, что кадр начинается или заканчивается указанным числом строк до или после текущей строки.

б) в режиме GROUPS *смещение* должно давать ненулевое, неотрицательное целое число, и это число определяет сдвиг (по количеству групп родственных строк) с которым начало рамки позиционируется перед группой строк, родственных текущей, а конец - после этой группы.

в) В режиме RANGE эти параметры требуют, чтобы предложение ORDER BY указывало ровно один столбец. *Смещение* определяет максимальную

разницу между значением этого столбца в текущей строке и его значением в предыдущих или последующих строках кадра. Тип данных выражения смещения варьируется в зависимости от типа данных столбца.

3.1.2.10. Приведение типов

Приведение типов определяет преобразование из одного типа данных в другой. В СУБД «Jatoba» принимает два эквивалентных синтаксиса для приведения типов:

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

Когда приведение типа выполняется для значения выражения одного из известных типов, оно выполняется как преобразование типа в момент выполнения. Такое приведение типа будет успешным, только если определена подходящая операция преобразования типов. Приведение типа, которое выполняется для строковых констант, 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).

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.

3.1.2.11. Выражение сравнения

Пункт COLLATE отменяет сопоставление выражения и имеет:

exprCOLLATEтип_сортировки,

где *тип_сортировки* является идентификатором сортировки, при необходимости, с указанием имени схемы. Пункт COLLATE имеет более важный приоритет, чем у операторов, при необходимости можно использовать круглые скобки.

Если параметры сортировки явно не указаны, то тогда СУБД либо выводит параметры сортировки из столбцов, указанных в выражениях, либо по умолчанию использует параметры сортировки базы данных.

Внизу представлен пример наиболее распространенных варианта применения пункта COLLATE:

а) в переопределении порядка сортировки в выражении ORDER BY.

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

б) в переопределении способа сортировки при вызове функции или оператора:

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

В последнем случае пункт COLLATE присоединяется к входному аргументу оператора, на который необходимо влиять. 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. Однако присоединение несоответствующих предложений COLLATE к нескольким аргументам является ошибкой, например, команда:

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

выдаст ошибку, так как пытается применить способ сортировки к результату оператора >, который работает с логическим типом данным boolean.

3.1.2.12. Скалярный подзапрос

Скалярный подзапрос - это обычный запрос SELECT в скобках, который возвращает ровно одну строку с одним столбцом. После выполнения запроса SELECT, его результат возвращает значение во внешнее выражение. Ошибочно использовать запрос, который возвращает более одной строки или более одного столбца в качестве скалярного подзапроса. Подзапрос может ссылаться на переменные из внешнего запроса, которые будут действовать как константы во время выполнения подзапроса.

Внизу представлен пример, который ищет наибольшее количество людей, проживающих в городах для каждого штата:

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

3.1.2.13. Конструктор массива

Конструктор массива - это выражение, которое выполняет построение массива из его значений элементов.

Простой конструктор массива состоит из ключевого слова «ARRAY», левой квадратной скобки [, списка выражений (разделенных запятыми) для значений элементов массива и правой квадратной скобки].

```
SELECTARRAY[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)
```

3.1.2.14. Конструктор строк

Конструктор строки - это выражение, которое создает значение строки (также называемое составным значением), используя значения для его полей-членов.

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**)**. 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.

3.1.2.15. Правила оценки выражений

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

3.1.3. Вызов функции

В СУБД «Jatoba» присутствует механизм вызова функций и именованными параметрами с использованием позиционной или именованной нотации. В позиционной нотации вызов функции записывается со значениями аргументов в том же порядке, в

котором они определены в объявлении функции. Именованная нотация полезна для функций, которая состоит из большего количества параметров, так как функция делает ассоциации между параметрами и фактическими аргументами более явными и надежными. В именованной нотации аргументы сопоставляются с параметрами функции по имени и могут быть записаны в любом порядке.

3.2. Типы данных

3.2.1. Тип «money» (денежный)

Тип «money» хранит сумму в валюте фиксированной дробной частью (см. таблицу 3.3). Дробная точность определяется параметром lc_monetary. При попытке загрузить данные с другим параметром, в отличие от lc_monetary, выходные данные могут не работать. Диапазон, указанный в таблице, предполагает наличие двух дробных цифр. Входные данные принимаются в различном формате, включая целые числа, так числа с плавающей запятой, а также типичное форматирование валюты.

Таблица 3.3 – Тип «money»

Имя	Размер хранилища, Байт	Описание	Диапазон
money	8	Сумма в валюте	от -92233720368547758.08 до +92233720368547758.07

Значения типов данных «numeric», «int» и «bigint» могут быть приведены в «money». Преобразование из типов данных реальной и двойной точности можно выполнить, приведя сначала к числовому типу, например,

SELECT '12.34'::float8::numeric::money;

Однако использовать числа с плавающей запятой не рекомендуется из-за возможной ошибки в округлении.

Значения типа «money» можно без проблем перевести в тип «numeric», например,

SELECT '52093.89'::money::numeric::float8;

3.2.2. Типы доменов

Домен - пользовательский тип данных, основанный на другом *нижележащем типе*. Его определение задается условиями, которые ограничивают множество допустимых значений подмножеством значений нижележащего типа. Его поведение схоже с нижележащим типом - например, с доменным типом будут работать те же

операторы или функции, которые работают с нижележащим типом. Нижележащим типом может быть любой встроенный или пользовательский базовый тип, тип-перечисление, массив, составной тип, диапазон или другой домен.

Например, мы можем создать домен поверх целых чисел, принимающий только положительные числа:

CREATE DOMAIN posint AS integer CHECK (VALUE > 0);

CREATE TABLE mytable (id posint);

INSERT INTO mytable VALUES(1); -- работает

INSERT INTO mytable VALUES(-1); -- ошибка

Для доменного типа применяются операторы или функции, предназначенные для нижележащего типа, в таком случае домен автоматически приводится к нижележащему типу. Предположим, что результат операции mytable.id - 1 будет считаться имеющим тип integer, а не posint. Вариант записи (mytable.id - 1)::posint, чтобы снова привести результат к типу posint, повлечёт перепроверку ограничений домена. В этом случае, если данное выражение будет применено к id, равному 1, произойдёт ошибка. Значение нижележащего типа можно присвоить полю или переменной доменного типа, не записывая приведение явно, но и в этом случае ограничения домена будут проверяться.

3.2.3. Числовые типы

Числовые типы состоят из:

а) двух-, четырех и восьмибайтовых целых чисел;

б) восьмибайтовых чисел с плавающей запятой;

в) десятичных дробей с выбираемой точностью.

В таблице 3.4 перечислены все доступные числовые типы

Таблица 3.4 – Числовые типы

Имя	Размер хранилища	Описание	Диапазон
smallint	2 байта	целое в малом диапазоне	от -32768 до +32767
integer	4 байта	типичный выбор для целого числа	от -2147483648 до +2147483647
bigint	8 байта	целое в большом диапазоне	от -9223372036854775808 до +9223372036854775807
decimal	переменная	указанная пользователем точность	до 131072 цифр перед десятичной точкой; до 16383 цифр после запятой

Имя	Размер хранилища	Описание	Диапазон
numeric	переменная	указанная пользователем точность	до 131072 цифр перед десятичной точкой; до 16383 цифр после запятой
real	4 байта	переменная точность	точность в пределах 6 десятичных цифр
double precision	8 байта	переменная точность	точность в пределах 15 десятичных цифр
smallserial	2 байта	небольшое целое число с автоувелечением	от 1 до 32767
serial	4 байта	целое число с автоувелечением	от 1 до 2147483647
bigserial	8 байта	большое целое число с автоувелечением	от 1 до 92233720368547758

3.2.3.1. Целочисленные типы

Типы: smallint, integer и bigint хранят исключительно целые числа в разном допустимом диапазоне и не имеют дробных частей. При попытке сохранить значения за приделами допустимого диапазона приведет к ошибке.

3.2.3.2. Числа произвольной точностью

Тип numeric позволяет хранить очень большое количество цифр, его обычно используют для хранения денежных сумм и других количествах, где требуется точность.

Точность числа представляет собой общее количество цифр во всем числе, то есть количество цифр по обеим сторонам десятичной точки. Шкала чисел - это количество десятичных цифр в дробной части, справа от десятичной точки. Таким образом, число 23,5141 имеет точность 6 и шкалу 4. Можно настроить как максимальную точность, так и максимальный масштаб числового столбца. Чтобы объявить столбец числового типа, используйте синтаксис:

NUMERIC(точность, шкала)

При этом:

а) «*точность*» должен быть положительным;

б) «*шкала*» должен быть положительным, либо ноль.

Если «*шкала*» должен быть указан ноль, то можно применить следующий синтаксис:

NUMERIC(точность)

Если масштаб сохраняемого значения больше, чем объявленный масштаб столбца, система округляет значение до указанного числа дробных цифр. Если число цифр слева от десятичной запятой превышает объявленную точность, возникает ошибка.

Типы decimal и numeric эквивалентны.

3.2.3.3. Тип данных с переменной точностью

Типы real и double precision являются данными с переменной точностью.

Переменная точность означает, что некоторые значения не могут быть преобразованы точно во внутренний формат и хранятся в виде приближений, поэтому при сохранении и извлечении значения могут обнаруживаться небольшие расхождения.

3.2.3.4. Типы последовательности

Типы данных: smallserial, serial и bigserial не являются настоящими типами, а представляют собой просто удобное средство для создания столбцов с уникальными идентификаторами.

3.2.4. Символьные типы

Символьные типы представлены в таблице 3.5.

Таблица 3.5 – Символьные типы

Имя	Описание
character varying(n), varchar(n)	переменная длина с ограничением
character(n), char(n)	строка фиксированной длины с пробелами
text	переменная длина без ограничений

В SQL существуют два основных символьных типов: character varying(n), varchar(n), где n – это положительное целое число. Оба этих типа могут хранить строки длиной до n символов. Попытка сохранить более длинную строку приведет к ошибке.

Записи varchar(n) и char(n) являются синонимами character varying(n) и character(n), соответственно. Записи character без указания длины соответствует character(1). Если же длина не указывается для character varying, этот тип будет принимать строки любого размера.

Тип text можно хранить строки произвольной длины.

3.2.5. Идентификаторы объектов

Идентификатор объекта (Object Identifier, OID) используется внутри СУБД «Jatoba» в качестве первичного ключа различных системных таблиц. Столбец OID добавляется в пользовательские таблицы, в случаях, когда при создании таблицы указывается WITH OIDS или включён параметр конфигурации default_with_oids. Идентификатор объекта представляется в типе oid. Также для типа oid определены следующие псевдонимы: regproc, regprocedure, regoper, regoperator, regclass, regtype, regr ole, regnamespace, regconfig и regdictionary.

На данный момент тип oid реализован как целое число из четырёх байтов. Ввиду этого значение, значения этого типа могут быть недостаточно большими для обеспечения уникальности в базе данных или даже в отдельных больших таблицах. Поэтому в пользовательских таблицах использовать столбец типа OID в качестве первичного ключа не рекомендуется. Самым разумным будет ограничить применение этого типа лишь для обращений к системным таблицам.

Определен ряд операторов для самого типа oid помимо сравнения. Однако его можно привести к целому и затем задействовать в обычных целочисленных вычислениях.

Отличие типов-псевдонимов OID лишь в специализированных функциях ввода/вывода, сами по себе они не вводят новых операций. Эти функции способны принимать и выводить не просто числовые значения, как тип oid, а символические имена системных объектов. Поиск объектов значительно упрощается по значениям OID. Например, чтобы выбрать из pg_attribute строки, относящиеся к таблице mytable, можно написать:

*SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;*

вместо:

*SELECT * FROM pg_attribute*

WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');

Хотя второй вариант выглядит не таким уж плохим, но это лишь очень простой запрос. Сложность вложенного подзапроса повысится, если же потребуется выбрать правильный OID, когда таблица mytable есть в нескольких схемах. Преобразователь вводимого значения типа regclass находит таблицу согласно заданному пути поиска схем,

так что он делает «всё правильно» автоматически. Подобным образом, приведя идентификатор таблицы к типу regclass, можно получить символическое представление числового кода.

Таблица 3.6 – Идентификаторы объектов

Имя	Ссылки	Описание	Пример значения
oid	any	числовой 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 аргументов	*(integer,integer) или -(NONE,integer)
regclass	pg_class	имя отношения	pg_type
regtype	pg_type	имя типа данных	integer
regrole	pg_authid	имя роли	smithee
regnamespace	pg_namespace	пространство имён	pg_catalog
regconfig	pg_ts_config	конфигурация текстового поиска	english
regdictionary	pg_ts_dict	словарь текстового поиска	simple

Все типы псевдонимов OID для объектов, группируемые в пространство имён, принимают имена, дополненные именем схемы, и выводят имена со схемой, если данный объект нельзя будет найти в текущем пути поиска без имени схемы. Типы regproc и regoper способны принимать только уникальные вводимые имена (не перегруженные), что ограничивает их применимость; в большинстве случаев лучше использовать regprocedure или regoperator. Для типа regoperator в записи унарного оператора неиспользуемый операнд заменяется словом NONE.

Еще одно свойство большинства типов псевдонимов OID заключается в образовании зависимостей. Когда в сохранённом выражении фигурирует константа одного из этих типов это создаёт зависимость от целевого объекта. Например, если значение по умолчанию определяется выражением nextval('my_seq'::regclass), СУБД «Jatoba» понимает, что это выражение зависит от последовательности my_seq, и не позволит удалить последовательность раньше, чем будет удалено это выражение. Единственным ограничением является тип regrole. Не допускается использование констант этого типа в выражениях.

Типы псевдонимов OID не полностью следуют правилам изоляции транзакций. Имеется риск неоптимального планирования запросов, когда планировщик воспринимает транзакции как простые константы.

Есть ещё один тип системных идентификаторов, xid, представляющий идентификатор транзакции (сокращённо xact). Состоит из системных столбцов xmin и xmax. Идентификаторы транзакций определяются 32-битными числами.

cid - третий тип идентификаторов, используемых в системе, идентификатор команды (command identifier). Этот тип данных имеют системные столбцы cmin и cmax. Идентификаторы команд - это тоже 32-битные числа.

И наконец, последний тип системных идентификаторов - tid, идентификатор строки/кортежа (tuple identifier). Этот тип данных имеет системный столбец ctid. Идентификатор кортежа представляет собой, идентифицирующую физическое расположение строки в таблице.

3.2.6. Типы даты и времени

СУБД «Jatoba» поддерживается полный набор типов даты и времени, который представлены в таблице 3.7.

Таблица 3.7 – Набор типов даты и времени

Тип даты и времени	Размер хранения, байт	Описание
timestamp [(p)] [without time zone]	8	дата и время (без часового пояса)
timestamp [(*p*)] with time zone	8	дата и время с часовым поясом
date	4	дата (без времени суток)
time [(*p*)] [without time zone]	8	время суток (без даты)
time [(*p*)] with time zone	12	время суток (без даты) с часовым поясом
interval [*fields*] [(*p*)]	16	интервал времени

Типы time, timestamp и interval accept принимают необязательное значение точности «p», которое указывает количество дробных цифр, оставшихся в поле секунд. По умолчанию нет явного ограничения точности.

Тип interval имеет дополнительную опцию, которая заключается в ограничении набора хранимых полей путем записи одного из них:

а) YEAR

б) MONTH

в) DAY

г) HOUR

д) MINUTE

е) SECOND

ж) YEAR TO MONTH

з) DAY TO HOUR

и) DAY TO MINUTE

к) DAY TO SECOND

л) HOUR TO MINUTE

м) HOUR TO SECOND

н) MINUTE TO SECOND

Пример входных данных типа «date»:

а) 2019-01-08

б) January 8, 2019

в) 2019-Jan-08

г) Jan-08-2019

д) 08-Jan-2019

Допустимый ввод для типов time [(p)] without time zone и time [(p)] with time zone. time состоит из времени суток, за которым следует необязательный часовой пояс. Если часовой пояс указан на входе для времени без часового пояса, он игнорируется. Можно указать дату, но она будет игнорироваться, за исключением случаев, когда используется имя часового пояса, которое включает правило перехода на летнее время. В этом случае указание даты требуется для определения того, применяется ли стандартное или летнее время. Соответствующее смещение часового пояса записывается вовремя со значением часового пояса.

Допустимый ввод для типов отметок времени состоит из объединения даты и времени, за которым следует необязательный часовой пояс и AD или BC (в качестве альтернативы AD / BC могут появляться перед часовым поясом, но это не предпочтительный порядок), таким образом:

2020-01-08 04:05:06 и*2020-01-08 04:05:06 -8:00* или *January 8 04:05:06 2020 PST*

Язык SQL имеет возможность различать метку времени (timestamp) без часового пояса, например, *TIMESTAMP '2020-02-19 10:20:46',* и метку времени с литерами часового пояса по наличию следующих символов: «+» и «-» и смещение часового пояса, например, *TIMESTAMP '2020-01-19 10:*23*:54+02'*. Допустимо использовать временную метку с указанием часового пояса «timestamp with time zone», например:

*TIMESTAMP WITH TIME ZONE '2020-01-19 10:*23*:54+02'*

Дополнительно в СУБД «Jatoba» поддерживаются специальные значения ввода даты/времени, которые приведены в таблице 3.8.

Таблица 3.8 – Специальные значения ввода даты/времени

Строка ввода	Действительные типы	Описание
epoch	date, timestamp	1970-01-01 00:00:00+00 (системное время Unix)
infinity	date, timestamp	позже всех других отметок времени
-infinity	date, timestamp	раньше всех других отметок времени
now	date, time, timestamp	время начала текущей транзакции
today	date, timestamp	полночь (00:00) сегодня
tomorrow	date, timestamp	полночь (00:00) завтра
yesterday	date, timestamp	полночь (00:00) вчера
allballs	time	00:00:00.00 UTC

СУБД «Jatoba» позволяет указывать часовые пояса в следующих формах:

а) полное имя часового пояса. Имена часовых поясов перечислены в представлении «pg_timezone_names». Представление «pg_timezone_names» состоит из столбцов, представленных в таблице 3.9.

Таблица 3.9 – Столбцы представления «pg_timezone_names»

Имя	Тип	Описание
name	text	Имя часового пояса
abbrev	text	Аббревиатура часового пояса
utc_offset	interval	Смещение от UTC (положительное значение к востоку от Гринвича)
is_dst	boolean	True, если в настоящее время наблюдается переход на летнее время

б) аббревиатура часового пояса. Аббревиатура часовых поясов перечислены в представление «pg_timezone_abbrevs». Представление «pg_timezone_abbrevs» состоит из столбцов, представленных в таблице 3.10.

Таблица 3.10 – Столбцы представления «pg_timezone_abbrevs»

Имя	Тип	Описание
abbrev	text	Аббревиатура часового пояса
utc_offset	interval	Смещение от UTC (положительное значение к востоку от Гринвича)
is_dst	boolean	True, если в настоящее время наблюдается переход на летнее время

3.2.7. Логический тип данных

Данный тип данных может иметь следующий статус: «true», «false» и «unknown», который будет представлен как NULL.

В таблице 3.11 представлен логический тип данных

Таблица 3.11 – Логический тип данных

Имя	Размер хранилища, байт	Описание
boolean	1	статус: «true» или «false»

Функция ввода для типа данных «boolean» для

а) «true»:

true
yes
on
1

б) false

false
no
off
0

Также допускаются уникальные префиксы этих строк, например, t или n. Начальные или конечные пробелы игнорируются, и регистр не имеет значения.

3.2.8. Двоичные типы данных

Тип данных bytea позволяет хранить бинарную строку. Бинарная строка – это последовательность байтов. Бинарные строки отличаются от символьных строк двумя способами:

а) двоичные строки специально позволяют хранить байты с нулевым значением и другие «непечатные» байты (обычно байты вне десятичного диапазона от 32 до 126). Строки символов запрещают нулевые байты, а также запрещают любые другие значения байтов и последовательности значений байтов, которые являются недопустимыми в соответствии с выбранной кодировкой набора символов базы данных.

б) операции над двоичными строками обрабатывают фактические байты, тогда как обработка символьных строк зависит от настроек. Двоичные строки подходят для хранения данных, которые считаются «необработанными байтами», тогда как символьные строки подходят для хранения текста.

Тип данных bytea поддерживается двумя формата: «hex» и «escape».

Формат вывода зависит от параметра конфигурации bytea_output по умолчанию используется шестнадцатеричное.

3.2.9. Массивы

СУБД «Jatoba» предоставляет возможность определять столбцы таблицы в качестве многомерные массивы переменной длины. Содержимое массивов может состоять из любых встроенных или определённых пользователями базовых типов, перечислений, составных типов, типов-диапазонов или доменов.

3.2.9.1. Трансформации

Иллюстрация использование массивов ниже:

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

Команда CREATE TABLE имеет возможность указывать точный размер массивов, например так:

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

Однако текущая версия не привязана к указанным размерам, т. е. размер массива определить невозможно.

Текущая версия также не ограничивает число размерностей. Все элементы одного типа, если нет зависимости от размера и числа размерностей массива.

Для объявления одномерных массивов можно применять альтернативную запись с ключевым словом ARRAY, соответствующую стандарту SQL. Столбец pay_by_quarter можно было бы определить так:

```
    pay_by_quarter  integer ARRAY[4],
```

Или без указания размера массива:

```
    pay_by_quarter  integer ARRAY,
```

Заметьте, что и в этом случае СУБД «Jatoba» не накладывает ограничения на фактический размер массива.

3.2.9.2. Ввод значений массива

Заключения значения массива в фигурные скобки и разграничение запятыми, констатирует значение массива. Значение любого элемента обязательно заключается в двойные кавычки. Таким образом, общий формат константы массива выглядит так:

'{ значение1 разделитель значение2 разделитель ... }'

где *разделитель* - символ, указанный в качестве разделителя в соответствующей записи в таблице pg_type. Стандартные типы данных, присутствующие в дистрибутиве СУБД «Jatoba», разделены запятыми (,), за исключением лишь типа box, в котором разделитель -точка с запятой (;). Каждое *значение* здесь - это либо константа типа элемента массива, либо вложенный массив. Например, константа массива может быть такой:

'{{1,2,3},{4,5,6},{7,8,9}}'

В примере константа определяет двухмерный массив 3x3, состоящий из трёх вложенных массивов целых чисел.

Значение NULL присваивается элементу массива, вводом NULL. Строка с «NULL», добавляется через двойные кавычки.

Теперь мы можем показать несколько операторов INSERT:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

Результат двух предыдущих команд:

```
SELECT * FROM sal_emp;
name |     pay_by_quarter      |              schedule
-----+-------------------------+-----------------------------------
Bill |{10000,10000,10000,10000}|{{meeting,lunch},{training,presentation}}
```

Carol|{20000,25000,25000,25000}|{{breakfast,consulting},{meeting,lunch}}

(2 rows)

В многомерных массивах число элементов в каждой размерности должно быть одинаковым; в противном случае возникает ошибка. Например:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"meeting"}}');
ОШИБКА: для многомерных массивов должны задаваться выражения
  с соответствующими размерностями
```

Также можно использовать синтаксис конструктора ARRAY:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

Необходимо отметить, что в отличие от буквальной константы массива, элементы массива здесь - это простые SQL-константы или выражения; и поэтому, например, строки будут заключаться в одинарные апострофы.

3.2.9.3. Обращение к массивам

Наполнение таблицы данными, дает возможность перейти к выборкам. В первую очередь получим один элемент массива. Создается запрос на получение имен сотрудников, зарплата которых изменилась во втором квартале:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

 name
```

Carol

(1 row)

Индексы элементов массива записываются в квадратных скобках. По умолчанию в СУБД «Jatoba» действует соглашение о нумерации элементов массива начиная с 1, то есть в массиве из *n* элементов первым считается array[1], а последним - array[*n*].

Запрос выдающий зарплату всех сотрудников в третьем квартале в примере ниже:

SELECT pay_by_quarter[3] FROM sal_emp;

pay_by_quarter

10000

25000

(2 rows)

Имеется возможность получения обычных прямоугольных срезов массивов, то есть подмассивы. Подмассивы обозначаются как *нижняя-граница*:*верхняя-граница* для одной или нескольких размерностей. Запрос в примере получает первые пункты в графике Билла в первые два дня недели:

SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

schedule

{{meeting},{training}}

(1 row)

Если одна из размерностей записана с двоеточием, тогда срез распространяется на все размерности. Если при этом для размерности указывается только одно число, в срез войдут элемент от 1 до заданного номера. Например, в этом примере [2] будет равнозначно [1:2]:

SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';

schedule

{{meeting,lunch},{training,presentation}}

(1 row)

Срезы лучше всегда записывать явно для всех измерений, например [1:2][1:1] вместо [2][1:1].

Значения *нижняя-граница* и/или *верхняя-граница* в указании среза можно опустить; опущенная граница заменяется нижним или верхним пределом индексов массива. Например:

SELECT schedule[:2][2:] FROM sal_emp WHERE name = 'Bill';

schedule

{{lunch},{presentation}}

(1 row)

SELECT schedule[:][1:1] FROM sal_emp WHERE name = 'Bill';

schedule

{{meeting},{training}}

(1 row)

Значение NULL вернет выражение обращения к элементу массива, если сам массив или одно из выражений индексов элемента равны NULL. Значение NULL также возвращается, если индекс Выход индекса за границы массива вернет NULLНапример, если schedule в настоящее время имеет размерности [1:3][1:2], результатом обращения к schedule[3][3] будет NULL. Подобным образом, при обращении к элементу массива с неправильным числом индексов возвращается NULL, а не ошибка.

Выражение обращения вернет NULL, при обращении к подмассиву, если сам массив или одно из выражений, определяющих индексы элементов, равны NULL. Другие случаи, например, когда границы среза выходят за рамки массива, возвращается не NULL, а

пустой массив (с размерностью 0). Если запрошенный срез пересекает границы массива, тогда возвращается не NULL, а подмассив, сокращённый до области пересечения.

Функция array_dims позволяет получить текущие размеры значения массива:

SELECT array_dims(schedule) FROM sal_emp WHERE name = ‘Carol’;

array_dims

[1:2][1:2]

(1 row)

Функция array_dims выдаёт результат типа text, это необходимо для восприятия людей, нежели для программ. Размеры массива также можно получить с помощью функций array_upper и array_lower, которые возвращают соответственно верхнюю и нижнюю границу для указанной размерности:

SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';

array_upper

2

(1 row)

array_length возвращает число элементов в указанной размерности массива:

SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';

array_length

2

(1 row)

cardinality возвращает общее число элементов массива по всем измерениям. Фактически это число строк, которое вернёт функция unnest:

SELECT cardinality(schedule) FROM sal_emp WHERE name = 'Carol';

cardinality

4

(1 row)

3.2.9.4. Изменение массивов

Полная замена значения массива в примере:

```
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';
```

Изменение одного элемента массива:

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
   WHERE name = 'Bill';
```

или среза:

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
   WHERE name = 'Carol';
```

При этом в указании подмассива может быть опущена *нижняя-граница* и/или *верхняя-граница*, но только для массива, отличного от NULL, и имеющего ненулевую размерность.

Присутствует возможность расширения сохранённого массива, после определения значений, ранее отсутствовавших в нем элементов. За счет этого все элементы, располагающиеся между старыми и новыми, принимают значения NULL. Например, если массив myarray содержит 4 элемента, после присвоения значения элементу myarray[6] его длина будет равна 6, а myarray[5] будет содержать NULL. В настоящее время подобное расширение поддерживается только для одномерных, но не многомерных массивов.

Определяя элементы по индексам, можно создавать массивы, в которых нумерация элементов может начинаться не с 1. Например, можно присвоить значение выражению myarray[-2:7] и таким образом создать массив, в котором будут элементы с индексами от -2 до 7.

Оператор конкатенации также может конструировать значения массива:, ||:

```
SELECT ARRAY[1,2] || ARRAY[3,4];
 ?column?
-----------
 {1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
     ?column?
--------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

Оператор конкатенации позволяет вставить один элемент в начало или в конец одномерного массива. Он также может принять два *N*-мерных массива или массивы размерностей *N* и *N+1*.

Когда в начало или конец одномерного массива вставляется один элемент, в образованном в результате массиве будет та же нижняя граница, что и в массиве-операнде. Например:

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
 array_dims
------------
 [1:3]
(1 row)
```

Сложение двух массивов одинаковых размерностей, сохраняет нижняя граница внешней размерности левого операнда. Выходной массив включает все элементы левого операнда, после которых добавляются все элементы правого. Например:

SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);

array_dims

[1:5]

(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);

array_dims

[1:5][1:2]

(1 row)

Когда к массиву размерности *N+1* спереди или сзади добавляется *N*-мерный массив, он вставляется аналогично тому, как в массив вставляется элемент. Любой *N*-мерный массив по сути является элементом во внешней размерности массива, имеющего размерность *N+1*. Например:

SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);

array_dims

[1:3][1:2]

(1 row)

Массив также можно сконструировать с помощью функций array_prepend, array_append и array_cat. Первые две функции поддерживают только одномерные массивы, а array_cat поддерживает и многомерные. Несколько примеров:

SELECT array_prepend(1, ARRAY[2,3]);

array_prepend

{1,2,3}

(1 row)

SELECT array_append(ARRAY[1,2], 3);

```
 array_append
--------------
 {1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
 array_cat
-----------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
     array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
     array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

Для несложных случаев в примерах выше предпочтительнее использовать оператор конкатенации, чем непосредственного вызова этих функций. Однако, на операторе конкатенации решаются все три задачи, возможны ситуации, когда применение одной из этих функций во избежание неоднозначности лучше. Например:

```
SELECT ARRAY[1, 2] || '{3, 4}';  -- нетипизированная строка воспринимается как массив
 ?column?
-----------
 {1,2,3,4}
```

SELECT ARRAY[1, 2] || '7'; -- как и эта

ERROR: malformed array literal: "7"

SELECT ARRAY[1, 2] || NULL; -- как и буквальный NULL

?column?

{1,2}

(1 row)

SELECT array_append(ARRAY[1, 2], NULL); -- это могло иметься в виду на самом деле

array_append

{1,2,NULL}

Рассмотренные примеры характерны тем, что в них анализатор запроса видит целочисленный массив с одной стороны оператора конкатенации и константу неопределённого типа с другой. Следуя своим правилам разрешения типа констант, анализатор полагает, что она имеет тот же тип, что и другой операнд - в данном случае целочисленный массив. На основе этого строится предположение, что оператор конкатенации здесь представляет функцию array_cat, а не array_append. В случае ошибочного решения, его можно скорректировать, приведя константу к типу элемента массива; однако может быть лучше явно использовать функцию array_append.

3.2.9.5. Поиск значений в массивах

Поиск значений в массиве осуществляется путем проверки всех его элементов. Это можно сделать вручную, если вы знаете размер массива. Например:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

Этот метод не удобен для больших массивов и не работает, если размер массива неизвестен. Показанный выше запрос можно было переписать так:

*SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);*

А так можно найти в таблице строки, в которых массивы содержат только значения, равные 10000:

*SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);*

Кроме того, для обращения к элементам массива можно использовать функцию generate_subscripts. Например, так:

```
SELECT * FROM
  (SELECT pay_by_quarter,
          generate_subscripts(pay_by_quarter, 1) AS s
     FROM sal_emp) AS foo
 WHERE pay_by_quarter[s] = 10000;
```

Дополнительно поиск значения в массиве организован, через оператор &&, который проверяет, перекрывается ли левый операнд с правым. Например:

*SELECT * FROM sal_emp WHERE pay_by_quarter && ARRAY[10000];*

Вы также можете искать определённые значения в массиве, используя функции array_position и array_positions. Первая функция возвращает позицию первого вхождения значения в массив, а вторая - массив позиций всех его вхождений. Например:

```
SELECT array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon');
 array_positions
-----------------
 2

SELECT array_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1], 1);
 array_positions
-----------------
 {1,4,8}
```

Массивы не являются множествами; необходимость поиска определённых элементов в массиве говорит о неудачно сконструированной базе данных. В таких

случаях рекомендуется вместо массива использовать отдельную таблицу, строки которой будут содержать данные элементов массива. Это оптимизирует поиск для работы с большим количеством элементов.

3.2.9.6. Синтаксис вводимых и выводимых значений массива

Значение массива представляется в текстовом виде, состоящим из записи элементов, интерпретируемых по правилам ввода/вывода для типа элемента массива, и оформления структуры массива. Оформление заключается в фигурных скобках ({ и }), окружающих значение массива, и знаков-разделителей между его элементами. Знаком-разделителем является запятая (,), однако допускаются и другие символы; за это отвечает параметр typdelim для типа элемента массива. Для стандартных типов данных, существующих в дистрибутиве СУБД «Jatoba», разделителем является запятая (,), за исключением лишь типа box, в котором разделитель - точка с запятой (;). В многомерном массиве у каждой размерности фигурные скобки разграничивают уровни, а соседние значения в фигурных скобках на одном уровне разделяются знаками.

Функция вывода массива заключает значение элемента в кавычки, если это пустая строка или оно содержит фигурные скобки, знаки-разделители, кавычки, обратную косую черту, пробельный символ или это текст NULL. Функция кавычек и обратной косой черты, включённых в такие значения, преобразовать в спецпоследовательность с обратной косой чертой. Числовые типы данных рассчитаны на то, что значения никогда не будут выводиться в кавычках, это не работает для текстовых типов, так как выводимое значение массива может содержать кавычки.

По умолчанию нижний предел всех размерностей массива равняется одному. Другие нижние границы представляются, указанием перед содержимым массива диапазонов индексов. Такое оформление массива будет содержать квадратные скобки ([]) вокруг нижней и верхней границ каждой размерности с двоеточием (:) между ними. За таким указанием размерности следует знак равно (=). Например:

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
 FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

 e1 | e2
```

```
----+----
 1 | 6
(1 row)
```

Результат процедуры вывода массива включает в себя явное указание размерностей, в случае если нижняя граница в одной или нескольких размерностях отличается от 1.

Если в качестве значения элемента задаётся NULL, этот элемент считается равным непосредственно NULL. Если же оно включает кавычки или обратную косую черту, элементу присваивается текстовая строка «NULL». Кроме того, для обратной совместимости с версиями СУБД «Jatoba», параметр конфигурации array_nulls можно выключить, чтобы строки NULL не воспринимались как значения NULL.

В прошлом примере, записаны значения массива, любой его элемент можно заключить в кавычки. Это ***необходимо*** делать, если при разборе значения массива без кавычек возможна неопределенность. Также в кавычки заключаются пустые строки и строки, содержащие одно слово NULL. Чтобы включить кавычки или обратную косую черту в значение, заключённое в кавычки, добавьте обратную косую черту перед таким символом. Обойтись без кавычек можно, достаточно экранированием защитить все символы в данных, которые могут быть восприняты как часть синтаксиса массива.

Ограничивать скобки можно пробельными символами. Пробелы также могут окружать каждую отдельную строку значения. Во всех случаях такие пробельные символы игнорируются. Необходимо учитывать, что все пробелы в строках, заключённых в кавычки, или окружённые не пробельными символами, напротив, считаются.

Конструктор ARRAY упрощает запись значения массивов в командах SQL. В ARRAY отдельные значения элементов записываются так же, как если бы они не были членами массива.

3.2.10. Перечисляемые типы

Перечисляемые типы (emun) представляют собой тип данных, который содержит статический упорядоченный набор значений. Примером данного типа могут быть, например, дни недели, или какой-нибудь набор значений состоящий из фрагмента данных.

3.2.10.1. Объявление перечисляемых типов

Тип emun создается с помощью команды CREATE TYPE. Пример выполнения команды CREATE TYPE:

CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');

Только после создания типа emun он может использоваться в определениях таблиц и функций, так же, как и любой другой тип:

```
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)
```

3.2.10.2. ***Порядок***

Порядок значений в типе emun – это порядок, в котором значения были перечислены при создании типа. Для перечисляемого типа все стандартные операторы сравнения и связанные агрегатные функции поддерживаются. 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)
```

3.2.10.3. Безопасность перечисляемого типа

Каждый перечисляемый тип данных является отдельным и не моет сравниваться с другими перечисляемыми типами. Пример:

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (
    num_weeks integer,
    happiness happiness
);
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ERROR:  invalid input value for enum happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood = holidays.happiness;
ERROR:  operator does not exist: mood = happiness
```

3.2.10.4. Детали реализации

Метка emun чувствительна к регистру и пробелам.

Значение enum занимает четыре байта на диске. Длина текстовой метки значения перечисления ограничена настройкой NAMEDATALEN, скомпилированной в СУБД «Jatoba»; в стандартных сборках это означает максимум 63 байта. Переводы из внутренних значений перечисления в текстовые метки хранятся в системном каталоге pg_enum. Запросы к этому каталогу напрямую могут быть полезны.

3.2.11. Геометрические типы

Геометрические типы данных представляют собой двумерные пространственные объекты. Геометрические типы данных представлены в таблице 3.12.

Таблица 3.12 – Геометрические типы данных

Имя	Размер хранилища, байт	Описание	Пример
point	16	Точка на плане	(x,y)
line	32	Бесконечная линия	{A,B,C}
lseg	32	Конечный отрезок	((x1,y1),(x2,y2))
box	32	Прямоугольник	((x1,y1),(x2,y2))
path	16+16n	Закрытый путь (похож на многоугольник)	((x1,y1),...)
path	16+16n	Открытый путь	[(x1,y1),...]
polygon	40+16n	Многоугольник (похож на закрытый путь)	((x1,y1),...)
circle	24	Круг	<(x,y),r> (центральная точка и радиус)

3.2.11.1. Точка (point)

Точки являются фундаментом двумерных строительным блоком для геометрических типов. Значения типа point указываются с использованием одного из следующих синтаксисов:

(x , y)
x , y

где *x* и *y* - соответствующие координаты в виде чисел с плавающей точкой.

Точки выводятся с использованием первого синтаксиса.

3.2.11.2. Линия (line)

Линия представляется линейным уравнением

$Ax + By + C = 0$,

где *A* и *B* не равны нулю. Значение типа line вводятся и выводятся в следующем виде:

{ A, B, C }

В качестве альтернативы для ввода можно использовать любую из следующих форм:

[(x1 , y1) , (x2 , y2)]
((x1 , y1) , (x2 , y2))
(x1 , y1) , (x2 , y2)
x1 , y1 , x2 , y2

где *(x1, y1)* и *(x2, y2)* две разные точки на линии.

3.2.11.3. Конечный отрезок (lseg)

Конечный отрезок представляется парами точек, которые являются конечными точками сегмента. Значения типа lseg указываются с использованием любого из следующих синтаксисов:

[(x1 , y1) , (x2 , y2)]

((x1 , y1) , (x2 , y2))

(x1 , y1) , (x2 , y2)

x1 , y1 , x2 , y2

где (*x1, y1) и (x2, y2)* - конечные точки отрезка.

Сегменты строки выводятся с использованием первого синтаксиса.

3.2.11.4. Прямоугольники (box)

Прямоугольники представляются парами точек, которые находятся в противоположных углах прямоугольника. Значение типа box указываются с использованием любого из следующих синтаксисов:

((x1 , y1) , (x2 , y2))

(x1 , y1) , (x2 , y2)

x1 , y1 , x2 , y2

где *(x1, y1)* и *(x2, y2)* - любые два противоположных угла прямоугольника.

Прямоугольник выводятся с использованием второго синтаксиса.

Любые два противоположных угла могут быть представлены на входе, но значения будут переупорядочены по мере необходимости, чтобы сохранить верхний правый и нижний левый углы, в этом порядке.

3.2.11.5. Paths

Пути (Paths) представлены списками связанных точек. Пути могут быть открытыми, если первая и последняя точки в списке считаются несвязанными или закрытыми, если первая и последняя точки считаются связанными.

Значения типа path указываются с использованием любого из следующих синтаксисов:

[(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 отрезков, составляющих путь. Квадратные скобки ([]) указывают открытый путь, а круглые скобки (()) указывают закрытый путь. Когда внешние скобки опущены, как в синтаксисах с третьего по пятый, предполагается закрытый путь.

3.2.11.6. Многоугольник (Polygons)

Многоугольник представляется списками точек (вершины многоугольника). Многоугольники очень похожи на закрытые пути, но хранятся по-разному и имею собственный набор подпрограмм поддержки. Значения типа polygon указываются с использованием любого из следующих синтаксисов:

((x1 , y1) , ... , (xn , yn))

(x1 , y1) , ... , (xn , yn)

(x1 , y1 , ... , xn , yn)

x1 , y1 , ... , xn , yn

где точки являются конечными точками отрезков, составляющих границу многоугольника.

3.2.11.7. Круг (circle)

Круги представляются центральной точкой и радиусом. Значения типа circle указываются с использованием любого из следующих синтаксисов:

< (x , y) , r >

((x , y) , r)

(x , y) , r

x , y , r

где *(x, y)* - центральная точка, а *r* - радиус окружности.

3.2.12. Типы сетевых адресов

СУБД «Jatoba» имеет возможность использовать типы данных для хранения адресов IPv4, IPv6 и MAC (см. таблицу 3.13).

Таблица 3.13 – Типы сетевых адресов

Имя	Размер хранилища, Байт	Описание	Примечание
cidr	от 7 до 19	Сеть IPv4 и IPv6	Тип cidr содержит спецификацию сети IPv4 или IPv6. Форматы ввода и вывода соответствуют правилам маршрутизации. Форматом для указания сетей является адрес / y, где адрес - это сеть, представленная в виде адреса IPv4 или IPv6, а «y» - количество битов в маске сети
inet	от 7 до 19	Хост и сеть IPv4 и IPv6	Тип inet может содержать в одном поле адрес хоста IPv4 или IPv6 и, возможно, его подсеть. Подсеть представлена количеством битов сетевого адреса, присутствующих в адресе хоста («маска сети»). Если маска сети равна 32, а адрес - IPv4, то это значение не указывает на подсеть, только на один хост. В IPv6 длина адреса составляет 128 бит, поэтому 128 бит указывают уникальный адрес хоста.
macaddr	6	Mac-адрес	Тип macaddr хранит MAC-адреса
macaddr8	8	Mac-адрес (формат EUI-64)	Тип macaddr8 хранит MAC-адреса в формате EUI-64

3.2.1. Тип pg_lsn

Тип данных pg_lsn применяется для хранения значения LSN, которое представляет собой указатель на позицию в журнале WAL. Этот тип содержит XLogRecPtr и является внутренним системным типом СУБД «Jatoba».

LSN можно определить, как64-битное целое, представляющее смещение байтов в потоке журнала предзаписи. Он выводится в виде двух шестнадцатеричных чисел до 8 цифр каждое, через косую черту, например: 16/B374D848. Тип pg_lsn поддерживает стандартные операторы сравнения, такие как = и >. Можно произвести операцию вычитания одного LSN из другого с помощью оператора -; результатом будет число байт между этими двумя позициями в журнале предзаписи.

3.2.2. Типы битовых строк

За битовые строки в языке SQL принимается последовательность из нулей и единиц. Такие последовательности используются для хранения или отображения битовых масок. Язык SQL поддерживает два битовых типа: bit(n) и bit varying(n), где n - положительное целое число.

Тип bit должен иметь значение длиной равной *n*; попытки сохранения данных короче или длиннее вернут ошибку. Тип данных bit varying предполагает переменную длину, но не превышающую *n*; строки, превышающие длину *n,* не будут приняты. Запись bit, в которой не указана длина, равнозначна записи bit(1), в то время как bit varying без указания длины представляет строку неограниченной длины.

Приведение значения битовой строки к типу bit(*n*), это значение будет сокращено или дополнено нулями справа до длины равной *n* бит, ошибка при этом не возникнет. На подобии предыдущего примера, приведение значения битовой строки к типу bit varying(*n*), строка будет усечена справа, если её длина превышает *n* бит. Пример:

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');
```

ОШИБКА: длина битовой строки (2) не соответствует типу bit(3)

```
INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;

 a  |  b
-----+-----
101 | 00
100 | 101
```

Хранение битовой строки использует 1 байт для каждой группы из 8 бит, добавляя 5 или 8 байт дополнительно в зависимости от длины строки (но длинные строки могут быть сжаты или вынесены отдельно.

3.2.3. Тип данных UUID

Тип данных uuid включает в себя универсальные уникальные идентификаторы (Universally Unique Identifiers, UUID), определённые в RFC 4122, ISO/IEC 9834-8:2005 и связанных стандартах, в отличие от других систем, где этот тип называется GUID. Идентификатор представляет собой 128-битное значение, генерируемое при помощи специального алгоритма, который исключает возможность генерации второго такого значения в мире. Эти идентификаторы являются уникальными и в распределённых системах, а не только в единственной базе данных, как значения генераторов последовательностей.

UUID имеет вид последовательности шестнадцатеричных цифр в нижнем регистре, разделяемых знаками минуса на несколько групп, в определенном порядке: группа из 8 символов, за ней три группы из 4 символов и, наконец, группа из 12 символов, что в сумме составляет 32 символа и представляет 128 бит.

Пример UUID в этом стандартном виде:

a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11

СУБД «Jatoba» распознает также: символы в верхнем регистре, стандартную запись, заключённую в фигурные скобки, запись без минусов или с минусами, разделяющими любые группы из четырёх символов. Например:

A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11

{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}

a0eebc999c0b4ef8bb6d6bb9bd380a11

a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11

{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}

Выводится значение этого типа всегда в стандартном виде.

СУБД «Jatoba» содержит ряд функций хранения и сравнения идентификаторов UUID, но лишена внутренней функции генерирования UUID, ввиду того, что единого алгоритма, подходящего для всех приложений, не существует. Генерировать UUID можно дополнительными модулями.

3.2.4. Псевдотипы

В систему типов СУБД «Jatoba» включены несколько специальных элементов, которые в совокупности называются *псевдотипами*. Псевдотип нельзя использовать в качестве типа данных столбца, но можно объявить функцию с аргументом или результатом такого типа. Каждый из существующих псевдотипов полезен в ситуациях, когда характер функции не позволяет просто получить или вернуть определённый тип данных SQL.

Таблица 3.14 – Псевдотипы

Имя	**Описание**
anyelement	Указывает, что функция принимает любой тип данных.
anyarray	Указывает, что функция принимает любой тип массива.
anynonarray	Указывает, что функция принимает любой тип данных, кроме массивов.
anyenum	Указывает, что функция принимает любое перечисление.
anyrange	Указывает, что функция принимает любой диапазонный тип данных.
cstring	Указывает, что функция принимает или возвращает строку в стиле C.
internal	Указывает, что функция принимает или возвращает внутренний серверный тип данных.
language_handler	Обработчик процедурного языка объявляется как возвращающий тип language_handler.
fdw_handler	Обработчик обёртки сторонних данных объявляется как возвращающий тип fdw_handler.
index_am_handler	Обработчик метода доступа индекса объявляется как возвращающий тип index_am_handler.
tsm_handler	Обработчик метода выборки из таблицы объявляется как возвращающий тип tsm_handler.
record	Указывает, что функция принимает или возвращает неопределённый тип строки.
trigger	Триггерная функция объявляется как возвращающая тип trigger.
event_trigger	Функция событийного триггера объявляется как возвращающая тип event_trigger.
pg_ddl_command	Обозначает представление команд DDL, доступное событийным триггерам.
void	Указывает, что функция не возвращает значение.
unknown	Обозначает ещё не распознанный тип, то есть раскрытое строковое значение.

Имя	Описание
opaque	Устаревший тип, который раньше использовался во многих вышеперечисленных случаях.

Функции, написанные на языке C, объявляются с параметрами или результатами любого из этих типов. Ответственность за безопасное поведение функции с аргументами таких типов ложится на разработчика функции.

Использование псевдотипов, допускается функциями, написанными на процедурных языках, только если это позволяет соответствующий язык. Сейчас большинство процедурных языков запрещают использовать псевдотипы в качестве типа аргумента и позволяют использовать для результатов только типы void и record. Некоторые языки также поддерживают полиморфные функции с типами anyelement, anyarray, anynonarray, anyenum и anyrange.

Псевдотип internal используется в объявлениях функций, предназначенных только для внутреннего использования в СУБД, но не для прямого вызова в запросах SQL. Если у функции есть как хотя бы один аргумент типа internal, её нельзя будет вызывать из SQL. Чтобы сохранить безопасность типа при таком ограничении, есть важное правило: не создавать функцию, возвращающую результат типа internal, если у неё нет ни одного аргумента interna l.

3.2.5. Типы, предназначенные для текстового поиска

СУБД «Jatoba» поддерживает возможность полнотекстового поиска благодаря двум типам данных. Тип tsvector оптимизирует документ, для оптимального поиска текста, а tsquery предоставляет подобный вид запроса текстового поиска.

3.2.5.1. Тип tsvector

Тип tsvector содержит значение в виде отсортированного списка неповторяющихся *лексем*, т. е. слов, *нормализованных* так, что все словоформы сводятся к одной. Сортировка и исключение повторяющихся слов производится автоматически при вводе значения, как показано в этом примере:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
                    tsvector
----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

В апострофы заключаются пробелы или знаки препинания для представления их в виде лексем:

```
SELECT $$the lexeme '    ' contains spaces$$::tsvector;
                tsvector
-----------------------------------------
 '    ' 'contains' 'lexeme' 'spaces' 'the'
```

Строки в примерах, заключены в доллары, чтобы не дублировать апострофы. Дублируются апостроф и обратная косая черта:

```
SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;
                  tsvector
-----------------------------------------------
 'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

Также для лексем можно указать их целочисленные *позиции*:

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11
  rat:12'::tsvector;
                              tsvector
-------------------------------------------------------------------------
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12
  'sat':4
```

Позиция указывает на положение исходного слова. Информация о расположении слов затем может использоваться для *оценки близости*. Позиция может задаваться числом от 1 до 16383; большие значения просто заменяются на 16383. Указание положения лексемы не повторяется.

Также лексемам можно назначить *вес*, через буквы A, B, C или D. Вес D подразумевается по умолчанию и поэтому он не показывается при выводе:

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
        tsvector
----------------------------
 'a':1A 'cat':5 'fat':2B,4C
```

Для придания особого значение словам в заголовке по сравнению со словами в обычном тексте, обычно применяются веса. Назначенным весам можно сопоставить числовые приоритеты в функциях ранжирования результатов.

Важно понимать, что тип tsvector сам по себе не выполняет нормализацию слов; предполагается, что в сохраняемом значении слова уже нормализованы приложением. Например:

```
SELECT 'The Fat Rats'::tsvector;
     tsvector
--------------------
 'Fat' 'Rats' 'The'
```

Большинство англоязычных приложений приведённые выше слова будут считать ненормализованными, но для типа tsvector это не важно. Поэтому исходный документ обычно следует обработать функцией to_tsvector, нормализующей слова для поиска:

```
SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
'fat':2 'rat':3
```

3.2.5.2. Тип tsquery

Значение tsquery содержит искомые лексемы, объединяемые логическими операторами & (И), | (ИЛИ) и ! (НЕ), а также оператором поиска фраз <-> (ПРЕДШЕСТВУЕТ). Также допускается вариация оператора ПРЕДШЕСТВУЕТ вида <*N*>, где *N* - целочисленная константа, задающая расстояние между двумя искомыми лексемами. Запись оператора <-> равнозначна <1>.

Для группировки операторов могут использоваться скобки. Без скобок эти операторы имеют разные приоритеты, в порядке убывания: ! (НЕ), <-> (ПРЕДШЕСТВУЕТ), & (И) и | (ИЛИ).

Несколько примеров:

```
SELECT 'fat & rat'::tsquery;
    tsquery
---------------
 'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
---------------------------
 'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
------------------------
 'fat' & 'rat' & !'cat'
```

Лексемам в tsquery сопоставляются буквы весов, при этом они будут соответствовать только тем лексемам в tsvector, которые имеют какой-либо из этих весов:

```
SELECT 'fat:ab & cat'::tsquery;
    tsquery
```

```
-----------------
 'fat':AB & 'cat'
```

В лексемах tsquery знак * используется для поиска по префиксу:

```
SELECT 'super:*'::tsquery;
  tsquery
-----------
 'super':*
```

Для типа tsquery также используются апострофы. Необходимая нормализация слова должна выполняться до приведения значения к типу tsquery. Для такой нормализации удобно использовать функцию to_tsquery:

```
SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
```

Функция to_tsquery будет обрабатывать префиксы подобно другим словам, поэтому следующее сравнение возвращает true:

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' );
 ?column?
----------
 t
```

так как postgres преобразуется алгоритмом стемминга в postgr:

```
SELECT to_tsvector( 'postgraduate' ), to_tsquery( 'postgres:*' );
  to_tsvector  | to_tsquery
```

```
--------------+------------
 'postgradu':1 | 'postgr':*
```

и эта приставка находится в преобразованной форме слова postgraduate.

3.2.6. Тип XML

За хранение XML-данных отвечает тип xml. Основное преимущество в сравнении с типом text в проверке вводимых значений на правила XML и для работы с ним есть типобезопасные функции. Для использования этого типа дистрибутив должен быть скомпилирован в конфигурации configure --with-libxml.

Правильно оформленные «документы» сохраняются, в соответствии со стандартом XML, содержимое фрагментируется, на менее ограниченные «узлы документа» в модели данных XQuery и XPath. Является ли некоторое значение типа xml полным документом или фрагментом содержимого, позволяет определить выражение xml-значение IS DOCUMENT.

3.2.6.1. Создание XML-значений

Функция xmlparse применяется для получения значения типа xml из текстовой строки:

XMLPARSE ({ DOCUMENT | CONTENT } value)

Примеры:

XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>')

XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')

Допускается специфический синтаксис СУБД «Jatoba» отходящий от стандартов SQL:

xml '<foo>bar</foo>'

'<foo>bar</foo>'::xml

Схема DTD (Document Type Declaration, Объявления типа документа) не подходит для проверки вводимых значений, даже если в них присутствуют ссылка на DTD. В настоящее время в СУБД «Jatoba» не поддерживает других разновидностей схем, например XML Schema.

Получение текстовой строки из xml, реализуется функцией xmlserialize:

XMLSERIALIZE ({ DOCUMENT | CONTENT } значение AS тип)

Допускаются типы - character, character varying или text (или их псевдонимы). SQL предусматривает только один способ преобразования xml в тип текстовых строк, но СУБД «Jatoba» позволяет просто привести значение к нужному типу.

Команда в примере устанавливает параметр конфигурации сеанса «XML option», при преобразовании текстовой строки в тип xml или наоборот без использования функций XMLPARSE и XMLSERIALIZE, при выборе режима DOCUMENT или CONTENT:

SET XML OPTION { DOCUMENT | CONTENT };

или такой командой:

SET xmloption TO { DOCUMENT | CONTENT };

По умолчанию этот параметр имеет значение CONTENT, так что допускаются все формы XML-данных.

3.2.6.2. Обработка кодировки

Разные кодировки символов на стороне сервера и клиента и в XML-данных, гарантируют возникновением проблем. Когда запросы передаются на сервер, а их результаты возвращаются клиенту в обычном текстовом режиме, СУБД «Jatoba» преобразует все передаваемые текстовые данные в кодировку для соответствующей стороны. Обычно это говорит о недействительных объявлениях кодировок, содержащихся в XML-данных, когда текстовая строка преобразуется из одной кодировки

в другую при передаче данных между клиентом и сервером, так как подобные включённые в данные объявления не будут изменены автоматически. Решение проблемы объявления кодировки, заключается в игнорировании текстовых строк и предполагается, что XML-содержимое представлено в текущей кодировке сервера. Это говорит о том, что правильная обработка таких строк с XML-данными возможно при передаче клиентом их в своей текущей кодировке. Для сервера не важно, будет ли клиент для этого преобразовывать документы в свою кодировку, или изменит её, прежде чем передавать ему данные. Вывод значения типа xml не содержат объявления кодировки, а клиент предполагает, что все данные поступают в его текущей кодировке.

Иная ситуация происходит в случае передачи параметров запроса на сервер, и он возвращает результаты клиенту в двоичном режиме, кодировка символов не преобразуется. Делается акцент на объявлении кодировки в XML, а если его нет, то предполагается, что данные закодированы в UTF-8. При выводе в данных будет добавлено объявление кодировки, выбранной на стороне клиента (однако в случае с UTF-8, объявления не будет).

Эффективность обработки XML-данных в СУБД «Jatoba» повышается, когда и в XML-данных, и клиент, и сервер используют одну кодировку. Оптимальным вариантом является, когда на сервере выбрана кодировка UTF-8, так как внутри XML-данные представляются в UTF-8.

Некоторые XML-функции способны работать исключительно с ASCII-данными, если кодировка сервера не UTF-8. Этим известны функции xmltable() и xpath().

3.2.6.3. Обращение к XML-значениям

Для типа xml не определены операторы сравнения. Сравнивая столбец xml с искомым значением отфильтровать строки таблицы невозможно. Поэтому обычно XML-значения должны дополняться отдельным ключевым полем, например, ID. Можно также сравнивать XML-значения, преобразовывая их сначала в текстовые строки, с учётом специфики XML-данных этот метод не работоспособен.

Отсутствие операторов сравнения для типа xml, сделало невозможным создание индекса для столбца этого типа. Вариант ускоренного поиска XML данных возможен, достаточно обойти это ограничение, приводя данные к типу текстовой строки и

проиндексировав эти строки, либо проиндексировав выражение XPath. Сам запрос при этом следует изменить, чтобы поиск выполнялся по индексированному выражению.

3.2.7. Составные типы

Составной тип представляет структуру табличной строки или записи; на практике это просто список из имен полей и соответствующих типов данных. СУБД «Jatoba» использует составные типы так же, как и простые типы. Например, в определении таблицы можно объявить столбец составного типа.

3.2.7.1. Объявление составных типов

Ниже приведены два простых примера определения составных типов:

```
CREATE TYPE complex AS (
    r       double precision,
    i       double precision
);

CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric
);
```

Синтаксис схож с CREATE TABLE, отличие в том, что допускаются только названия полей и их типы, какие-либо ограничения в настоящее время не поддерживаются. Стоит отметить, что ключевое слово AS здесь имеет значение; без него система будет считать, что подразумевается другой тип команды CREATE TYPE, и вернет синтаксическую ошибку.

Определив такие типы, мы можем использовать их в таблицах:

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer
);
```

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);

или функциях:

CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric
*AS 'SELECT $1.price * $2' LANGUAGE SQL;*

SELECT price_extension(item, 10) FROM on_hand;

При каждом создании таблицы, вместе с ней автоматически создаётся составной тип. Этот тип представляет тип строки таблицы, и его именем становится имя таблицы. Например, при выполнении команды:

```
CREATE TABLE inventory_item (
    name            text,
    supplier_id     integer REFERENCES suppliers,
    price           numeric CHECK (price > 0)
);
```

в качестве побочного эффекта будет создан составной тип inventory_item, в точности соответствующий тому, что был показан выше, и использовать его можно так же. Необходимо обратить внимание на то, что в текущей версии есть один недостаток: так как с составным типом не могут быть связаны ограничения, то описанные в определении таблицы ограничения ***не применимы*** к значениям составного типа вне таблицы.

3.2.7.2. Конструирование составных значений

Значение составного типа записывается в виде текстовой константы, его поля нужно заключить в круглые скобки, разделяя их запятыми. Значение любого поля можно заключить в кавычки, а если оно содержит запятые или скобки, это делать обязательно. Таким образом, в общем виде константа составного типа записывается так:

'(значение1 , значение2 , ...)'

Например, эта запись:

'("fuzzy dice",42,1.99)'

будет допустимой для описанного выше типа inventory_item. Чтобы присвоить NULL одному из полей, в соответствующем месте в списке нужно оставить пустое место. Например, эта константа задаёт значение NULL для третьего поля:

'("fuzzy dice",42,)'

Если же вместо NULL требуется вставить пустую строку, нужно записать пару кавычек:

'("",42,)'

Здесь в первом поле окажется пустая строка, а в третьем - NULL.

Константа изначально воспринимается как строка и передаётся процедуре преобразования составного типа. При этом может потребоваться явно указать тип, к которому будет приведена константа.)

Синтаксис выражения ROW используется для конструкций значения составных типов. В большинстве случаев это значительно проще, чем записывать значения в строке, так как при этом не нужно беспокоиться о вложенности кавычек. Мы уже обсуждали этот метод ранее:

ROW('fuzzy dice', 42, 1.99)

ROW('', 42, NULL)

Ключевое слово ROW на самом деле может быть необязательным, если в выражении определяются несколько полей, так что эту запись можно упростить до:

('fuzzy dice', 42, 1.99)

('', 42, NULL)

3.2.7.3. Обращение к составным типам

Обращаясь к полю столбца составного типа, добавляется точка и имя поля после имени столбца нужно, идентично, как указывается столбец после имени таблицы. Отличить эти обращения невозможно, так что часто бывает необходимо использовать скобки, чтобы команда была разобрана правильно. В примере, можно попытаться выбрать поле столбца из тестовой таблицы on_hand таким образом:

SELECT item.name FROM on_hand WHERE item.price > 9.99;

Однако это неработоспособно, так как согласно правилам, SQL имя item здесь воспринимается как имя таблицы, а не столбца в таблице on_hand. Поэтому этот запрос нужно переписать так:

SELECT (item).name FROM on_hand WHERE (item).price > 9.99;

либо указать также и имя таблицы, примерно так:

SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;

Результатом будет правильная интерпретация объекта в скобках как ссылка на столбец item, из которого выбирается поле.

При выборке поля из значения составного типа также могут возникать подобные синтаксические ошибки. Например, чтобы выбрать одно поле из результата функции, возвращающей составное значение, потребуется написать что-то подобное:

SELECT (my_func(...)).field FROM ...

Отсутствие дополнительных скобок в этом запросе приведет к синтаксической ошибке. Специальное имя поля * означает «все поля».

3.2.7.4. Изменение составных типов

Пример ниже представляет правильные команды добавления и изменения значений составных столбцов. Первые команды иллюстрируют добавление или изменение всего столбца:

INSERT INTO mytab (complex_col) VALUES((1.1,2.2));

UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;

В первом примере не используется ключевое слово ROW, а во втором оно есть; присутствовать или отсутствовать оно может в обоих случаях.

Мы можем изменить также отдельное поле составного столбца:

UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;

Необходимо сделать акцент на том, что при этом недопустимо заключать в скобки имя столбца, следующее сразу за предложением SET, но в ссылке на тот же столбец в выражении, находящемся по правую сторону знака равенства, скобки обязательны.

Командой INSERT указываем поля в качестве цели:

INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);

Отсутствие значений для всех полей столбца, заполнит оставшиеся поля значениями NULL.

3.2.7.5. Использование составных типов в запросах

Составные типы в запросах регулируются особыми правилами синтаксиса и поведения. Эти правила образуют полезные конструкции, но необходимо понимать логику их построения, иначе они могут быть неочевидными.

В СУБД «Jatoba» ссылка на имя таблицы в запросе по сути является ссылкой на составное значение текущей строки в этой таблице. Например, имея таблицу inventory_item, показанную в прошлом примере, мы можем написать:

SELECT c FROM inventory_item c;

Результатом этого запроса выдаёт один столбец с составным значением:

```
                c
-----------------------
 ("fuzzy dice",42,1.99)
(1 row)
```

Из примера видно, что простые имена сопоставляются сначала с именами столбцов, а затем с именами таблиц, так что такой результат получается только потому, что в таблицах запроса не оказалось столбца с именем c.

Обычную запись полного имени столбца вида *имя_таблицы.имя_столбца* понимают как использование выбора поля к составному значению текущей строки таблицы. (Для оптимизации на практике это реализовано иначе.)

Когда мы пишем

SELECT c. FROM inventory_item c;*

то, согласно стандарту SQL, должно вернуться содержимое таблицы, развёрнутое в отдельные столбцы:

```
   name    | supplier_id | price
-----------+-------------+-------
 fuzzy dice |          42 |  1.99
(1 row)
```

как с запросом

SELECT c.name, c.supplier_id, c.price FROM inventory_item c;

СУБД «Jatoba» развертывает таким образом любые выражения с составными значениями, однако в примере выше, необходимо заключить в скобки значение, к которому применяется .*, если только это не простое имя таблицы. Например, если myfunc() - функция, возвращающая составной тип со столбцами a, b и c, то эти два запроса выдадут одинаковый результат:

SELECT (myfunc(x)). FROM some_table;*

SELECT (myfunc(x)).a, (myfunc(x)).b, (myfunc(x)).c FROM some_table;

СУБД «Jatoba» разворачивает столбцы фактически переводя первую форму во вторую. Из этого следует, что в данном примере myfunc() будет вызываться три раза для каждой строки и с одним, и с другим синтаксисом. В случае если функция имеет вес, и необходимо избежать лишних вызовов, есть смысл использовать такой запрос:

SELECT m. FROM some_table, LATERAL myfunc(x) AS m;*

Размещение вызова функции в элементе FROM LATERAL гарантирует, что она будет вызываться для строки не более одного раза. Конструкция m.* так же разворачивается в m.a, m.b, m.c, но теперь эти переменные просто ссылаются на выходные значения FROM.

Запись *составное_значение*.* приводит к такому развёртыванию столбцов, когда она присутствует на верхнем уровне выходного списка SELECT, в списке RETURNING команд INSERT/UPDATE/DELETE, в предложении VALUES или в конструкторе строки. Во всех других контекстах, добавление .* к составному значению не изменит это значение, так как это воспринимается как «все столбцы» и поэтому возвращает то же составное значение. Например, если функция somefunc() принимает в качестве аргумента составное значение, эти запросы имеют одинаковую силу:

SELECT somefunc(c.) FROM inventory_item c;*

SELECT somefunc(c) FROM inventory_item c;

Оба случая в которых текущая строка таблицы inventory_item передаётся функции как один аргумент с составным значением. Учитывая, что дополнение .* в этих случаях не играет роли, его использование принимается за хороший стиль, так как это явно указатель использования составного значения. Частный случай, если анализатор запроса воспримет c в записи c.* как ссылку на имя или псевдоним таблицы, а не имя

столбца, что избавляет от неоднозначности; тогда как без .* неочевидно, означает ли с имя таблицы или имя столбца, и на самом деле при наличии столбца с именем с будет выбрано второй вариант.

Следующий пример демонстрирует одинаковые запросы, действующие одинаково:

*SELECT * FROM inventory_item c ORDER BY c;*

*SELECT * FROM inventory_item c ORDER BY c.*;*

*SELECT * FROM inventory_item c ORDER BY ROW(c.*);*

Все эти предложения ORDER BY обращаются к составному значению строки, вследствие чего строки сортируются. Однако, если inventory_item содержит столбец с именем с, первый запрос будет отличаться от других, так как в нём выполнится сортировка только по данному столбцу. С приведенными в прошлом примере именами столбцов предыдущим запросам также равнозначны следующие:

*SELECT * FROM inventory_item c ORDER BY ROW(c.name, c.supplier_id, c.price);*

*SELECT * FROM inventory_item c ORDER BY (c.name, c.supplier_id, c.price);*

Последний запрос использует конструктор строки, в котором опущено ключевое слово ROW.)

Синтаксис, связан с составными значениями, это говорит о том, что мы можем использовать *функциональную запись* для извлечения поля составного значения. Можно объяснить это тем, что записи *поле*(*таблица*) и *таблица.поле* взаимозаменяемы. Например, следующие запросы равнозначны:

SELECT c.name FROM inventory_item c WHERE c.price > 1000;

SELECT name(c) FROM inventory_item c WHERE price(c) > 1000;

Кроме того, имея функцию, принимающую один аргумент составного типа, возможно вызвать её в любой записи. Все эти запросы равносильны:

SELECT somefunc(c) FROM inventory_item c;

SELECT somefunc(c.) FROM inventory_item c;*

SELECT c.somefunc FROM inventory_item c;

Эта особенность, заключающаяся в равнозначности записи с полем и функциональной записи позволяет использовать с составными типами функции, реализующие «вычисляемые поля». Использование последнего из предыдущих запросов, не вынуждает приложение знать, что фактически somefunc - не настоящий столбец таблицы.

Учитывая все эти факторы, не имеет смысла называть функцию, принимающей один аргумент составного типа, тем же именем, что и одно из полей данного составного типа. В случае неопределенности прочтение имени поля будет выбрано при использовании синтаксиса обращения к полю, а прочтение имени функции - если используется синтаксис вызова функции. Однако в СУБД «Jatoba» до 11 версии предпочтение отдавалось прочтению имени поля, если только синтаксис вызова не подталкивал к прочтению имени функции. Есть способ принудительного выбора прочтения имени функции, в предыдущих версиях надо было дополнить это имя схемой, то есть написать *схема*.*функция*(*составное_значение*).

3.2.7.6. Синтаксис вводимых и выводимых значений составного типа

Внешнее текстовое представление составного значения построено из записи элементов, интерпретируемых согласно правил ввода/вывода для соответствующих типов полей, и оформления структуры составного типа. Оформление заключается в круглых скобках ((и)) окружающих всё значение, и запятых (,) между его элементами. Символы пробела за скобками игнорируются, но внутри они считаются частью соответствующего элемента и могут учитываться или не учитываться в зависимости от правил преобразования вводимых данных для типа этого элемента. Например, в записи:

'(42)'

пробелы не будут учитываться, если соответствующее поле имеет целочисленный тип, но не текстовый.

Исходя из вышесказанного, записывая составное значение, есть возможность каждый его элемент заключить в кавычки. Требуется делать так, если при разборе этого значения без кавычек возможна неопределенность. Например, в кавычки нужно заключать элементы, содержащие скобки, кавычки, запятую или обратную косую черту. Способ включения в поле составного значения, заключённое в кавычки, таких символов,

как кавычки или обратная косая черта, необходимо перед ними добавить обратную косую черту. Другой вариант использования, обойтись без кавычек, защищая таким образом все символы в данных, которые могут быть восприняты как часть синтаксиса составного значения, с помощью спецпоследовательностей.

Значение NULL в этой записи представлено пустым местом. Чтобы ввести именно пустую строку, а не NULL, нужно написать "".

Функция вывода составного значения заключает значения полей в кавычки, если они представляют собой пустые строки, либо содержат скобки, запятые, кавычки или обратную косую черту, либо состоят из одних пробелов. Кавычки и обратная косая черта, заключённые в значения полей, при выводе дублируются.

Необходимо помнить, что написанная SQL-команда в первую очередь интерпретируется текстовой строкой, и только потом как составное значение. Как следствие число символов обратной косой черты увеличивается в два раза. Например, чтобы ввести в поле составного столбца значение типа text с обратной косой чертой и кавычками, команду нужно будет записать так:

INSERT ... VALUES ('("\"\\")');

Сначала обработчик спецпоследовательностей удаляет один уровень обратной косой черты, так что анализатор составного значения получает на вход ("\"\\"). Далее, он передаёт эту строку процедуре ввода значения типа text, там ее преобразуют в "\. Во избежание такого дублирования спецсимволов строки можно заключать в доллары.

Конструктор ROW упрощает запись составных значений в командах SQL. В ROW отдельные значения элементов записываются так же, как если бы они не были членами составного выражения.

3.2.8. Типы данных JSON

Существуют два типа данных JSON: json и jsonb. Тип json сохраняет точную копию введённого текста. Данные jsonb разбираются в двоичном формате. Кроме того, jsonb поддерживает индексацию. Они принимают на вход почти идентичные наборы значений, а отличаются лишь эффективностью.

Типы JSON соответственно хранят данные JSON (JavaScript Object Notation, Запись объекта JavaScript) согласно стандарту, RFC 7159. Такие данные можно хранить и

в типе text, однако приоритет типов JSON в проверке на соответствие вводимого значения формату JSON.

Вследствие того, что тип json сохраняет точную копию введённого текста, сохраняются семантически незначащие пробелы между символами, а также порядок ключей в JSON-объектах.

Большинство приложений хранят данные JSON в типе jsonb.

СУБД «Jatoba» ограничена использованием одной кодировки символов в базе данных. Это означает, что если кодировка базы данных не UTF-8, данные JSON не будут полностью соответствовать спецификации. При этом не будет возможности вставить символы, непредставимые в кодировке сервера, и наоборот, допустимыми будут символы, представимые в кодировке сервера, но не в UTF-8.

Многие функции обработки JSON, преобразуют спецпоследовательности Unicode в обычные символы, поэтому возвращают подобные ошибки, даже если им на вход поступает тип json, а не jsonb. Функция ввода в тип json не производит этих проверок, это и позволяет просто сохранять в JSON спецкоды Unicode в базе данных с кодировкой отличной от UTF8. Настоятельно рекомендуется, избегать смешения спецкодов Unicode в JSON с кодировкой базой данных не UTF8.

Таблица 3.15 показывает какие есть ограничения в формате ввода примитивных типов JSON, не актуальные для соответствующих типов СУБД «Jatoba».

Таблица 3.15 – Примитивные типы JSON и соответствующие им типы СУБД «Jatoba»

Примитивный тип JSON	**Тип СУБД «Jatoba»**	**Замечания**
string	text	\u0000 не допускается, как не ASCII символ, если кодировка базы данных не UTF8
number	numeric	Значения NaN и infinity не допускаются
boolean	boolean	Допускаются только варианты true и false (в нижнем регистре)
null	(нет)	NULL в SQL имеет другой смысл

3.2.8.1. Синтаксис вводимых и выводимых значений JSON

Стандартом RFC 7159 продиктованы правила синтаксиса ввода/вывода типов данных JSON.

Примеры допустимых выражений с типом json (или jsonb):

-- Простое скалярное/примитивное значение

-- Простыми значениями являются числа, строки в кавычках, значения true, false или null

SELECT '5'::json;

-- Массив из нуля и более элементов

SELECT '[1, 2, "foo", null]'::json;

-- Объект, содержащий пары ключей и значений

-- Заметьте, что ключи объектов - это всегда строки в кавычках

SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;

-- Массивы и объекты могут вкладываться произвольным образом

SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;

Как было сказано ранее, когда значение JSON вводится и затем выводится без дополнительной обработки, тип json выводит тот же текст, что поступил на вход, а jsonb не сохраняет семантически незначащие детали, такие как пробелы. Например, посмотрите на эти различия:

SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;

json

{"bar": "baz", "balance": 7.77, "active":false}

(1 row)

SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;

jsonb

{"bar": "baz", "active": false, "balance": 7.77}

(1 row)

Первая семантически незначимая деталь, заслуживающая внимания: с jsonb числа выводятся по правилам нижележащего типа numeric. На практике это означает, что числа, заданные в записи с E, будут выведены без неё, например:

SELECT '{"reading": 1.230e-5}'::json, '{"reading": 1.230e-5}'::jsonb;

json | jsonb

----------+----------

{"reading": 1.230e-5} | {"reading": 0.00001230}

(1 row)

Однако, как видно из этого примера, jsonb сохраняет конечные нули дробного числа, хотя они и не имеют семантической значимости, в частности для проверки на равенство.

3.2.8.2. Проверки на вхождение и существование jsonb

Проверка вхождения определяет, входит ли один документ jsonb в другой. Отличающая особенность от типа json, проверка *вхождения* - важная особенность типа jsonb. Примеры ниже показывают, как возвращается истинное значение (кроме упомянутых исключений):

-- Простые скалярные/примитивные значения включают только одно идентичное значение:

SELECT '"foo"'::jsonb @> '"foo"'::jsonb;

-- Массив с правой стороны входит в массив слева:

SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;

-- Порядок элементов в массиве не важен, поэтому это условие тоже выполняется:

SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;

-- А повторяющиеся элементы массива не имеют значения:

SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;

-- Объект с одной парой справа входит в объект слева:

SELECT '{"product": "Jatoba", "version": 9.4, "jsonb": true}'::jsonb @> '{"version": 9.4}'::jsonb;

-- Массив справа не считается входящим в

-- массив слева, хотя в последний и вложен подобный массив:

SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb; -- выдаёт false

-- Но если добавить уровень вложенности, проверка на вхождение выполняется:

SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;

-- Аналогично, это вхождением не считается:

SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb; -- выдаёт false

-- Ключ с пустым объектом на верхнем уровне входит в объект с таким ключом:

SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;

Проверка состоит в, что входящий объект должен соответствовать объекту, содержащему его, по структуре и данным, возможно, после исключения из содержащего объекта лишних элементов массива или пар ключ/значение. Проверка на вхождение не привязана к порядку элементов массива, а повторяющиеся элементы массива считаются только один раз.

В качестве особого исключения для требования идентичности структур, массив может содержать примитивное значение:

-- В этот массив входит примитивное строковое значение:

SELECT '["foo", "bar"]'::jsonb @> '"bar"'::jsonb;

-- Это исключение действует только в одну сторону -- здесь вхождения нет:

SELECT '"bar"'::jsonb @> '["bar"]'::jsonb; -- выдаёт false

Для типа jsonb введён также оператор *существования*, который является альтернативой на тему вхождения: он проверяет, является ли строка (заданная в виде значения text) ключом объекта или элементом массива на верхнем уровне значения jsonb. В следующих примерах возвращается истинное значение (кроме упомянутых исключений):

-- Строка существует в качестве элемента массива:

SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';

-- Строка существует в качестве ключа объекта:

SELECT '{"foo": "bar"}'::jsonb ? 'foo';

-- Значения объектов не рассматриваются:

SELECT '{"foo": "bar"}'::jsonb ? 'bar'; -- выдаёт false

-- Как и вхождение, существование определяется на верхнем уровне:

SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar'; -- выдаёт false

-- Строка считается существующей, если она соответствует примитивной строке JSON:

SELECT '"foo"'::jsonb ? 'foo';

Поиск элемента не будет линейным, т.к. объекты JSON для проверок на существование и вхождение со множеством ключей или элементов подходят больше, чем массивы, так как, в отличие от массивов, они внутри оптимизируются для поиска.

Так как вхождение в JSON проверяется с учётом вложенности, правильно сформированный запрос может заменить явную выборку внутренних объектов. Например, предположим, что у нас есть столбец doc, содержащий объекты на верхнем уровне, и большинство этих объектов содержит поля tags с массивами вложенных объектов. Данный запрос найдёт записи, в которых вложенные объекты содержат ключи "term":"paris" и "term":"food", и при этом пропустит такие ключи, находящиеся вне массива tags:

SELECT doc->'site_name' FROM websites

WHERE doc @> '{"tags":[{"term":"paris"}, {"term":"food"}]}';

Этого же результата можно добиться, например, так:

SELECT doc->'site_name' FROM websites

WHERE doc->'tags' @> '[{"term":"paris"}, {"term":"food"}]';

Но данный подход менее гибкий и часто также менее эффективный.

С другой стороны, оператор существования JSON не учитывает вложенность: он будет искать заданный ключ или элемент массива только на верхнем уровне значения JSON.

3.2.8.3. Трансформации

Для различных процедурных языков существуют дополнительные расширения, реализующие трансформации для типа jsonb.

Названия расширений для PL/Perl: jsonb_plperl и jsonb_plperlu. Когда они используются, значения jsonb передаются в соответствующие структуры Perl: массивы, хеши или скаляры.

Названия расширений для PL/Python jsonb_plpythonu, jsonb_plpython2u и jsonb_plpython3u. Когда они используются, значения jsonb передаются в соответствующие структуры Python: массивы, хеши или скаляры.

3.2.8.4. Индексация jsonb

Поиск ключей или пар ключ/значение в большом количестве документов jsonb сделает эффективным применение индексов GIN. Для этого предлагаются два «класса операторов» GIN, выбирая между производительностью и гибкостью.

Операторы класса GIN по умолчанию для jsonb поддерживает запросы с операторами существования ключа на верхнем уровне (?, ?& и ?|) и оператором существования пути/значения (@>). Пример создания индекса с этим классом операторов:

CREATE INDEX idxgin ON api USING GIN (jdoc);

Дополнительный класс операторов GIN jsonb_path_ops поддерживает индексацию только для оператора @>. Пример создания индекса с этим классом операторов:

CREATE INDEX idxginp ON api USING GIN (jdoc jsonb_path_ops);

Рассмотрим пример таблицы, в которой хранятся документы JSON, получаемые от сторонней веб-службы, с документированным определением схемы. Типичный документ:

```
{
   "guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",
   "name": "Angela Barton",
   "is_active": true,
   "company": "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 сохраняем эти документы в таблице api, в столбце jdoc типа jsonb. Если по этому столбцу создаётся GIN-индекс, он может применяться в подобных запросах:

```
-- Найти документы, в которых ключ "company" имеет значение "Magnafone"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"company": "Magnafone"}';
```

Однако, в следующих запросах он не будет использоваться, потому что, несмотря на то, что оператор? - индексируемый, он применяется не к индексированному столбцу jdoc непосредственно:

```
-- Найти документы, в которых ключ "tags" содержит ключ или элемент массива "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ? 'qui';
```

Стоит отметить, что правильно применяя индексы выражений, в этом запросе можно задействовать индекс. Если запрос определённых элементов в ключе "tags" выполняется часто, вероятно стоит определить такой индекс:

CREATE INDEX idxgintags ON api USING GIN ((jdoc -> 'tags'));

Теперь предложение WHERE jdoc -> 'tags' ? 'qui' будет выполняться как применение индексируемого оператора ? к индексируемому выражению jdoc -> 'tags'.

Ещё один подход к использованию проверок на существование:

-- Найти документы, в которых ключ "tags" содержит элемент массива "qui"

SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"tags": ["qui"]}';

Этот запрос может задействовать простой GIN-индекс по столбцу jdoc.

Класс операторов jsonb_path_ops поддерживает только запросы с оператором @>.Индекс jsonb_path_ops обычно гораздо меньше индекса jsonb_ops для тех же данных и более точен при поиске.

Различие между GIN-индексами jsonb_ops и jsonb_path_ops состоит в том, что для первых создаются независимые элементы индекса для каждого ключа/значения в данных, тогда как для вторых создаются элементы только для значений. На деле, каждый элемент индекса jsonb_path_ops это хеш значения и ключа(ей), приводящего к нему; например, при индексации {"foo": {"bar": "baz"}} будет создан один элемент индекса с хешем, рассчитанным по всем трём значениям: foo, bar и baz.

Класс jsonb_path_ops имеет ряд недостатков, в частности то, что он не учитывает в индексе структуры JSON, не содержащие никаких значений {"a": {}}.

Тип jsonb также поддерживает индексы btree и hash. Они применимы, только если требуется проверка равенства JSON-документов в целом. Нет необходимости в порядке сортировки btree для типа jsonb, но для полноты он приводится ниже:

Объект > Массив > Логическое значение > Число > Строка > Null

Объект с n парами > Объект с n - 1 парами

Массив с n элементами > Массив с n - 1 элементами

Объекты с равным количеством пар сравниваются в таком порядке:

ключ-1, значение-1, ключ-2 ...

Заметьте, что ключи объектов сравниваются по порядку при хранении; в частности, из-за того, что короткие ключи хранятся раньше длинных, результаты могут оказаться несколько не предсказуемыми:

{ "aa": 1, "c": 1} > {"b": 1, "d": 1}

Массивы с равным числом элементом упорядочиваются аналогично:

элемент-1, элемент-2 ...

Правила сравнения распространяются на примитивные значения JSON. Строки сравниваются с учётом порядка сортировки по умолчанию в текущей базе данных.

3.2.9. Диапазонные типы

Диапазонные типы представляют диапазоны значений некоторого типа данных. К примеру, диапазон типа timestamp может представлять временной интервал, когда зарезервирован зал заседаний. Выбранным типом данных будет tsrange, а подтипом - timestamp. Подтип должен иметь возможность упорядочиваться полностью, для однозначного определения, нахождения значения по отношению к диапазону: внутри, до или после него.

Диапазонные типы используются для того, чтобы позволять представить множество возможных значений одной структурой данных и чётко выразить такие понятия, как пересечение диапазонов. Их практическое применение - использовать диапазоны даты и времени для составления расписания, но также полезными могут оказаться диапазоны цен, интервалы измерений и т. д.

3.2.9.1. Встроенные диапазонные типы

В дистрибутиве СУБД «Jatoba» встроены следующие диапазонные типы:

- int4range - диапазон подтипа integer
- int8range - диапазон подтипа bigint
- numrange - диапазон подтипа numeric
- tsrange - диапазон подтипа timestamp without time zone
- tstzrange - диапазон подтипа timestamp with time zone
- daterange - диапазон подтипа date

Есть возможность определять собственные типы; подробнее это описано в CREATE TYPE.

3.2.9.2. Примеры

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');

-- Вхождение
    SELECT int4range(10, 20) @> 3;

-- Перекрытие
    SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- Получение верхней границы
    SELECT upper(int8range(15, 25));

-- Вычисление пересечения
    SELECT int4range(10, 20) * int4range(15, 25);

-- Является ли диапазон пустым?
    SELECT isempty(numrange(1, 5));
```

3.2.9.3. Включение и исключение границ

Каждый непустой диапазон имеет две границы, верхнюю и нижнюю, также включает все точки между этими значениями. Точка, лежащая на границе, если диапазон включает эту границу, захватывается диапазоном. От обратного, если диапазон не включает границу, точку, лежащую на этой границе, диапазон не захватывает.

В текстовой записи диапазона захват нижней границы обозначается символом «[», а исключением - символом «(». Для верхней границы захват обозначается аналогично, символом «]», а исключение - символом «)».

Для проверки, захвачена ли нижняя или верхняя граница в диапазон, предназначены функции lower_inc и upper_inc, соответственно.

3.2.9.4. Неограниченные (бесконечные) диапазоны

Для того, чтобы захватить все значения ниже верхней границы, опускается нижняя граница диапазона и определяет тем самым диапазон, например: (,3]. Если не определена верхняя граница, диапазон захватит все значения, лежащие выше нижней границы. Если же опущена и нижняя, и верхняя границы, такой диапазон будет включать все возможные значения своего подтипа. При указании отсутствующей границы как находящейся в диапазоне автоматически добавится в исключения; например, [,] преобразуется в (,). Отсутствие значений можно принимать как плюс/минус бесконечность, однако это особые значения диапазонного типа, которые захватывают и возможные для подтипа значения плюс/минус бесконечность.

Для подтипов, в которых есть понятие «бесконечность», infinity может использоваться в качестве явного значения границы. При этом, например, в диапазон [today,infinity) с подтипом timestamp не будет входить специальное значение infinity данного подтипа, однако это значение будет входить в диапазон [today,infinity], как и в диапазоны [today,) и [today,].

Проверка, определена ли верхняя или нижняя граница, осуществляется функциями lower_inf и upper_inf, соответственно.

3.2.9.5. Ввод/вывод диапазонов

Значения, ктоторые вводятся в диапазон должны записываться согласно формам:

(*нижняя-граница*,*верхняя-граница*)

(*нижняя-граница*,*верхняя-граница*]

[*нижняя-граница*,*верхняя-граница*)

[*нижняя-граница*,*верхняя-граница*]

empty

Пример выше описывает, что тип скобок определяет, захвачены ли в диапазон соответствующие границы. Необходимо сделать акцент на том, что последняя форма, содержащая слово empty, определяет пустой диапазон.

Допускается, что *нижняя-граница* будет строкой с допустимым значением подтипа или будет пустой. Подобным образом, *верхняя-граница* задается одним из значений подтипа или будет неопределённой.

Возможно заключить в кавычки (") любое значение диапазона. Использование кавычек необходимо, если значение содержит круглые или квадратные скобки, запятые, кавычки или обратную косую черту, так как эти символы не должны рассматриваться как часть синтаксиса диапазона. Для случаев, когда необходимо включить в значение диапазона, заключённое в кавычки, такие символы, как кавычки или обратная косая черта, перед ними ставится обратная косая черта. (Ранее упоминалось, что продублированные кавычки в значении диапазона, заключённого в кавычки, воспринимаются как одинарные, подобно апострофам в строках SQL.) Наличие кавычек не обязательно, если защитить все символы в данных, которые могут быть восприняты как часть синтаксиса диапазона, с помощью спецпоследовательностей. Границей можно представить пустую строку, нужно ввести "", иначе пустая строка без кавычек будет означать отсутствие границы.

Символы пробелов до и после определения диапазона игнорируются, но, когда они присутствуют внутри скобок, они воспринимаются как часть значения верхней или нижней границы.

Правила схожи с правилами записи значений для полей составных типов.

Примеры:

```
-- в диапазон включается 3, не включается 7 и включаются все точки между ними
    SELECT '[3,7)'::int4range;

-- в диапазон не включаются 3 и 7, но включаются все точки между ними
    SELECT '(3,7)'::int4range;

-- в диапазон включается только одно значение 4
    SELECT '[4,4]'::int4range;

-- диапазон не включает никаких точек (нормализация заменит его определение
--  на 'empty')
    SELECT '[4,4)'::int4range;
```

3.2.9.6. Конструирование диапазонов

Для каждого диапазонного типа определена функция конструктора, имеющая то же имя, что и данный тип. Удобнее пользоваться таким конструктором, чем записывать текстовую константу диапазона, так как это освобождает от дополнительных кавычек. В функцию конструктора можно внести два или три параметра. Два параметра дают создать диапазон в стандартной форме, вариант с тремя параметрами включает границы определения третьим параметром. Третий параметр должен содержать одну из строк: «()», «(]», «[)» или «[]». Например:

-- Полная форма: нижняя граница, верхняя граница и текстовая строка, определяющая
-- включение/исключение границ.

SELECT numrange(1.0, 14.0, '(]');

-- Если третий аргумент опущен, подразумевается '[)'.

SELECT numrange(1.0, 14.0);

-- Хотя здесь указывается '(]', при выводе значение будет приведено к
-- каноническому виду, так как int8range - тип дискретного диапазона (см. ниже).

SELECT int8range(1, 14, '(]');

-- Когда вместо любой границы указывается NULL, соответствующей границы
-- у диапазона не будет.

SELECT numrange(NULL, 2.2);

3.2.9.7. Типы дискретных диапазонов

Диапазон называют дискретным, подтип которого имеет однозначно определённый «шаг», как например для типов integer и date. Значения этих двух типов можно окажутся соседними, если между ними нет никаких других значений. Такого не происходит в непрерывных диапазонах, в них всегда (или почти всегда) можно найти ещё одно значение между соседними данными. Например, непрерывным диапазоном будет диапазон с подтипами numeric и timestamp.

Допускается считать дискретным подтип диапазона, в котором чётко определены понятия «следующего» и «предыдущего» элемента для каждого значения. Такие определения позволяют исключать границы диапазона из включаемых, выбирая следующий или предыдущий элемент вместо заданного значения. Например, диапазоны целочисленного типа [4,8] и (3,9) описывают одно и то же множество значений; но для диапазона подтипа numeric это не так.

Для типа дискретного диапазона определяется функция *канонизации*, которая учитывает размер шага для данного подтипа. У этой функции есть задача, заключающаяся в преобразовании равнозначных диапазонов к единственному представлению, в частности нормализация включаемых и исключаемых границ. Бывает так, что функция канонизации не определена. В таком случае диапазоны с различным определением будут всегда считаться разными, даже те, которые представляют одно множество значений.

Встроенные типы int4range, int8range и daterange в каноническом представлении включает нижнюю границу и не включает верхнюю; то есть диапазон приводится к виду [). Однако для нестандартных типов можно использовать и другие соглашения.

3.2.9.8. Определение новых диапазонных типов

Пользователи имеют возможность определять собственные диапазонные типы. Это применяется, если нужно использовать диапазоны с подтипами, для которых нет встроенных диапазонных типов. Например, можно определить новый тип диапазона для подтипа float8:

```
CREATE TYPE floatrange AS RANGE (
   subtype = float8,
   subtype_diff = float8mi
);

SELECT '[1.234, 5.678]'::floatrange;
```

Так как для float8 однозначное значение «шага» не определено, функция канонизации в данном примере не задаётся.

За счет определения собственного диапазонного типа, пользователь может выбрать другие правила сортировки или класс оператора B-дерева для его подтипа, это позволит изменить порядок значений, от которого зависит, какие значения попадают в заданный диапазон.

Если подтип рассматривается как дискретный, а не непрерывный, командой CREATE TYPE следует задать функцию канонизации. Эта функция будет принимает значение диапазона, а возвращает равнозначное значение, но, возможно, изменяя границы и с другим форматированием. Для двух диапазонов, представляющих одно множество значений, например, целочисленные диапазоны [1, 7] и [1, 8), функция канонизации должна выдавать один результат. Нет никакой разницы, в том, какое из представлений будет считаться каноническим - важно, чтобы два равнозначных диапазона, с различным форматированием, всегда преобразовывались в одно значение с одинаковым форматированием. Дополнительно функция канонизации помимо исправления формата включаемых/исключаемых границ, может округлять значения границ, в случае превышения размером шага точности хранения подтипа. Например, в типе диапазона для подтипа timestamp можно определить размер шага, равный одному часу, тогда функция канонизации должна будет округлить границы, заданные, например, с точностью до минут, либо вместо этого выдать ошибку.

Помимо этого, для любого диапазонного типа, ориентированного на использование с индексами GiST или SP-GiST, должна быть определена разница значений подтипов, функция subtype_diff. Эта функция принимает на вход два значения подтипа и возвращает их разницу (т. е. *X* минус *Y*) в значении типа float8. Функция subtype_diff, насколько это возможно, должна быть согласована с порядком сортировки, вытекающим из выбранных правил сортировки и класса оператора; то есть, её результат должен быть положительным, если согласно порядку сортировки первый её аргумент больше второго.

Ещё один, нераспространенный пример функции subtype_diff:

CREATE FUNCTION time_subtype_diff(x time, y time) RETURNS float8 AS

'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT IMMUTABLE;

CREATE TYPE timerange AS RANGE (

subtype = time,

subtype_diff = time_subtype_diff

);

SELECT '[11:10, 23:00]'::timerange;

3.2.9.9. Индексация

Для столбцов, имеющих диапазонный тип, можно создать индексы GiST и SP-GiST. Например, так создаётся индекс GiST:

CREATE INDEX reservation_idx ON reservation USING GIST (during);

Индекс GiST или SP-GiST помогает ускорить запросы со следующими операторами: =, &&, <@, @>, <<, >>, -|-, &< и &> .

Более того, для таких столбцов есть возможность создать индексы на основе хеша и B-деревьев. Для индексов таких типов полезен по сути только один оператор диапазона - равно. Порядок сортировки B-дерева определяется для значений диапазона соответствующими операторами < и >, но порядок сортировки способен быть произвольным и не критичен в реальности. Поддержка B-деревьев и хешей с помощью диапазонных типов нужна в первую очередь для сортировки и хеширования при выполнении запросов, но не для создания самих индексов.

3.2.9.10. Ограничения для диапазонов

UNIQUE, являясь естественным ограничением для скалярных значений, не подходит диапазонным типам. Вместо этого чаще подходят ограничения-исключения. Такие ограничения позволяют, например, определить условие «непересечения» диапазонов. Например:

CREATE TABLE reservation (

during tsrange,

EXCLUDE USING GIST (during WITH &&)

);

Это ограничение не позволит одновременно сохранить в таблице несколько диапазонов, которые накладываются друг на друга:

INSERT INTO reservation VALUES

('[2010-01-01 11:30, 2010-01-01 15:00)');

INSERT 0 1

INSERT INTO reservation VALUES

('[2010-01-01 14:45, 2010-01-01 15:45)');

ОШИБКА: конфликтующее значение ключа нарушает ограничение-исключение "reservation_during_excl"

ПОДРОБНОСТИ: Ключ (during)=(["2010-01-01 14:45:00","2010-01-01 15:45:00")) конфликтует с существующим ключом (during)=(["2010-01-01 11:30:00","2010-01-01 15:00:00"))

Максимальная гибкость в ограничении-исключении достигается сочетанием простых скалярных типов данных с диапазонами, используя расширение btree_gist. Например, если btree_gist установлено, следующее ограничение не допустит пересекающиеся диапазоны, только если совпадают также и номера комнат:

```
CREATE EXTENSION btree_gist;
CREATE TABLE room_reservation (
    room text,
    during tsrange,
    EXCLUDE USING GIST (room WITH =, during WITH &&)
);

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');
ОШИБКА:  конфликтующее значение ключа нарушает ограничение-исключение
  "room_reservation_room_during_excl"
ПОДРОБНОСТИ:  Ключ (room, during)=(123A, [ 2010-01-01 14:30:00,
```

2010-01-01 15:30:00)) конфликтует

с существующим ключом (room, during)=(123A, ["2010-01-01 14:00:00","2010-01-01 15:00:00")).

INSERT INTO room_reservation VALUES

('123B', '[2010-01-01 14:30, 2010-01-01 15:30)');

INSERT 0 1

Перечень сокращений

DDL	–	Data Manipulation Language, язык манипулирования данными
SQL	–	Structured Query Language (язык структурированных запросов)
БД	–	База данных
ОС	–	Операционная система
СУБД	–	Система управления базами данных
ФСТЭК России	–	Федеральная служба по техническому и экспортному контролю России
ЭВМ	–	Электронно-вычислительная машина

Приложение А
(обязательное)
Описание синтаксиса команды: «CREATE TABLE» и расшифровка его параметров

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY } | UNLOGGED ] TABLE [ IFNOTEXISTS ] имя_таблицы( [  { имя_столбцатип_данных[ COLLATEтип_сортировки ]
[ ограничение_столбца  [ ... ] ]    | ограничение_таблицы    | LIKEисх_табл [вариант_копир... ] }    [, ... ]] )
[ INHERITS ( родит_табл [, ... ] ) ]
[ PARTITIONBY { RANGE | LIST | HASH } ( { имя_столбца | ( выражение ) } [ COLLATEтип_сортировки ] [ по_классу ] [, ... ] ) ] [ USINGметод]
[ WITH ( параметры_хранения [= значение] [, ... ] ) ]
[ ONCOMMIT { PRESERVEROWS | DELETEROWS | DROP } ]
[ TABLESPACEнаименование_табличного_пространства]


CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY } | UNLOGGED ] TABLE [ IFNOTEXISTS ] имя_таблицыOFимя_типа [ (  { имя_столбца [ WITHOPTIONS ] [
ограничение_столбца  [ ... ] ]    | ограничение_таблицы }    [, ... ]) ]
[ PARTITIONBY { RANGE | LIST | HASH } ( { имя_столбца | (выражение ) } [ COLLATEтип_сортировки ] [по_классу ] [, ... ] ) ] [ USINGметод ]
[ WITH ( параметры_хранения [= значение] [, ... ] ) ]
[ ONCOMMIT { PRESERVEROWS | DELETEROWS | DROP } ]
[ TABLESPACEнаименование_табличного_пространства]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY } | UNLOGGED ] TABLE [ IFNOTEXISTS ] имя_таблицыPARTITIONOFродит_табл [ (  { имя_столбца [
WITHOPTIONS ] [ограничение_столбца[ ... ] ]    | ограничение_таблицы }    [, ... ]) ] { FORVALUESспец_связ_разд | DEFAULT }
[ PARTITIONBY { RANGE | LIST | HASH } ( { имя_столбца | (выражение ) } [ COLLATEтип_сортировки ] [по_классу ] [, ... ] ) ] [ USINGметод ]
[ WITH (параметры_хранения [=значение] [, ... ] )]
[ ONCOMMIT { PRESERVEROWS | DELETEROWS | DROP } ]
[ TABLESPACEнаименование_табличного_пространства ]
```

В данных синтаксисах:

а) *ограничение_столбца* это:

```
[ CONSTRAINT имя_ограничения]
{ NOT NULL |
  NULL |
  CHECK ( выражение ) [ NO INHERIT ] |
```

```
  DEFAULT выражение_по_умолчанию |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( вариант_последовательности ) ] |
  UNIQUE параметры_индекса |
  PRIMARY KEY параметры_индекса |
  REFERENCES целев_таблица [ ( целев_столбец ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]    [ ON DELETE действие ] [ ON UPDATE действие ] }
[ DEFERRABLE | NOT DEFERRABLE ]
[ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

б) *ограничение_таблицы* это:

```
[ CONSTRAINT имя_ограничения ]
{ CHECK ( выражение ) [ NO INHERIT ] |
UNIQUE ( имя_столбца [, ... ] ) параметры_индекса |
PRIMARYKEY ( имя_столбца [, ... ] ) параметры_индекса |
EXCLUDE [ USINGиндексный_метод ] ( искл_элементWITHоператор [, ... ] ) параметры_индекса [ WHERE ( утверждение ) ] |
FOREIGNKEY ( имя_столбца [, ... ] ) REFERENCESцелев_таблица [ ( целев_столбец [, ... ] ) ]    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE
действие ] [ ON UPDATE действие ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

в) *вариант_копир* это:

```
{ INCLUDING | EXCLUDING } { COMMENTS | CONSTRAINTS | DEFAULTS | IDENTITY | INDEXES | STATISTICS | STORAGE | ALL }
```

г) *спец_связ_разд* это:

```
IN ( { числ_литер| строк_литер | TRUE | FALSE | NULL } [, ...] ) |
FROM ( { числ_литер | строк_литер | TRUE | FALSE | MINVALUE | MAXVALUE } [, ...] )
  TO ( { числ_литер | строк_литер | TRUE | FALSE | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS числ_литер, REMAINDER числ_литер )
```

д) *параметры_индекса* в ограничении UNIQUE, PRIMARY KEY, and EXCLUDE:

```
[ INCLUDE ( имя_столбца [, ... ] ) ]
```

```
[ WITH ( параметры_хранения [= значение] [, ... ] ) ]
[ USINGINDEXTABLESPACEнаименование_табличного_пространства]
```

е) *искл_элемент* в ограничении EXCLUDE:

```
{ имя_столбца | ( выражение ) } [по_классу ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

Расшифровка параметров команды «CREATE TABLE» представлена в таблице А.1.

Таблица А.1 – Расшифровка параметров команды «CREATE TABLE»

Параметр/выражение	Описание
TEMPORARY	Данный параметр используется для создание временной таблицы. Она автоматически удаляется либо после завершение текущий транзакции, либо после завершения сеанса. Если постоянная таблица имеет, то же имя, что и временная, то тогда данная таблица не будет видна в текущим сеансе, но обратиться к ней можно. Механизмом автоочистки (автоматизировать выполнение команд VACUUM и ANALYZE), не может получить доступ к временной таблицы, и поэтому не может очистить или проанализировать временные таблицы
UNLOGGED	Данный параметр используется если необходимо создать таблицу, при котором данные не проходят через журнал предзаписи (журнал предзаписи (WAL) - необходим для обеспечения целостности данных). Данная таблица будет работать быстрее, но не защищена от сбоев, так как в случае сбоев она будет автоматически проводить усечение.
IF NOT EXISTS	При применения данного условия, система не выдает ошибку, только уведомляет, если зависимости с таким же именем уже существуют.
имя_таблицы	В данном параметре прописывается наименование таблицы, которая должна быть создана.
имя_типа	При указание данного параметра создается типизированная таблица. Типизированная таблица, имя, которой может быть дополнено схемой, берет свою структуру из составного типа. Так же данная таблица будет привязана к своему типу, поэтому если тип будет удален, таблица так же будет удалена.
имя_столбца	В данном параметре прописывается наименование столбца, который должен быть создан в данной таблице.
тип_данных	Данный параметр указывает, каким типом данных будет включать себя данный столбец. Типы данных представлены в таблице А.2
COLLATE *тип_сортировки*	Данная строка указывает, какой тип сортировки будет в столбце. Если данный параметр не указан, то по умолчанию будет использоваться сортировка по умолчанию, установленного для типа данных столбца
LIKE *исх_табл* [*вариант_копир* ...]	Выражение LIKE указывает таблицу, из которой автоматически копирует все имена столбцов, типы данных и ограничения, не равные NULL. Новая и исходная таблица становятся автономными по отношению друг к другу.

Параметр/выражение	Описание
	Если необходимо скопировать выражения по умолчанию, которые были определены в копированных столбцов, необходимо указать параметр INCLUDING DEFAULTS, без указания данного параметра в новую таблицу будут скопированы столбцы со значением NULL. Если необходимо скопировать расширенную статистику в новую таблицу, необходимо указать параметр INCLUDING STATISTICS. Если необходимо скопировать индексы, с ограничениями PRIMARY KEY, UNIQUE и EXCLUDE, то необходимо указать значение INCLUDING INDEXES. Если необходимо скопировать настройки STORAGE, то необходимо указать значение INCLUDING STORAGE Если необходимо скопировать комментарии для столбцов, ограничений и индексов, то необходимо указать значение INCLUDING COMMENTS. INCLUDING ALL – это сокращенная форма: INCLUDING COMMENTS INCLUDING CONSTRAINTS INCLUDING DEFAULTS INCLUDING IDENTITY INCLUDING INDEXES INCLUDING STATISTICS INCLUDING STORAGE
INHERITS (*родит_табл* [, ...])	Выражение (необязательное) INHERITS указывает список таблиц, из которых новая таблица автоматически наследует все столбцы родительской таблице. INHERITS позволять создавать постоянную связь между новой и ее родительской таблицей. При модификации родительской таблицей, данная модификация распространяется на ее дочерние таблицы. При существовании имен столбцов с аналогичными типами данных из нескольких родительских таблиц, то выдается сообщение об ошибке, если данного конфликта нет, то можно дублированные столбцы объединить в один. Ограничения CHECK объединяются по существу так же, как столбцы: если несколько родительских таблиц и / или новое определение таблицы содержат ограничения CHECK с одинаковыми именами, все эти ограничения должны иметь одно и то же проверочное выражение, иначе будет сообщено об ошибке. Настройки столбца STORAGE также копируются из родительских таблиц. Если столбец в родительской таблице является столбцом идентификаторов, это свойство не наследуется. При желании столбец в дочерней таблице может быть объявлен как столбец идентификаторов
PARTITION BY { RANGE | LIST | HASH } ({ *имя_столбца* | (*выражение*) } [COLLATE *тип_сортировки*] [*по_классу*] [, ...])	Выражение (необязательно) PARTITION BY определяет стратегию разделения таблицы (секционированная таблица). В скобках данного выражение записывается список столбцов или выражений образующий ключ раздела для таблицы.
CONSTRAINT *имя_ограничения*	Выражение CONSTRAINT используется для ограничения таблиц и столбцов. Имя указывать необязательно, система сама его сгенерирует, но при указании имени, его следуют заключить в двойные кавычки. При нарушении ограничения, система выдаст сообщение об ошибке с указанием имени ограничения.
NOT NULL	Столбец не может содержать нулевые значения
NULL	Столбец может содержать нулевые значения.
CHECK (*выражение*) [NO INHERIT]	Выражение CHECK определяет производящее логический результат, которому должны соответствовать новые или обновленные строки, чтобы операция вставки или обновления выполнялась успешно. Операция выполняется успешно если выражение имеет значение: TRUE или UNKNOWN. Если какая-либо строка операции вставки или обновления выдает результат FALSE, возникает исключение ошибки, и вставка или обновление не осуществляются.

Параметр/выражение	Описание
	Ограничение, помеченное NO INHERIT, не будет распространяться на дочерние таблицы. Если таблица имеет несколько ограничений CHECK, они будут проверяться для каждой строки в алфавитном порядке по имени после проверки ограничений NOT NULL.
DEFAULT *выражение_по_умолчанию*	Данная строчка задает значение для столбца по умолчанию (без переменных). Выражение по умолчанию будет использоваться в любой операции вставки, в которой не указано значение для столбца. Если для столбца нет значения по умолчанию используется значение NULL.
GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [(*вариант_последовательности*)]	Данная строчка задает столбец как столбец идентификаторов. В столбец в новых строках будет автоматически иметь значения из назначенной ему последовательности. Предложения ALWAYS и BY DEFAULT определяют, как вариант последовательности и получают приоритет над заданным пользователем инструкции INSERT.
UNIQUE (*ограничение столбца*)	Ограничение UNIQUE указывает, что группа из одного или нескольких столбцов таблицы может содержать только уникальные значения. Добавление уникального ограничения автоматически создаст уникальный индекс btree для столбца или группы столбцов, используемых в ограничении.
PRIMARY KEY (*ограничение столбца*)	Ограничение PRIMARY KEY указывает, что столбец или столбцы таблицы могут содержать только уникальные (не повторяющиеся) ненулевые значения. Добавление ограничения PRIMARY KEY автоматически создаст уникальный индекс btree для столбца или группы столбцов, используемых в ограничении
REFERENCES *целевая_таблица* [(*целевая_столбец*)] [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE] FOREIGN KEY (*имя_столбца* [, ...]) REFERENCES *целев_таблица* [(*целев_столбец* [, ...])] [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE] [ON DELETE *действие*] [ON UPDATE *действие*] }	В данных выражениях указывается ограничение внешнего ключа UNIQUE, которое требует, чтобы группа из одного или нескольких столбцов новой таблицы содержала только те значения, которые соответствуют значениям в столбце, указанном в некоторой строке таблицы, на которую имеется ссылка. Если список ц*елев_столбец* опущена, используется первичный ключ PRIMARY KEY - *целев_таблица*. Пользователь должен иметь разрешение REFERENCES для ссылочной таблицы (либо для всей таблицы, либо для конкретных ссылочных столбцов). Добавление ограничения внешнего ключа требует блокировки для ссылочной таблицы. Ограничения внешнего ключа не могут быть определены между временными таблицами и постоянными таблицами.
DEFERRABLE | NOT DEFERRABLE	Данное выражение определяет, может ли ограничение быть отложено. По умолчанию используется значение NOT DEFERRABLE
INITIALLY DEFERRED | INITIALLY IMMEDIATE	Если ограничение является отложенным данное выражение определяет время для проверки ограничения. – IMMEDIATE (по умолчанию) проверяется после каждого оператора. – DEFERRED, проверяется только в конце транзакции.
WITH (*параметры_хранения* [= *значение*] [, ...]	Данное выражение является необязательным в нем представлены параметры хранения для индексов или таблиц. Параметры хранения представлены в таблице А.3
ON COMMIT	Контроль поведение временных таблиц в конце блока транзакции
PRESERVE ROWS	В конце транзакции нет ни каких специальных действий
DELETE ROWS	В конце транзакции будут удаляться строки из временной таблице

Параметр/выражение	Описание
DROP	В конце транзакции будет удалена временная таблица
TABLESPACE *наименование_табличного_пространства*]	Указывается имя табличного пространства где будет создана новая таблица

Типы данных представлены в таблице А.2.

Таблица А.2 – Типы данных

Наименование	Псевдоним	Описание
bigint	int8	8-байтное целое значение
bigserial	serial8	8-байтное целое значение с автоопределением
bit [(n)]		строка (бит) фиксированной длины
bit varying [(n)]	varbit [(n)]	строка (бит) переменной длины
boolean	bool	логическое значение (true/false)
box		прямоугольник в плоскости
bytea		двоичные данные («байтовый массив»)
character [(n)]	char [(n)]	символьная строка фиксированной длины
character varying [(n)]	varchar [(n)]	символьная строка переменной длины
cidr		сетевой адрес IPv4 или IPv6
circle		круг на плоскости
date		календарная дата (год, месяц, день)
double precision	float8	число с плавающей запятой двойной точности (8 байт)
inet		адрес хоста IPv4 или IPv6
integer	int, int4	четырехбайтовое целое со знаком
interval [fields] [(p)]		промежуток времени
json		текстовые данные JSON
jsonb		двоичные данные JSON, разложенные
line		бесконечная линия на плоскости
lseg		отрезок на плоскости
macaddr		MAC (Media Access Control) адрес
macaddr8		MAC (Media Access Control) адрес (EUI-64 формат)
money		сумма в валюте
numeric [(p, s)]	decimal [(p, s)]	точное число выбираемой точности
path		геометрический путь на плоскости
pg_lsn		порядковый номер журнала Jatoba
point		геометрическая точка на плоскости
polygon		замкнутый геометрический путь на плоскости

Наименование	Псевдоним	Описание
real	float4	число с плавающей точкой одинарной точности (4 байта)
smallint	int2	двухбайтовое целое со знаком
smallserial	serial2	автоинкрементное двухбайтовое целое число
serial	serial4	автоинкрементное четырехбайтовое целое число
text		символьная строка переменной длины
time [(p)] [without time zone]		время суток (без часового пояса)
time [(p)] with time zone	timetz	время суток с учётом часового пояса
timestamp [(p)] [without time zone]		дата и время (без часового пояса)
timestamp [(p)] with time zone	timestamptz	дата и время с учётом часового пояса
tsquery		запрос текстового поиска
tsvector		документ текстового поиска
txid_snapshot		снимок идентификатора транзакций
uuid		универсальный уникальный идентификатор
xml		XML-данные

Параметры хранения представлены в таблице А.3.

Таблица А.3 – Параметры хранения

Значение поля	Описание
fillfactor (integer)	Коэффициент заполнения таблицы составляет от 10 до 100 процентов (100 процентов по умолчанию). Если указать коэффициент заполнения таблицы меньше 100, то операция INSERT упаковывают страницы таблицы только до указанного процента, а остальное будет зарезервировано, это дает UPDATE возможность разместить обновленную копию строки на той же странице, что и оригинал, что более эффективно, чем размещение ее на другой странице. Этот параметр нельзя установить для таблиц TOAST.
parallel_workers (integer)	Устанавливается количество рабочих процессов при параллельном сканировании таблицы
autovacuum_vacuum_threshold, toast.autovacuum_vacuum_threshold (integer)	Минимальное количество обновленных или удаленных кортежей, необходимое для запуска VACUUM в любой одной таблице. По умолчанию 50 кортежей.
autovacuum_vacuum_scale_factor, toast.autovacuum_vacuum_scale_factor (float4)	Определяет часть размера таблицы, добавляемую к autovacuum_vacuum_threshold при принятии решения о запуске VACUUM. По умолчанию используется значение 0,2 (20% от размера таблицы).
autovacuum_analyze_threshold (integer)	Задает минимальное количество вставленных, обновленных или удаленных кортежей, необходимое для запуска ANALYZE в любой отдельной таблице. По умолчанию это 50 кортежей.
autovacuum_analyze_scale_factor (float4)	Определяет часть размера таблицы, добавляемую к autovacuum_analyze_threshold при принятии решения о необходимости запуска ANALYZE. По умолчанию используется значение 0,1 (10% от размера таблицы).

Значение поля	Описание
autovacuum_vacuum_cost_delay, toast.autovacuum_vacuum_cost_delay (integer)	Задает значение задержки стоимости, которое будет использоваться в автоматических операциях VACUUM. Значение по умолчанию составляет 20 миллисекунд
autovacuum_vacuum_cost_limit, toast.autovacuum_vacuum_cost_limit (integer)	Указывает предельное значение стоимости, которое будет использоваться в автоматических операциях VACUUM.
autovacuum_freeze_max_age, toast.autovacuum_freeze_max_age (integer)	Задает максимальный возраст (в транзакциях), прежде чем операция VACUUM будет вынуждена предотвратить обход идентификатора транзакции в таблице. VACUUM также позволяет удалять старые файлы, поэтому по умолчанию это относительно низкие 200 миллионов транзакций

Приложение Б
(обязательное)
Описание синтаксиса команды: «ALTER TABLE» и расшифровка его параметров

Синтаксис команды «ALTER TABLE» представлен ниже:

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]действие [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]    RENAME [ COLUMN ] имя_столбца TO новое_имя_столбца
ALTER TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]    RENAME CONSTRAINT имя_ограничения TO новое_имя_ограничения
ALTER TABLE [ IF EXISTS ] имя    RENAME TO новое_имя
ALTER TABLE [ IF EXISTS ] имя    SET SCHEMA новая_схема
ALTER TABLE ALL IN TABLESPACE имя [ OWNED BY имя_роли [, ... ] ]    SETTABLESPACEновое_табличного_пространства [ NOWAIT ]
ALTER TABLE [ IF EXISTS ] имя    ATTACH PARTITION имя_раздела { FOR VALUES спец_связ_разд | DEFAULT }
ALTER TABLE [ IF EXISTS ] имя    DETACH PARTITION имя_раздела
```

В данных синтаксисах:

а) где *действие*:

- ADD [COLUMN] [IFNOTEXISTS] *имя_столбцатип_данных* [COLLATE*тип_сортировки*] [*ограничение_столбца* [...]]
- DROP [COLUMN] [IFEXISTS] *имя_столбца* [RESTRICT | CASCADE]
- ALTER [COLUMN] *имя_столбца* [SETDATA] TYPE*тип_данных* [COLLATE*тип_сортировки*] [USING*выражение*]
- ALTER [COLUMN] *имя_столбца* SET DEFAULT *выражение*
- ALTER [COLUMN] *имя_столбца* DROP DEFAULT
- ALTER [COLUMN] *имя_столбца* { SET | DROP } NOT NULL
- ALTER [COLUMN] *имя_столбца* ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [(*вариант_последов*)]
- ALTER [COLUMN] *имя_столбца* { SET GENERATED { ALWAYS | BY DEFAULT } | SET *вариант_последов* | RESTART [[WITH] *перезапуск*] } [...]
- ALTER [COLUMN] *имя_столбца* DROP IDENTITY [IF EXISTS]
- ALTER [COLUMN] *имя_столбца* SET STATISTICS *integer*
- ALTER [COLUMN] *имя_столбца*SET (*настройка_атрибута = значение* [, ...])
- ALTER [COLUMN] *имя_столбца*RESET (*настройка_атрибута* [, ...])
- ALTER [COLUMN] *имя_столбца*SETSTORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
- ADD*ограничение_таблицы* [NOTVALID]
- ADD *ограничение_таблицы_исп_индекс*
- ALTER CONSTRAINT *имя_ограничения* [DEFERRABLE | NOT DEFERRABLE] [INITIALLY DEFERRED | INITIALLY IMMEDIATE]
- VALIDATE CONSTRAINT *имя_ограничения*

- DROP CONSTRAINT [IF EXISTS] *имя_ограничения* [RESTRICT | CASCADE]
- DISABLE TRIGGER [*имя_триггера* | ALL | USER]
- ENABLE TRIGGER [*имя_триггера* | ALL | USER]
- ENABLE REPLICA TRIGGER *имя_триггера*
- ENABLE ALWAYS TRIGGER *имя_триггера*
- DISABLERULE*перезап_имени_роли*
- ENABLERULE*перезап_имени_роли*
- ENABLE REPLICA RULE *перезап_имени_роли*
- ENABLE ALWAYS RULE *перезап_имени_роли*
- DISABLE ROW LEVEL SECURITY
- ENABLE ROW LEVEL SECURITY
- FORCE ROW LEVEL SECURITY
- NO FORCE ROW LEVEL SECURITY
- CLUSTER ON *имя_индекса*
- SET WITHOUT CLUSTER
- SET TABLESPACE *новое_табличного_пространства*
- SET { LOGGED | UNLOGGED }
- SET (*параметр_хранения* = *значение* [, ...])
- RESET (*параметр_хранения* [, ...])
- INHERIT*родит_табл*
- NOINHERIT*родит_табл*
- OF*имя_типа*
- NOT OF
- OWNER TO { *новый_владелец*| CURRENT_USER | SESSION_USER }
- REPLICA IDENTITY { DEFAULT | USING INDEX *имя_индекса* | FULL | NOTHING }

б) где *спец_связ_разд*:

- IN ({ *числ_литер* | *строк_литер* | TRUE | FALSE | NULL } [, ...]) |
- FROM ({ *числ_литер* | *строк_литер* | TRUE | FALSE | MINVALUE | MAXVALUE } [, ...])
- TO ({ *числ_литер* | *строк_литер* | TRUE | FALSE | MINVALUE | MAXVALUE } [, ...]) |
- WITH (MODULUS*числ_литер*, REMAINDER*числ_литер*)

в) где *ограничение_столбца*:

- [CONSTRAINT *имя_ограничения*]
- { NOT NULL |
- NULL |

- CHECK (*выражение*) [NOINHERIT] |
- DEFAULT*выражение_по_умолчанию*|
- GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [(*вариант_последовательности*)] |
- UNIQUE *параметры_индекса* |
- PRIMARY KEY *параметры_индекса* |
- REFERENCES *целев_таблица* [(*целев_столбец*)] [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]
- [ON DELETE *действие*] [ON UPDATE *действие*] }
- [DEFERRABLE | NOT DEFERRABLE] [INITIALLY DEFERRED | INITIALLY IMMEDIATE]

г) где *ограничение_таблицы*:

- [CONSTRAINT*имя_ограничения*]
- { CHECK (*выражение*) [NOINHERIT] |
- UNIQUE (*имя_столбца* [, ...]) *параметры_индекса* |
- PRIMARYKEY (*имя_столбца* [, ...]) *параметры_индекса* |
- EXCLUDE [USING*индексный_метод*] (*искл_элемент*WITH*оператор* [, ...]) *параметры_индекса* [WHERE (*предикат*)] |
- FOREIGNKEY (*имя_столбца* [, ...]) REFERENCES*целев_таблица* [(*целев_столбец* [, ...])]
- [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE] [ON DELETE *действие*] [ON UPDATE *действие*] }
- [DEFERRABLE | NOT DEFERRABLE] [INITIALLY DEFERRED | INITIALLY IMMEDIATE]

д) где *ограничение_таблицы_исп_индекс*:

- [CONSTRAINT *имя_ограничения*]
- { UNIQUE | PRIMARY KEY } USING INDEX *имя_индекса*
- [DEFERRABLE | NOT DEFERRABLE] [INITIALLY DEFERRED | INITIALLY IMMEDIATE]

е) где *параметры_индекса* в ограничении UNIQUE, PRIMARY KEY, and EXCLUDE:

- [INCLUDE (*имя_столбца* [, ...])]
- [WITH (*параметр_хранения* [= *значение*] [, ...])]
- [USINGINDEXTABLESPACE*наименование_табличного_пространства*]

ж) где *искл_элемент* в ограничении EXCLUDE:

- { *имя_столбца* | (*выражение*) } [*по_классу*] [ASC | DESC] [NULLS { FIRST | LAST }]

Расшифровка параметров команды «ALTER TABLE» представлена в таблице Б.1.

Таблица Б.1 – Расшифровка параметров команды «ALTER TABLE»

Параметр/выражение	**Описание**
ADD COLUMN	Данная форма используется для добавления нового столбца в таблицу. Слово COLUMN не является обязательным и может быть при выполнении команды опущено.
DROP COLUMN [IF EXISTS]	Данная форма используется для удаления столбца из таблицы. Все ограничение таблицы и индексов, включающий данный столбец будут удалены. Если есть у данного столбца есть зависимости вне данной таблице, например, ссылки на внешние ключи, то необходимо указать значение «CASCADE» Если столбца не существует и указан IF EXISTS, то система ошибки не выдаст, а на экране отобразится уведомление
SET DATA TYPE	Данная форма используется для изменения типы данных столбца. Все ограничение таблицы и индексов будут автоматически преобразованы под новый тип данных. Необязательный параметр COLLATE указывает параметры сортировки для нового столбца. Параметр USING (необязательный) определяет способ вычисления значение нового столбца из старого.
SET/DROP DEFAULT	Данные формы позволяет устанавливать и удалять значения для столбца по умолчанию. Значения по умолчанию применяются только к последующим командам INSERT, так как данные формы не изменяют значения строк.
SET/DROP NOT NULL	Данные формы изменяются независимо от того, помечен ли столбец, чтобы разрешить нулевые значения или отклонить нулевые значения. Использовать SET NOT NULL возможно в том случае, если столбец не содержит нулевое значение
ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY SET GENERATED { ALWAYS | BY DEFAULT } DROP IDENTITY [IF EXISTS]	Данные формы изменяют столбец идентификаторов или атрибут генерации существующего столбца идентификаторов. Если задано DROP IDENTITY IF EXISTS и столбец не является столбцом идентификаторов, ошибка не выдается, а на экране высветится только уведомление
SET *вариант_последов* RESTART	Данные формы изменяют последовательность, которая лежит в основе существующего столбца идентификаторов
SET STATISTICS	Данная форма позволяет устанавливает цель сбора цель сбора статистики по столбцам для последующих операций ANALYZE.
SET (*настройка_атрибута* = *значение* [, ...]) RESET (*настройка_атрибута* [, ...])	Данные формы позволяют устанавливать или сбрасывать параметры для каждого атрибута
SET STORAGE	Данная форма устанавливает режим хранения для столбца
ADD *ограничение_таблицы* [NOT VALID]	Данная форма добавляет новые ограничения для таблицы
ADD *ограничение_таблицы_исп_индекс*	Данная форма добавляет новое ограничение PRIMARY KEY или UNIQUE к таблице на основе существующего уникального индекса
ALTER CONSTRAINT	Данная форма изменяет атрибуты ограничения, которое было создано ранее.
VALIDATE CONSTRAINT	Данная форма проверяет внешний ключ или ограничение, которое было создано ранее как NOT VALID.
DROP CONSTRAINT [IF EXISTS]	Данная форма удаляет указанное ограничение для таблицы вместе с любым индексом, лежащим в основе ограничения. Если указано IF EXISTS и ограничение не существует, на экране отобразится только уведомление, а не ошибка.
DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER	Данные формы настраивают триггера в таблице.

Параметр/выражение	Описание
DISABLE/ENABLE [REPLICA | ALWAYS] RULE	Данные формы настраивают запуск правил перезаписи, принадлежащих таблице.
DISABLE/ENABLE ROW LEVEL SECURITY	Данные формы управляют применением политик безопасности строк, принадлежащих таблице.
CLUSTER ON	Данная форма выбирает индекс по умолчанию для кластеризации таблицы по индексу, но данная команда не кластеризирует саму таблицу. Изменение параметров кластера получает блокировку SHARE UPDATE EXCLUSIVE
SET WITHOUT CLUSTER	Данная форма удаляет из таблицы самую последнюю использованную спецификацию индекса CLUSTER. Это влияет на будущие кластерные операции, которые не указывают индекс. Изменение параметров кластера получает блокировку SHARE UPDATE EXCLUSIVE.
SET TABLESPACE	Данная форма изменяет табличное пространство таблицы на указанное табличное пространство и перемещает файлы данных, связанные с таблицей, в новое табличное пространство
SET { LOGGED | UNLOGGED }	Данная форма, которая не применима к временной таблице, изменяет таблицу с незарегистрированной на зарегистрированную или наоборот (см. UNLOGGED в таблице А.1)
SET (*параметр_хранения* = *значение* [, ...])	Данная форма изменяет один или несколько параметров хранения для таблицы
RESET (*параметр_хранения* [, ...])	Данная форма сбрасывает один или несколько параметров хранения к значениям по умолчанию.
INHERIT *родит_табл*	Данная форма добавляет целевую таблицу в качестве нового дочернего элемента указанной родительской таблицы. Впоследствии запросы к родителю будут включать записи целевой таблицы
NO INHERIT *родит_табл*	Данная форма удаляет целевую таблицу из списка дочерних элементов указанной родительской таблицы. Запросы к родительской таблице больше не будут включать записи, взятые из целевой таблицы.
OF *имя_типа*	Данная форма связывает таблицу с составным типом. Список имен и типов столбцов в таблице должен точно соответствовать составному типу
NOT OF	Данная форма отделяет типизированную таблицу от ее типа
OWNER TO	Данная форма изменяет владельца таблицы на указанного пользователя.
REPLICA IDENTITY	Данная форма изменяет информацию, которая записывается в журнал предварительной записи, чтобы идентифицировать строки, которые обновляются или удаляются.
RENAME	Данная форма изменяют имя таблицы), имя отдельного столбца в таблице или имя ограничения таблицы
SET SCHEMA	Данная форма перемещает таблицу в другую схему. Связанные индексы, ограничения и последовательности, принадлежащие столбцам таблицы, также перемещаются.
ATTACH PARTITION *имя_раздела* { FOR VALUES *спец_связ_разд* | DEFAULT }	Данная форма присоединяет существующую таблицу (которая может быть сама разбита) как раздел целевой таблицы
DETACH PARTITION *имя_раздела*	Данная форма отделяет указанный раздел целевой таблицы. Отсоединенный раздел продолжает существовать как отдельная таблица, но больше не имеет связей с таблицей, от которой он был отсоединен. Все индексы, которые были прикреплены к индексам целевой таблицы, отсоединяются.
IF EXISTS	При введение данного параметра от системы приходит не ошибка, а уведомление, если таблица не существует
имя	Наименование таблицы, которую нужно изменить
имя_столбца	В данном параметре прописывается наименование нового или существующего столбца
новое_имя_столбца	новое имя столбца

Параметр/выражение	Описание
новое_имя	новое имя
тип_данных	Тип данных нового столбца или новый тип данных для существующего столбца
ограничение_таблицы	Новое ограничение таблицы для таблицы
имя_ограничения	Имя нового или существующего ограничения.
CASCADE	Данный параметр автоматически отбрасывать объекты, которые зависят от удаленного столбца или ограничения
RESTRICT	Данный параметр позволяет отказаться от удаления столбца или ограничения, если есть какие-либо зависимые объекты
TRIGGER [*имя_триггера*| ALL | USER]	В данном выражение указывается имя триггера. Параметр ALL - отключить или включить все триггеры, принадлежащие таблице (роль пользователя - суперпользователь). Параметр USER - отключите или включите все триггеры, принадлежащие таблице, за исключением внутренних сгенерированных триггеров ограничений, таких как те, которые используются для реализации ограничений внешнего ключа или отложенных ограничений уникальности и исключений.
имя_индекса	Наименование существующего индекса
параметр_хранения	Наименование параметра хранения таблицы
значение	Новое значение для параметра хранения таблицы. Это может быть число или слово в зависимости от параметра.
родит_табл	Родительская таблица для связи или отмены связи с этой таблицей.
новый_владелец	Наименование нового владельца данной таблице
новое_табличного_пространства	Наименование табличного пространства, в которое будет перемещена таблица.
новая_схема	Наименование схемы, в которую будет перемещена таблица.
имя_раздела	Наименование таблицы, которую нужно присоединить как новый раздел или отсоединить от этой таблицы.
спец_связ_разд	Описание привязки раздела для нового раздела

Приложение В
(обязательное)
Описание синтаксиса команды: «SELECT» и расшифровка его параметров

Синтаксис команды «SELECT» представлен ниже:

```
[ WITH [ RECURSIVE ] запрос_with [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( выражение [, ...] ) ] ]    [ * | выражение [ [ AS ] вых_имя ] [, ...] ]
   [ FROMсооб_from [, ...] ]
   [ WHEREусловие ]
   [ GROUPBYэлемент_групп [, ...] ]
   [ HAVINGусловие [, ...] ]
   [ WINDOWимя_окнаAS ( определ_окна ) [, ...] ]
   [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] выборка]
   [ ORDER BY выражение [ ASC | DESC | USING оператор] [ NULLS { FIRST | LAST } ] [, ...] ]
   [ LIMIT { число | ALL } ]
   [ OFFSET старт [ ROW | ROWS ] ]
   [ FETCH { FIRST | NEXT } [ число ] { ROW | ROWS } ONLY ]
   [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF имя_таблицы [, ...] ] [ NOWAIT | SKIP LOCKED ] [...] ]
```

а) где *сооб_from*:

```
   [ ONLY ] имя_таблицы [ * ] [ [ AS ] псевд [ ( столбец_псевд [, ...] ) ] ]
   [ TABLESAMPLEметод_отбор ( аргумент [, ...] ) [ REPEATABLE ( источник ) ] ]
   [ LATERAL ] ( выборка ) [ AS ] псевд [ ( столбец_псевд [, ...] ) ]    имя_запрос_with [ [ AS ] псевд [ ( столбец_псевд [, ...] ) ] ]
   [ LATERAL ] имя_функции ( [ аргумент [, ...] ] )
[ WITH ORDINALITY ] [ [ AS ] псевд [ ( столбец_псевд [, ...] ) ] ]
   [ LATERAL ] имя_функции ( [ аргумент [, ...] ] ) [ AS ] псевд ( столбец_описание [, ...] )
   [ LATERAL ] имя_функции ( [ аргумент [, ...] ] ) AS ( столбец_описание [, ...] )
   [ LATERAL ] ROWS FROM( имя_функции ( [ аргумент[, ...] ] ) [ AS ( столбец_описание [, ...] ) ] [, ...] )
            [ WITH ORDINALITY ] [ [ AS ] псевд [ ( столбец_псевд [, ...] ) ] ]
сооб_from [ NATURAL ] тип_соединсооб_from [ ON соедин_услов | USING ( соедин_столбец [, ...] ) ]
```

б) где *элемент_групп*:

```
( )
выражение
   ( выражение [, ...] )
ROLLUP ( { выражение | ( выражение [, ...] ) } [, ...] )
CUBE ( { выражение | ( выражение [, ...] ) } [, ...] )
GROUPINGSETS ( элемент_групп [, ...] )

andзапрос_withis:

имя_запрос_with [ (имя_столбца [, ...] ) ] AS ( выборка | значение | вставка | обновление | удаление )

TABLE [ ONLY ] имя_таблицы [ * ]
```

Расшифровка параметров команды «SELECT» представлена в таблице В.1.

Таблица В.1 – Расшифровка параметров команды «SELECT»

Параметр/выражение	**Описание**
Опция WITH	
WITH	Данная опция позволяет указать один или несколько подзапросов, на которые по имени можно ссылаться в первичном запросе. Данный подзапрос действуют как временная таблица или представления на время отработки основного запроса. Каждый подзапрос может быть оператором: SELECT, TABLE, INSERT, UPDATE или DELETE. При написании оператора изменения данных (INSERT, UPDATE или DELETE) в WITH обычно включается опция RETURNING. Опция RETURNING выдает выходные параметры и по сути формирует временную таблицу, если RETURNING опущен, оператор все еще выполняется, но он не выдает выходных данных, поэтому первичный запрос не может ссылаться на него как на таблицу.
ОпцияFROM	
FROM	Опция FROM для SELECT указывает на одну или несколько исходных таблиц. Если указано несколько источников, результатом будет являться перекрестное соединение всех источников. Обычно добавляются квалификационные условия (через WHERE), чтобы ограничить возвращаемые строки небольшим подмножеством перекрестного соединения. опция FROM может содержать следующие параметры: − *имя_таблицы.* имя (возможно, дополненное схемой) существующей таблицы или представления. − *псевд.*заменяющие имя для элемента FROM
TABLESAMPLE *метод_отбор* (*аргумент* [, ...]) [REPEATABLE (*источник*)]	В данном предложение для указания извлечение подмножество строк к таблице указывается метод *метод_отбор*: − метод BERNOULLI сканирует всю таблицу и выбирает или игнорирует отдельные строки независимо, с указанной вероятностью.

Параметр/выражение	Описание
	− метод SYSTEM выполняет выборку на уровне блоков, причем каждый блок имеет заданную вероятность выбора, все строки в каждом выбранном блоке возвращаются. Каждый из методов принимает один *аргумент*, представляющий собой долю таблицы в выборке, выраженную в процентах от 0 до 100. REPEATABLE не обязательный пункт, указывает на начальный номер, который будет использоваться в методах для генерации случайных чисел
выборка	ОпцияFROM может содержать вложенный запрос SELECT. Результат вызова вложенного запроса SELECT при выполнении команды SELECT будет создание временной таблицы. Вложенный запрос SELECT должен содержатся в круглых скобках и для него должен быть предоставлен псевдоним.
имя_запрос_with	На запрос WITH ссылается запись его имени, как если бы имя запроса было именем таблицы
имя_функции	ОпцияFROM может содержать вызов функции. Результат вызова функции при выполнении команды SELECT будет создание временной таблицы. При применении пункта WITH ORDINALITY к выходным параметрам добавляется еще один столбец с нумерации строк.
тип_соедин	Параметр «*тип_соедин*» может содержать следующие варианты применения: − [INNER] JOIN − LEFT [OUTER] JOIN − RIGHT [OUTER] JOIN − FULL [OUTER] JOIN − CROSS JOIN Для типов соединения INNER и OUTER должно быть указано следующие условие соединения: − NATURAL, − ON *соедин_услов* − USING (*соедин_услов* [, ...]) Для CROSS JOIN ни одно из этих условий не допускается. Пункт JOIN объединяет два элемента FROM. Два элемента FROM представляют собой любой тип элемента FROM, но для простоты восприятие будут называться «таблицами». JOIN объединяет два элемента слева направо, но при другой последовательности, необходимо использовать круглые скобки. CROSS JOIN и INNER JOIN создают простое перекрёстное соединение, тот же результат, который можно получить, перечисляя две таблицы на верхнем уровне FROM, но с ограниченным условием соединения (если оно есть). CROSS JOIN эквивалентно INNER JOIN ON (TRUE), то есть по условию строки не удаляются. LEFT OUTER JOIN возвращает все строки в перекрестном соединении, плюс одну строку для каждой строки в левой таблице. RIGHT OUTER JOIN возвращает все строки в перекрестном соединении, плюс одну строку для каждой строки в правой таблице. FULL OUTER JOIN возвращает все соединенные строки, плюс одну строку для каждой не сопоставленной левой строки (расширенной с нулями справа), плюс одну строку для каждой не сопоставленной правой строки (расширенной с нулями слева).
ON *соедин_услов*	*соедин_услов* − данное выражение, результатом которого является значение типа boolean (аналогично пункту WHERE), определяет, какие строки в соединении считаются соответствующими.

Параметр/выражение	Описание
USING (*соедин_столбец* [, ...]	Предложение вида USING (a, b, ...) является сокращением для ON левая_таблица.a = правая_таблица.a AND левая_таблица.b = правая_таблица.b .... Кроме того, USING подразумевает, что только одна из каждой пары эквивалентных столбцы будут включены в выходные соединения, а не оба.
NATURAL	NATURAL - это сокращение для списка USING, в котором упоминаются все столбцы в двух таблицах с одинаковыми именами. Если нет общих имен столбцов, NATURAL эквивалентно ON TRUE.
LATERAL	Данный пункт позволяет вложенному SELECT ссылаться на столбцы элементов FROM, которые появляются перед ним в списке FROM. (Без слова LATERAL каждый вложенный SELECT оценивается независимо и поэтому не может перекрестно ссылаться на любой другой элемент FROM.)
Опция WHERE	
WHERE	Данная опция является необязательной и имеет следующую форму: WHERE*условие* где *условие* - любое выражение, которое оценивается как результат типа boolean. Любая строка, которая не удовлетворяет этому условию, будет исключена из вывода. Строка удовлетворяет условию, когда фактические значения заменяются на любые переменные из этой строки
Опция GROUP BY	
GROUP BY	Данная опция является необязательной и имеет следующую форму: GROUPBY*элемент_групп* [, ...] GROUP BY объединит в одну строку все выбранные строки, которые имеют одинаковые значения для сгруппированных выражений. Выражение, используемое внутри *элемент_групп*, может быть именем входного столбца или порядковым номером выходного столбца (элемент списка SELECT) или произвольным выражением, сформированным из значений входного столбца. В случае неоднозначности имя GROUP BY будет интерпретироваться как имя входного столбца, а не имя выходного столбца.
Опция HAVING	
HAVING	Данная опция является необязательной и имеет следующую форму: HAVING*условие* [, ...] HAVING исключает групповые строки, которые не удовлетворяют условию. В отличие от WHERE, которые фильтрует отдельные строки перед применением GROUP BY, HAVING фильтрует групповые строки, созданные с помощью GROUP BY
ОпцияWINDOW	
WINDOW	Даннаяопция является необязательной и имеет следующую форму: WINDOW*имя_окна*AS (*определ_окна*) [, ...] *имя_окна* - это имя, на которое можно ссылаться из предложений OVER или последующих определений окон. *определ_окна* имеет следующую форму: [*имя_существ_окна*] [PARTITIONBY*выражение* [, ...]] [ORDERBY*выражение* [ASC | DESC | USING*оператор*] [NULLS { FIRST | LAST }] [, ...]] [*предложж_рамка*]

Параметр/выражение	Описание
	При указании параметра *имя_существ_окна* необходимо учесть, что он должен ссылаться на более раннюю запись в списки WINDOWS. Новое окно копирует разделение из этой записи, а также предложение сортировки, в этом случае для нового окна нельзя указать PARTITION BY, а только ORDER BY и то при условии, что у него не было у копируемого окна. Элементы списка PARTITION BY интерпретируются почти так же, как элементы предложения GROUP BY, за исключением того, что они всегда являются простыми выражениями и никогда не являются именем или номером выходного столбца. Другое отличие состоит в том, что эти выражения могут содержать агрегатные вызовы функций, которые не допускаются в обычном пункт GROUP BY. Аналогично, элементы списка ORDER BY интерпретируются почти так же, как элементы пункта ORDER BY, за исключением того, что выражения всегда принимаются как простые выражения, а не как имя или номер выходного столбца. Параметр «*предложж_рамка*» является необязательным и определяет оконную раму для оконных функций, которые зависят от фрейма. Рамка окна - это набор связанных строк для каждой строки запроса (называемой текущей строкой). «*предложж_рам*» имеет следующую форму записи: { RANGE | ROWS | GROUPS } *начало_рамки* [*искл_рамки*] { RANGE | ROWS | GROUPS } BETWEEN*начало_рамки*AND*конец_рамки* [*искл_рамки*] Где: − *начало_рамки* и *конец_рамки*: UNBOUNDED PRECEDING смещ PRECEDING CURRENT ROW смещ FOLLOWING UNBOUNDED FOLLOWING − *искл_рамки*: EXCLUDE CURRENT ROW EXCLUDE GROUP EXCLUDE TIES EXCLUDE NO OTHERS Если *конец_рамки* опущен, то по умолчанию используется CURRENT ROW. Ограничения заключаются в том, что параметр *начало_рамки* не может использовать UNBOUNDED FOLLOWING, а параметр *конец_рамки* не может быть UNBOUNDED PRECEDING. Выбор *конец_рамки* не может быть выше в списки *начало_рамки*AND*конец_рамки.* Перед параметром *начало_рамки* синтаксис RANGE BETWEEN CURRENT ROW AND смещ PRECEDING не допускается. Опция *искл_рамки* позволяет делать исключение из рамки строку, которые окружают текущую строку. EXCLUDE GROUP исключает текущую строку и ее упорядочивающие одноранговые элементы из фрейма. EXCLUDE TIES исключает любые равноправные элементы текущей строки из фрейма, но не саму текущую строку. EXCLUDE NO OTHERS просто явно указывает поведение по умолчанию - не исключать текущую строку или ее одноранговые узлы.

Параметр/выражение	Описание
	Режим ROWS может привести к непредсказуемым результатам, если порядок ORDER BY не упорядочивает строки однозначно. Режимы RANGE и GROUPS предназначены для обеспечения того, чтобы строки, являющиеся одноранговыми элементами в порядке ORDER BY, обрабатывались одинаково, все строки данной группы равноправных узлов будут находиться во фрейме или исключены из него.
СписокSELECT	
SELECT	Список SELECT (между ключевыми словами SELECT и FROM) определяет выражения, которые формируют выходные строки оператора SELECT. Выражения ссылаются на столбцы, вычисленные из опции FROM. Как и в таблице, каждый выходной столбец SELECT имеет имя. В простом SELECT это имя просто используется для обозначения столбца для отображения, но когда SELECT является подзапросом запроса большего размера, имя воспринимается большим запросом как имя столбца виртуальной таблицы, созданной подпрограммой. -query. Чтобы указать имя, которое будет использоваться для выходного столбца, необходимо написать AS *вых_имя* после выражения столбца. Если не указывать имя столбца СУБД автоматически выберет имя. Если выражение столбца является простой ссылкой на столбец, то выбранное имя совпадает с именем этого столбца. Имя выходного столбца можно использовать для ссылки на значение столбца в предложениях ORDER BY и GROUP BY. Вместо выражения * в список вывода можно записать * как сокращение для всех столбцов выбранных строк. Кроме того, вы можете написать *имя_таблицы*. * Как сокращение для столбцов, поступающих только из этой таблицы. В этих случаях невозможно указать новые имена с AS; имена выходных столбцов будут такими же, как имена столбцов таблицы. Выражения в списке вывода должны быть вычислены до применения DISTINCT, ORDER BY или LIMIT. Это очевидно необходимо при использовании DISTINCT, так как в противном случае неясно, какие значения выделяются. Однако во многих случаях удобно, если выходные выражения вычисляются после ORDER BY и LIMIT; особенно если список вывода содержит какие-либо изменчивые или дорогостоящие функции. Выходные выражения, содержащие функции, возвращающие множество, эффективно оцениваются после сортировки и перед ограничением, так что LIMIT будет действовать, чтобы вырезать выход из функции возврата набора.
ОпцияDISTINCT	
DISTINCT	Если указано SELECT DISTINCT, все дублирующийся строки удаляются из результирующего набора (одна строка сохраняется из каждой группы дубликатов), а вSELECT ALL все строки сохраняются. Выражение SELECT [ALL | DISTINCT [ON (*выражение* [, ...])] сохраняет только первую строку каждого набора строк, где заданные выражения оцениваются как равные. Выражения DISTINCT ON интерпретируются с использованием тех же правил, что и для ORDER BY.
ОпцияUNION	
UNION	Опция UNION имеет следующую общую форму: *выбор_оператора*UNION [ALL | DISTINCT] выбор_оператора *выбор_оператора* − этолюбойоператор SELECT безпредложений ORDER BY, LIMIT, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE или FOR KEY SHARE. (ORDER BY и LIMIT могут быть присоединены к подвыражению, если оно заключено в круглые скобки. Без скобок эти пункты будут применяться для применения к результату UNION, а не к его правому входному выражению.)

Параметр/выражение	Описание
	Оператор UNION вычисляет объединение множеств строк, возвращаемых задействованными операторами SELECT. Строка находится в наборе объединений двух наборов результатов, если она появляется хотя бы в одном из наборов результатов. Два оператора SELECT, которые представляют прямые операнды UNION, должны создавать одинаковое количество столбцов, а соответствующие столбцы должны иметь совместимые типы данных. Результат UNION, если не указана опция ALL,не содержит повторяющихся строк. ALL предотвращает устранение дубликатов. DISTINCT может быть записан для явного указания поведения по умолчанию для удаления дублирующийся строк. Несколько операторов UNION в одном и том же операторе SELECT вычисляются слева направо, если в скобках не указано иное
ОпцияINTERSECT	
INTERSECT	ОпцияINTERSECT имеет следующую общую форму: *выбор_оператора*INTERSECT [ALL | DISTINCT] *выбор_оператора* *выбор_оператора* − этолюбойоператор SELECT безпредложений ORDER BY, LIMIT, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE или FOR KEY SHARE. Оператор INTERSECT вычисляет пересечение множеств строк, возвращаемых задействованными операторами SELECT. Строка находится на пересечении двух наборов результатов, если она появляется в обоих наборах результатов. Результат INTERSECT не содержит повторяющихся строк, если не указана опция ALL. DISTINCT может быть написан для явного указания поведения по умолчанию для устранения дублирующийся строк.
Опция ORDER BY	
ORDER BY	Опция ORDER BY имеет следующую общую форму: ORDERBY выражение [ASC | DESC | USING*оператор*] [NULLS { FIRST | LAST }] [, ...] Опция ORDER BY приводит к сортировке результирующих строк в соответствии с указанными выражениями. Если две строки равны в соответствии с крайним левым выражением, они сравниваются в соответствии со следующим выражением и так далее. Если они равны по всем указанным выражениям, они возвращаются в порядке, зависящем от реализации. Каждое выражение может быть именем или порядковым номером выходного столбца (элемент списка SELECT) или может быть произвольным выражением, сформированным из значений входного столбца. Порядковый номер относится к порядковому (слева направо) положению выходного столбца. Эта функция позволяет определять порядок на основе столбца, который не имеет уникального имени. Также можно использовать произвольные выражения в предложении ORDER BY, включая столбцы, которые не отображаются в списке вывода SELECT. Таким образом, следующее утверждение верно: SELECTnameFROMdistributorsORDERBYcode; Ограничением этой функции является то, что предложение ORDER BY, применяемое к результату предложения UNION, INTERSECT или EXCEPT, может указывать только имя или номер выходного столбца, а не выражение. ORDER BY - это простое имя, которое соответствует как имени выходного столбца, так и имени входного столбца, ORDER BY будет интерпретировать его как имя выходного столбца. Это противоположно выбору, который GROUP BY сделает в той же ситуации. При желании можно добавить ключевое слово ASC (по возрастанию) или DESC (по убыванию) после любого выражения в предложении ORDER BY. Если не указано, ASC принимается по умолчанию. В качестве альтернативы, в предложении USING можно указать конкретное имя оператора. Если указано NULLS LAST, нулевые значения сортируются после всех ненулевых значений; если указано NULLS FIRST, нулевые значения сортируются перед всеми ненулевыми значениями.

Параметр/выражение	Описание
Пункт LIMIT	
LIMIT	Предложение LIMIT состоит из двух независимых подпунктов: LIMIT { *число* | ALL } OFFSET*начало* − *число* − определяет максимальное количество строк, которые нужно вернуть. − *начало* − указывает количество строк, которые нужно пропустить, прежде чем начинать возвращать строки. Когда оба заданы, начальные строки пропускаются перед началом подсчета количества возвращаемых строк. Если выражение *число* оценивается как NULL, оно обрабатывается как LIMIT ALL, то есть без ограничений. Если *начало* имеет значение NULL, он обрабатывается так же, как OFFSET

Лист регистрации изменений									
Изм.	Номера листов (страниц)				Всего листов (страниц) в докум.	№ докум.	Входящий № сопроводительного докум. и дата	Подп.	Дата
	измененных	замененных	новых	аннулированных					

Изм.	Подп.	Дата