

УТВЕРЖДЕН

643.72410666.00067-07 96 01-ЛУ

**ЗАЩИЩЕННАЯ СИСТЕМА УПРАВЛЕНИЯ  
БАЗАМИ ДАННЫХ «ЈАТОВА»**

**Руководство пользователя**

**643.72410666.00067-07 96 01**

Листов 153

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

## АННОТАЦИЯ

Администратор СУБД «Jatoba» должен иметь навыки по работе с системами управления базами данных (СУБД) PostgreSQL или защищенной СУБД «Jatoba» (ООО «Газинформсервис»).

Степени важности примечаний, применяемые в документе:



**Важная информация** – указания, требующие особого внимания



**Дополнительная информация** – указания, позволяющие упростить работу с изделием

Все примеры в данном документе приведены для СУБД «Jatoba» версии 4.x, для других версий все шаги выполняются аналогично, разница состоит в именах директорий.

Например, СУБД «Jatoba» версии 5.x по умолчанию устанавливается в директорию:

- ОС Windows – «C:\Program Files\GIS\Jatoba\5\bin»;
- ОС Linux – «/usr/jatoba-5/bin».



**Важная информация**

Для сертифицированной версии СУБД «Jatoba» поддерживается работа только на ОС, указанных в формуляре на поставку!

## СОДЕРЖАНИЕ

1. Общие сведения о СУБД «Jatoba».....	7
1.1. Назначение СУБД «Jatoba».....	7
1.2. Функции СУБД «Jatoba».....	8
1.3. Требования к среде функционирования СУБД «Jatoba».....	8
2. Порядок подключения к СУБД «Jatoba».....	11
2.1. Порядок подключения к СУБД «Jatoba» под управлением ОС Windows Server .....	11
2.2. Порядок подключения к СУБД «Jatoba» под управлением ОС GNU/Linux .....	12
3. Описание операций СУБД «Jatoba» .....	14
4. Синтаксис SQL.....	15
4.1. Лексическая структура.....	15
4.1.1. Идентификаторы и ключевые слова .....	16
4.1.2. Константы.....	18
4.1.3. Операторы.....	23
4.1.4. Специальные знаки.....	23
4.1.5. Комментарии .....	24
4.1.6. Приоритеты операторов .....	25
4.2. Выражения значения.....	26
4.2.1. Ссылки на столбцы.....	27
4.2.2. Позиционные параметры.....	27
4.2.3. Индексы элементов .....	27
4.2.4. Выбор поля .....	28
4.2.5. Применение оператора .....	28
4.2.6. Вызовы функций .....	29
4.2.7. Агрегатные выражения .....	29
4.2.8. Вызовы оконных функций .....	33
4.2.9. Приведения типов.....	37
4.2.10. Применение правил сортировки .....	38
4.2.11. Скалярные подзапросы.....	39
4.2.12. Конструкторы массивов .....	40
4.2.13. Конструкторы табличных строк.....	42
4.2.14. Правила вычисления выражений .....	44
4.3. Вызов функций.....	46
4.3.1. Позиционная передача .....	47
4.3.2. Именная передача.....	47
4.3.3. Смешанная передача .....	48
5. Типы данных.....	49
5.1. Числовые типы .....	51

5.1.1. Целочисленные типы.....	51
5.1.2. Числа с произвольной точностью .....	52
5.1.3. Типы с плавающей точкой .....	54
5.1.4. Последовательные типы.....	56
5.2. Денежные типы .....	58
5.3. Символьные типы .....	59
5.4. Двоичные типы данных .....	61
5.4.1. Шестнадцатеричный формат bytea .....	62
5.4.2. Формат спецпоследовательностей bytea.....	62
5.5. Типы даты/времени.....	64
5.5.1. Ввод даты/времени .....	66
5.5.2. Вывод даты/времени .....	71
5.5.3. Часовые пояса.....	72
5.5.4. Ввод интервалов.....	74
5.5.5. Вывод интервалов .....	77
5.6. Логический тип .....	78
5.7. Типы перечислений.....	79
5.7.1. Объявление перечислений .....	79
5.7.2. Порядок.....	80
5.7.3. Безопасность типа .....	81
5.7.4. Тонкости реализации.....	82
5.8. Геометрические типы.....	82
5.8.1. Точки.....	83
5.8.2. Прямые.....	83
5.8.3. Отрезки .....	83
5.8.4. Прямоугольники.....	84
5.8.5. Пути .....	84
5.8.6. Многоугольники.....	85
5.8.7. Окружности .....	85
5.9. Типы, описывающие сетевые адреса.....	85
5.9.1. inet.....	86
5.9.2. cidr.....	86
5.9.3. Различия inet и cidr .....	87
5.9.4. macaddr.....	87
5.9.5. macaddr8.....	88
5.10. Битовые строки.....	89
5.11. Типы, предназначенные для текстового поиска.....	90
5.11.1. tsvector.....	90
5.11.2. tsquery.....	92

5.12. Тип UUID.....	93
5.13. Тип XML.....	94
5.13.1. Создание XML-значений.....	94
5.13.2. Обработка кодировки .....	96
5.13.3. Обращение к XML-значениям .....	97
5.14. Типы JSON .....	97
5.14.1. Проектирование документов JSON.....	100
5.14.2. Проверки на вхождение и существование jsonb.....	100
5.14.3. Индексация jsonb .....	103
5.14.4. Обращение по индексу к элементам jsonb.....	107
5.14.5. Трансформации.....	110
5.14.6. Тип jsonpath .....	110
5.15. Массивы.....	113
5.15.1. Объявления типов массивов.....	113
5.15.2. Ввод значения массива.....	114
5.15.3. Обращение к массивам.....	116
5.15.4. Изменение массивов .....	119
5.15.5. Поиск значений в массивах.....	123
5.15.6. Синтаксис вводимых и выводимых значений массива .....	125
5.16. Составные типы.....	126
5.16.1. Объявление составных типов.....	127
5.16.2. Конструирование составных значений .....	128
5.16.3. Обращение к составным типам.....	129
5.16.4. Изменение составных типов .....	130
5.16.5. Использование составных типов в запросах.....	131
5.16.6. Синтаксис вводимых и выводимых значений составного типа .....	135
5.17. Диапазонные типы .....	136
5.17.1. Встроенные диапазонные и мультидиапазонные типы.....	137
5.17.2. Включение и исключение границ .....	137
5.17.3. Неограниченные (бесконечные) диапазоны .....	137
5.17.4. Ввод/вывод диапазонов.....	138
5.17.5. Конструирование диапазонов и мультидиапазонов .....	139
5.17.6. Типы дискретных диапазонов.....	140
5.17.7. Определение новых диапазонных типов .....	141
5.17.8. Индексация .....	143
5.17.9. Ограничения для диапазонов .....	144
5.18. Типы доменов.....	144
5.19. Идентификаторы объектов .....	145
5.20. Тип pg_lsn.....	149

5.21. Псевдотипы .....	150
Перечень сокращений.....	152

## 1. ОБЩИЕ СВЕДЕНИЯ О СУБД «JAТОВА»

### 1.1. Назначение СУБД «Jatoba»

СУБД «Jatoba» является программным средством, предназначенным для создания и управления реляционными базами данных на базе ЭВМ под управлением ОС, представленных в таблице 1.1.

Таблица 1.1 – Перечень поддерживаемых ОС

№	Наименование ОС	Серверная часть	Клиентская часть	Docker (ver.)	Сертификат ФСТЭК	
					№ серт.	Дата выдачи
1	Windows 10	X	X	—	—	—
2	Windows 11	X	X	—	—	—
3	Windows Server 2016	X	X	—	—	—
4	Windows Server 2019	X	X	—	—	—
5	Windows Server 2022	X	X	—	—	—
6	Astra Linux 1.7 Special Edition Смоленск (x86-64)	X	X	20.10.2	2557	30.01.2012
7	Astra Linux 1.8 (x86-64)	X	X	—	—	—
8	Astra Linux 2.12 Common Edition Орел (x86-64)	X	X	—	—	—
9	Debian 10	X	X	24.0.2	—	—
10	Debian 11	X	X	24.0.2	—	—
11	Debian 12	X	X	24.0.2	—	—
12	АЛТ 8 СП	X	X	20.10.11	3866	10.08.2018
13	АЛТ 10 СП	X	X	20.10.11	3866	10.08.2018
14	АЛТ 9.1 Server	X	X	—	—	—
15	АЛТ 10 Server	X	X	23.0.1	—	—
16	Ubuntu 20.04	X	X	24.0.2	—	—
17	Ubuntu 22.04	X	X	24.0.2	—	—
18	Ubuntu 24.04	X	X	24.0.2	—	—
19	ОСНОВА2	X	X	25.05	4381	31.03.2021
20	РЕД ОС 7.3 Муром	X	X	20.10.1	4060	12.01.2019
21	РЕД ОС 8	X	X	—	—	—
22	РОСА 7.9	X	X	—	—	—
23	РОСА 12.4	X	X	—	—	—
24	RedHat Enterprise Linux 8	X	X	—	—	—
25	Oracle Linux 8.4	X	X	—	—	—

## 1.2. Функции СУБД «Jatoba»

СУБД «Jatoba» реализует следующие функциональные возможности:

- управление данными во внешней памяти;
- управление данными в оперативной памяти;
- выполнение запросов (DDL/DML);
- управление транзакциями;
- журнализация изменений, резервное копирование и восстановление базы данных после сбоев, репликация.

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

- хранение пространственных, географических и геометрических данных, поддержка запросов к ним и управление ими;
- синтаксическая совместимость с распространенными PL/SQL Oracle;
- расширенные возможности секционирования больших таблиц;
- протоколирование, анализ и контроль выполнения команд манипулирования данными (DDL/DML);
- сбор журналов аудита всех операций и загрузка конфигураций в СУБД;
- работа в составе отказоустойчивого кластера с механизмом переключения нагрузки на основной узел кластера;
- защита от несанкционированного изменения конфигурационных файлов;
- 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 и удовлетворяющими следующим аппаратным требованиям, указанным в таблице 1.2.

Таблица 1.2 – Программные и аппаратные требования к ЭВМ, на которых функционируют клиентская и серверная часть СУБД

Параметр	Характеристика	Сертифицированная ОС
<b>Требования к аппаратному обеспечению сервера СУБД</b>		
ОЗУ	Не менее 2 Гб	
Свободный объем жесткого диска	Минимальный объем от 40 Гб Рекомендуемый объем от 100 Гб	

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------



Параметр	Характеристика	Сертифицированная ОС
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	
Максимальное количество ядер	256	
Устройства ввода-вывода	Стандартные 105-клавишная клавиатура и манипулятор «мышь» с USB, либо PS/2-интерфейсами	
Адаптер Ethernet	100 Мбит/с	
<b>Требования к аппаратному обеспечению АРМ управления</b>		
ОЗУ	Не менее 4 Гб	
Свободный объем жесткого диска	От 3 Гб	
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, либо PS/2 интерфейсами	
Адаптер Ethernet	100 Мбит/с	
<b>Требования к программному обеспечению сервера</b>		
Операционная система	Требования приведены в таблице 1.1	
<b>Требования к программному обеспечению АРМ управления</b>		
Операционная система	Требования приведены в таблице 1.1	
<b>Требования к аппаратному обеспечению сервера Jatoba data safe</b>		
ОЗУ	Не менее 2 Гб	
Свободный объем жесткого диска	Минимальный объем от 40 Гб Рекомендуемый объем от 100 Гб	
Устройства видео вывода	Монитор и видеоадаптер с поддержкой VGA и разрешением 800x600 или выше	

Параметр	Характеристика	Сертифицированная ОС
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, либо PS/2 интерфейсами	
Адаптер Ethernet	100 Мбит/с	
<b>Требования к программному обеспечению сервера Jatoba data safe</b>		
Поддерживаемые платформы	• win-x86;	—
	• win-x64;	—
	• linux-x64	X
СУБД	Защищенная система управления базами данных «Jatoba»	
Веб-сервер	IIS 10	—
	Nginx	X
Компоненты	ASP.NET Core 6.0 Runtime (v6.0.1) – Windows Hosting Bundle Installer	—
Internet браузер	• Google Chrome;	X
	• Яндекс.Браузер;	X
	• Chromium;	X
	• Mozilla Firefox;	X
	• Opera;	X
	• Microsoft Edge	—

## 2. ПОРЯДОК ПОДКЛЮЧЕНИЯ К СУБД «ЯТОВА»

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

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

а) запустить командную строку «cmd»;

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

```
cd "C:\Program Files\GIS\Jatoba\4\bin\"
```



Рисунок 2.1 – Переход в каталог

в) ввести команду для подключения к СУБД:

```
psql -U <учетная запись пользователя в СУБД> -d <наименование БД, к которой нужно произвести подключение>
```

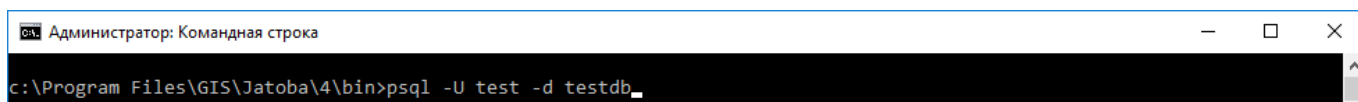


Рисунок 2.2 – Указание адреса подключения, пользователя и БД

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

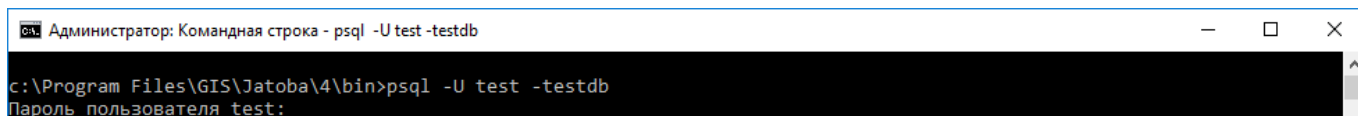


Рисунок 2.3 – Ввод пароля

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

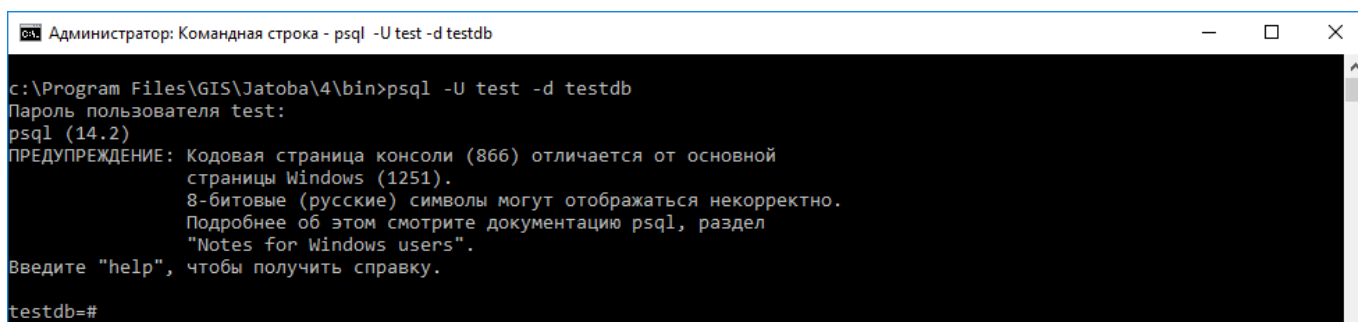


Рисунок 2.4 – Подключение к базе данных

## 2.2. Порядок подключения к СУБД «Jatoba» под управлением ОС GNU/Linux

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

а) запустить «Терминал»;

б) выполнить вход в систему под пользователем СУБД (в ходе установки СУБД по умолчанию создается пользователь «postgres»):

```
su -l postgres
```

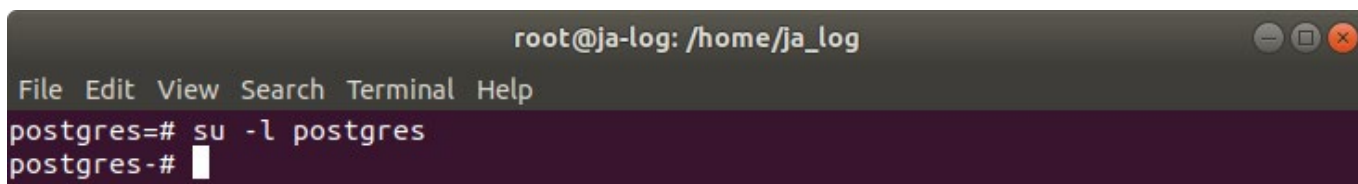


Рисунок 2.5 – Вход в СУБД

в) перейти в рабочий каталог при помощи команды:

```
cd /usr/jatoba-4/bin
```

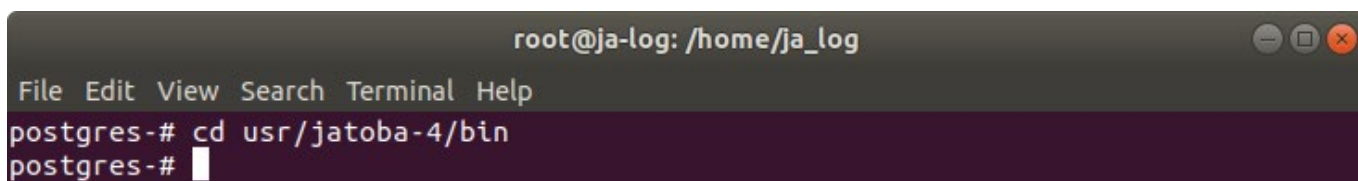
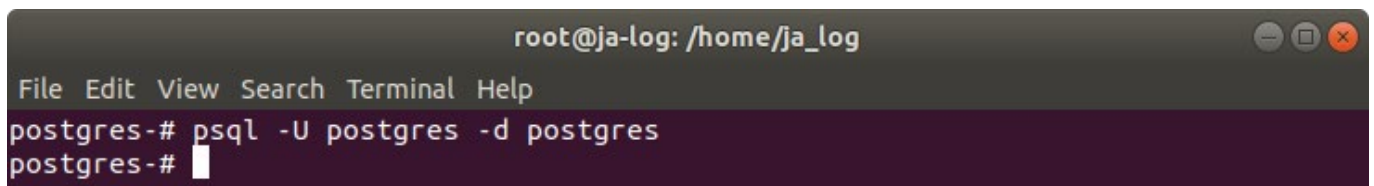


Рисунок 2.6 – Переход в каталог

г) выполнить команду подключения к СУБД «Jatoba»:

```
psql -U <учетная запись пользователя в СУБД> -d <наименование  
БД, к которой нужно произвести подключение>
```



The image shows a terminal window with a dark background. The title bar at the top reads "root@ja-log: /home/ja\_log". Below the title bar is a menu bar with the options "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal content shows a prompt "postgres-#" followed by the command "psql -U postgres -d postgres". The command has been executed, and the prompt has changed to "postgres-#" with a cursor. The terminal window has standard window control buttons (minimize, maximize, close) in the top right corner.

```
root@ja-log: /home/ja_log
File Edit View Search Terminal Help
postgres-# psql -U postgres -d postgres
postgres-#
```

Рисунок 2.7 – Подключение к БД

### **3. ОПИСАНИЕ ОПЕРАЦИЙ СУБД «ЈАТОВА»**

Пользователь работает с СУБД «Јatoba» 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.

## 4. СИНТАКСИС SQL

В подразделе приводится синтаксис языка запросов SQL, определена структура для хранения данных, способы наполнения базы данных и обращение к ней с запросами. Перечислены доступные типы данных, функции и операторы, которые могут быть использованы в командах SQL.

### 4.1. Лексическая структура

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

Компонентом команды может быть:

- ключевое слово;
- идентификатор;
- идентификатор в кавычках;
- строка (или константа);
- специальный символ.

Компоненты обычно разделяются пробельными символами (пробел, табуляция, перевод строки), но это не требуется, если нет неоднозначности (когда спецсимвол стоит рядом с компонентом другого типа).


Следующий текст является (синтаксически) правильной SQL-программой:

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

где «MY\_TABLE» – наименование таблицы.

Это последовательность трех команд, по одной в строке (можно размещать в одну строку или разделить команды на несколько строк).

Первые несколько компонентов обычно содержат имя команды, в данном примере это «SELECT», «UPDATE» и «INSERT».

-  Команде «UPDATE» необходимо, чтобы компонент «SET» всегда стоял в определенном положении, а «INSERT» требует наличие компонента «VALUES».


#### 4.1.1. Идентификаторы и ключевые слова

Компоненты SELECT, UPDATE и VALUES являются примерами ключевых слов – слов, имеющих фиксированное значение в языке SQL.

Компоненты MY\_TABLE и A – примеры идентификаторов, которые идентифицируют имена таблиц, столбцов или других объектов БД, в зависимости от того, где они используются. Иногда их называют просто «именами».

Ключевые слова и идентификаторы имеют одинаковую лексическую структуру.

Идентификаторы и ключевые слова SQL должны начинаться с буквы (a-z, хотя допускаются также не латинские буквы и буквы с диакритическими знаками) или подчеркивания (\_). Последующие символы в идентификаторе или ключевом слове – буквы, цифры (0-9), знаки доллара (\$) или подчеркивания.

-  Знаки доллара нельзя использовать в идентификаторах, так как их использование вредит переносимости приложений.

В SQL не может быть ключевых слов с цифрами, которые начинаются или заканчиваются подчеркиванием, поэтому идентификаторы такого вида защищены от возможных конфликтов с будущими расширениями стандарта.

Система выделяет для идентификатора не более NAMEDATALEN-1 байт, а более длинные имена сокращаются.

По умолчанию NAMEDATALEN = 64, поэтому максимальная длина идентификатора равна 63 байтам.

При необходимости этот предел можно увеличить, изменив константу NAMEDATALEN в файле src/include/pg\_config\_manual.h.



Идентификаторы делятся на два типа:

- без кавычек – воспринимаются системой без учета регистра. Запись команды `UPDATE MY_TABLE SET A = 5;` равносильна записи `uPDaTE my_TabLE SeT a = 5;`



Используется неформальное соглашение записывать ключевые слова заглавными буквами, а имена строчными:

`UPDATE my_table SET a = 5`

- в кавычках (отделенные идентификаторы) – образуются при заключении набора символов в двойные кавычки (").



Идентификаторы в кавычках не являются ключевыми словами.



«select» можно использовать для обозначения столбца или таблицы «select», но select без кавычек будет воспринято системой как ключевое слово, что приведет к ошибке.

Пример с идентификаторами в кавычках будет выглядеть следующим образом:

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

Идентификаторы в кавычках могут содержать любые символы, за исключением символа с кодом 0.

Также идентификаторы в кавычках позволяют использовать символы Unicode по их кодам. Такой идентификатор начинается с U& (строчная или заглавная U и амперсанд), а затем сразу без пробелов идет двойная кавычка, пример `U&"foo"`.



При этом возникает неоднозначность с оператором &. Для того, чтобы ее избежать, необходимо окружить этот оператор пробелами.

В кавычках можно записывать символы Unicode двумя способами:

- 1) обратная косая черта, а за ней код символа из четырех шестнадцатеричных цифр,
- 2) обратная косая черта, знак плюс, а затем код из шести шестнадцатеричных цифр.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

#### 4.1.2. Константы

В СУБД «Jatoba» есть три типа констант подразумеваемых типов: строки, битовые строки и числа. Константы можно записывать, указывая типы явно, что позволяет представить их более точно и обработать более эффективно.

##### 4.1.2.1 Строковые константы

Строковая константа в SQL – это обычная последовательность символов, заключенная в апострофы ('), пример: 'Это строка'. Для того, чтобы включить апостроф в строку, необходимо написать в ней два апострофа рядом, например: 'Это "строка'.



Это не то же самое, что двойная кавычка (").

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

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

эквивалентно:

```
SELECT 'foobar';
```

##### 4.1.2.1.1 Строковые константы со спецпоследовательностями в стиле C

СУБД «Jatoba» также принимает «спецпоследовательности», что является расширением стандарта SQL. Строка со спецпоследовательностями начинается с буквы E (заглавной или строчной), стоящей непосредственно перед апострофом, например, E'foo'.



Если константа со спецпоследовательностью разбивается на несколько строк, букву E нужно поставить только перед первым открывающим апострофом.

Внутри таких строк символ обратной косой черты (\) начинает C-подобные спецпоследовательности, в которых сочетание обратной косой черты со следующим символом(ами) дает определенное байтовое значение, как показано в таблице 4.1.

Таблица 4.1 – Спецпоследовательности

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

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



При параметре конфигурации `standard_conforming_strings = off`, PostgreSQL распознает обратную косую черту как спецсимвол и в обычных строках, и в строках со спецпоследовательностями.

При `standard_conforming_strings = on`, обратная косая черта распознается только в спецстроках. Это поведение может нарушить работу приложений, рассчитанных на предыдущий режим, когда обратная косая черта распознавалась везде. Если необходимо, чтобы обратная косая черта представляла специальный символ, необходимо задать ее строковую константу с E.


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

Строковая константа не может включать символ с кодом 0.

#### 4.1.2.1.2 Строковые константы со спецпоследовательностями Unicode

PostgreSQL также поддерживает еще один вариант спецпоследовательностей, позволяющий включать в строки символы Unicode по их кодам. Строковая константа со

спецпоследовательностями Unicode начинается с U& (строчная или заглавная U и амперсанд), а затем сразу без пробелов идет апостроф, например, U&'foo'.


-  Во избежание неоднозначности с оператором &, необходимо окружить этот оператор пробелами.

Затем в апострофах можно записывать символы Unicode двумя способами:

- 1) обратная косая черта, а за ней код символа из четырех шестнадцатеричных цифр,
- 2) обратная косая черта, знак плюс, а затем код из шести шестнадцатеричных цифр.

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

Чтобы включить спецсимвол в строку буквально, необходимо написать его дважды.

-  Спецпоследовательности Unicode в строковых константах работают только когда параметр конфигурации `standard_conforming_strings = on`.

При `standard_conforming_strings = off` спецпоследовательности будут вызывать ошибку.

#### 4.1.2.1.3 Строковые константы, заключенные в доллары

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

Внутри такой строки апострофы не нужно записывать особым образом. В строке, заключенной в доллары, все символы можно записывать в чистом виде: содержимое строки всегда записывается буквально. Ни обратная косая черта, ни даже знак доллара не являются

спецсимволами, если только они не образуют последовательность, соответствующую открывающему тегу.

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

Тег строки в долларах, если он присутствует, должен соответствовать правилам, определенным для идентификаторов без кавычек, и к тому же не должен содержать знак доллара.

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

Заключение строк в доллары не является частью стандарта SQL, но часто это более удобный способ записывать сложные строки, чем стандартный вариант с апострофами. Он особенно полезен, когда нужно представить строковую константу внутри другой строки, что часто требуется в определениях процедурных функций.

#### 4.1.2.2 Битовые строковые константы

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

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

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

#### 4.1.2.3 Числовые константы

Числовые константы могут быть заданы в следующем общем виде:

цифры

цифры. [цифры] [е [+ -] цифры]

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

```
[цифры] . цифры [е [+ -] цифры]
цифры е [+ -] цифры
```

где **цифры** — это одна или несколько десятичных цифр (0..9). До или после десятичной точки (при ее наличии) должна быть минимум одна цифра. Как минимум одна цифра должна следовать за обозначением экспоненты (е), если она присутствует. В числовой константе не может быть пробелов или других символов.

Любой знак минус или плюс в начале строки не считается частью числа; это оператор, примененный к константе.

Несколько примеров допустимых числовых констант:

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

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

#### 4.1.2.4 Константы других типов

Константу обычного типа можно ввести одним из следующих способов:

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

Текст строковой константы передается процедуре преобразования ввода для типа, обозначенного здесь `type`. Результатом становится константа указанного типа. Явное

приведение типа можно опустить, если нужный тип константы определяется однозначно (когда она присваивается непосредственно столбцу таблицы), так как в этом случае приведение происходит автоматически.

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

### 4.1.3. Операторы

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

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

Для имен операторов есть еще несколько ограничений:

- Сочетания символов – и /\* не могут присутствовать в имени оператора, так как они будут обозначать начало комментария.
- Многосимвольное имя оператора не может заканчиваться знаком + или -, если только оно не содержит также один из этих символов:

~ ! @ # % ^ & | ` ?

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

Записывая нестандартные SQL-операторы, обычно нужно отделять имена соседних операторов пробелами для однозначности.

### 4.1.4. Специальные знаки

Некоторые не алфавитно-цифровые символы имеют специальное значение, но при этом не являются операторами.

- Знак доллара (\$), предваряющий число, используется для представления позиционного параметра в теле определения функции или подготовленного оператора. В других контекстах знак доллара может быть частью идентификатора или строковой константы, заключенной в доллары.

- Круглые скобки (()) имеют обычное значение и применяются для группировки выражений и повышения приоритета операций. В некоторых случаях скобки – это необходимая часть синтаксиса определенных SQL-команд.
- Квадратные скобки ([]) применяются для выделения элементов массива.
- Запятые (,) используются в некоторых синтаксических конструкциях для разделения элементов списка.
- Точка с запятой (;) завершает команду SQL. Она не может находиться нигде внутри команды, за исключением строковых констант или идентификаторов в кавычках.
- Двоеточие (:) применяется для выборки «срезов» массивов. В некоторых диалектах SQL двоеточие может быть префиксом в имени переменной.
- Звездочка (\*) используется в некоторых контекстах как обозначение всех полей строки или составного значения. Она также имеет специальное значение, когда используется как аргумент некоторых агрегатных функций, а именно функций, которым не нужны явные параметры.
- Точка (.) используется в числовых константах, а также для отделения имен схемы, таблицы и столбца.

#### 4.1.5. Комментарии

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

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

Кроме этого, блочные комментарии можно записывать в стиле C:

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

где комментарий начинается с /\* и продолжается до соответствующего вхождения \*/.



#### 4.1.6. Приоритеты операторов

Большинство операторов имеют одинаковый приоритет и вычисляются слева направо. Приоритет и очередность операторов жестко фиксированы в синтаксическом анализаторе. В таблице 4.2 показаны приоритеты и очередность операторов, действующие в PostgreSQL.

Таблица 4.2 – Приоритет операторов (от большего к меньшему)

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

Правила приоритета операторов также применяются к операторам, определенным пользователем с теми же именами, что и вышеперечисленные встроенные операторы.

Если в конструкции OPERATOR используется имя оператора со схемой,

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

тогда OPERATOR имеет приоритет по умолчанию, соответствующий в таблице 4.2 строке «любой другой оператор». Это не зависит от того, какие именно операторы находятся в конструкции OPERATOR().

## 4.2. Выражения значения

Выражения значения применяются в самых разных контекстах:

- в списке результатов команды SELECT,
- в значениях столбцов в INSERT или UPDATE
- или в условиях поиска во многих командах.

Результат такого выражения иногда называют скаляром, чтобы отличить его от результата табличного выражения (который представляет собой таблицу). А сами выражения значения часто называют скалярными (или просто выражениями). Синтаксис таких выражений позволяет вычислять значения из примитивных частей, используя арифметические, логические и другие операции.

Выражениями значения являются:

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

- скобки (предназначены для группировки подвыражений и переопределения приоритета).

#### 4.2.1. Ссылки на столбцы

Ссылку на столбец можно записать в форме:

отношение.имя\_столбца

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

#### 4.2.2. Позиционные параметры

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

\$число

#### 4.2.3. Индексы элементов

Если в выражении есть массив, то можно извлечь его определенный элемент, написав:

выражение [индекс]

или несколько соседних элементов («срез массива»):

выражение [нижний\_индекс:верхний\_индекс]

(Здесь квадратные скобки [ ] должны присутствовать буквально.) Каждый индекс сам по себе является выражением, результат которого округляется к ближайшему целому.

В общем случае выражение массива должно заключаться в круглые скобки, но их можно опустить, когда выражение с индексом – это просто ссылка на столбец или

позиционный параметр. Кроме того, можно соединить несколько индексов, если исходный массив многомерный.

#### 4.2.4. Выбор поля

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

```
выражение.имя_поля
```

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

```
моя_таблица.столбец  
$1.столбец  
(функция_кортеж(a,b)).стол3
```

Таким образом, полная ссылка на столбец – это просто частный случай выбора поля. Важный особый случай здесь – извлечение поля из столбца составного типа:

```
(составной_столбец).поле  
(моя_таблица.составной_столбец).поле
```

Здесь скобки нужны, чтобы показать, что составной\_столбец – это имя столбца, а не таблицы, и что моя\_таблица – имя таблицы, а не схемы.

#### 4.2.5. Применение оператора

Существуют два возможных синтаксиса применения операторов:

выражение оператор выражение (бинарный инфиксный оператор)

оператор выражение (унарный префиксный оператор)

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

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

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

#### 4.2.6. Вызовы функций

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

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

Аргументам могут быть присвоены необязательные имена.



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

#### 4.2.7. Агрегатные выражения

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

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

– агрегатная функция вызывается для каждой строки.

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

– эквивалентна первой, так как указание ALL подразумевается по умолчанию.

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

– агрегатная функция вызывается для всех различных значений выражения (или набора различных значений, для нескольких выражений), выделенных во входных данных.

```
агрегатная_функция ( * ) [ FILTER ( WHERE условие_фильтра ) ]
```

– агрегатная функция вызывается для каждой строки, так как никакого конкретного значения не указано (обычно это имеет смысл только для функции count(\*)).

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

– используются сортирующие агрегатные функции, которые будут описаны ниже.

агрегатная\_функция – имя ранее определенной агрегатной функции (возможно, дополненное именем схемы),

выражение – любое выражение значения, не содержащее в себе агрегатного выражения или вызова оконной функции.

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

count(\*) подсчитывает общее количество строк, а count(f1) только количество строк, в которых f1 не NULL (так как count игнорирует NULL), а count(distinct f1) подсчитывает число различных и отличных от NULL значений столбца f1.

Обычно строки данных передаются агрегатной функции в неопределенном порядке и во многих случаях это не имеет значения. Функция min выдает один и тот же результат независимо от порядка поступающих данных. Однако некоторые агрегатные функции (такие как array\_agg и string\_agg) выдают результаты, зависящие от порядка данных. Для таких агрегатных функций можно добавить предложение\_order\_by и задать нужный порядок. Это предложение\_order\_by имеет тот же синтаксис, что и предложение ORDER BY на уровне запроса, за исключением того, что его выражения должны быть просто выражениями, а не именами результирующих столбцов или числами.

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



При использовании агрегатных функций с несколькими аргументами, предложение ORDER BY идет после всех аргументов.

Поэтому необходимо писать так:

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

а не так:

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

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

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

При добавлении ORDER BY в обычный список аргументов агрегатной функции, описанном до этого, выполняется сортировка входных строк для универсальных и статистических агрегатных функций, для которых сортировка необязательна. Но есть подмножество агрегатных функций, сортирующие агрегатные функции, для которых предложение `_order` является обязательным, обычно потому, что вычисление этой функции имеет смысл только при определенной сортировке входных строк.

Для сортирующей агрегатной функции предложение `_order_by` записывается внутри WITHIN GROUP (...). Выражения в предложении `_order_by` вычисляются однократно для каждой входной строки как аргументы обычной агрегатной функции, сортируются в соответствии с требованием предложения `_order_by` и поступают в агрегатную функцию как входящие аргументы.



Если же предложение `_order_by` находится не в `WITHIN GROUP`, оно не передается как аргумент(ы) агрегатной функции.

Выражения-аргументы, предшествующие `WITHIN GROUP` (если они есть), называются непосредственными аргументами, а выражения, указанные в предложении `_order_by` – агрегируемыми аргументами.

В отличие от аргументов обычной агрегатной функции, непосредственные аргументы вычисляются однократно для каждого вызова функции, а не для каждой строки. Это значит, что они могут содержать переменные, только если эти переменные сгруппированы в `GROUP BY`. Непосредственные аргументы обычно используются для указания значения процентиля, которое имеет смысл, только если это конкретное число для всего расчета агрегатной функции. Список непосредственных аргументов может быть пуст; в этом случае необходимо указывать `()`, но не `(*)`.

Пример вызова сортирующей агрегатной функции:

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

percentile_cont
-----
50489
```

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

Агрегатное выражение может фигурировать только в списке результатов или в предложении `HAVING` команды `SELECT`. Во всех остальных предложениях они запрещены, так как эти предложения логически вычисляются до того, как формируются результаты агрегатных функций.

Когда агрегатное выражение используется в подзапросе, оно обычно вычисляется для всех строк подзапроса. Но если в аргументах (или в условии `_filter`) агрегатной функции есть только переменные внешнего уровня, агрегатная функция относится к ближайшему внешнему уровню и вычисляется для всех строк соответствующего запроса. Такое агрегатное выражение в целом является внешней ссылкой для своего подзапроса и на



каждом вычислении считается константой. При этом допустимое положение агрегатной функции ограничивается списком результатов и предложением HAVING на том уровне запросов, где она находится.

#### 4.2.8. Вызовы оконных функций

Вызов оконной функции представляет собой применение функции, подобной агрегатной, к некоторому набору строк, выбранному запросом. Оконная функция имеет доступ ко всем строкам, вошедшим в группу текущей строки согласно указанию группировки (списку PARTITION BY) в вызове оконной функции. Вызов оконной функции может иметь следующие формы:

```
имя_функции ([выражение [, выражение ... ]]) [ FILTER ( WHERE
предложение_фильтра ) ] OVER имя_окна

имя_функции ([выражение [, выражение ... ]]) [ FILTER ( WHERE
предложение_фильтра ) ] OVER ( определение_окна )

имя_функции ( * ) [ FILTER ( WHERE предложение_фильтра ) ] OVER
имя_окна

имя_функции ( * ) [ FILTER ( WHERE предложение_фильтра ) ] OVER
( определение_окна )
```

Здесь определение\_окна записывается в виде

```
[ имя_существующего_окна ]

[ PARTITION BY выражение [, ...] ]

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

[ определение_рамки ]
```

Необязательное определение\_рамки может иметь вид:

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

{ RANGE | ROWS | GROUPS } BETWEEN начало_рамки AND конец_рамки
[ исключение_рамки ]
```

Здесь начало\_рамки и конец\_рамки задаются одним из следующих способов:

UNBOUNDED PRECEDING  
смещение PRECEDING  
CURRENT ROW  
смещение FOLLOWING  
UNBOUNDED FOLLOWING

и исключение\_рамки может быть следующим:

EXCLUDE CURRENT ROW  
EXCLUDE GROUP  
EXCLUDE TIES  
EXCLUDE NO OTHERS

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

имя\_окна – ссылка на именованное окно, определенное предложением WINDOW в данном запросе. Также возможно написать в скобках полное определение\_окна, используя тот же синтаксис определения именованного окна в предложении WINDOW.



Запись OVER имя\_окна не полностью равнозначна OVER (имя\_окна ...); последний вариант подразумевает копирование и изменение определения окна и не будет допустимым, если определение этого окна включает определение рамки.

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

принимает имена выходных столбцов или числа. Без ORDER BY строки обрабатываются в неопределенном порядке.

определение\_рамки задает набор строк, образующих рамку окна, которая представляет собой подмножество строк текущего раздела и используется для оконных функций, работающих с рамкой, а не со всем разделом. Подмножество строк в рамке может меняться в зависимости от того, какая строка является текущей. Рамку можно задать в режимах RANGE, ROWS или GROUPS; в каждом случае она начинается с положения начало\_рамки и заканчивается положением конец\_рамки. Если конец\_рамки не задается явно, подразумевается CURRENT ROW (текущая строка).

Если начало\_рамки задано как UNBOUNDED PRECEDING, рамка начинается с первой строки раздела, а если конец\_рамки определен как UNBOUNDED FOLLOWING, рамка заканчивается последней строкой раздела.

В режиме RANGE или GROUPS начало\_рамки, заданное как CURRENT ROW, определяет в качестве начала первую родственную строку (строку, которая при сортировке согласно указанному для окна предложению ORDER BY считается равной текущей), тогда как конец\_рамки, заданный как CURRENT ROW, определяет концом рамки последнюю родственную строку. В режиме ROWS вариант CURRENT ROW просто обозначает текущую строку.

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

- в режиме ROWS смещение должно задаваться отличным от NULL неотрицательным целым числом, и это число определяет сдвиг, с которым начало рамки позиционируется перед текущей строкой, а конец – после текущей строки.

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

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

- в режиме RANGE для использования этих указаний предложение ORDER BY должно содержать ровно один столбец. В этом случае смещение задает максимальную разницу между значением этого столбца в текущей строке и значением его же в предшествующих или последующих строках рамки. Тип данных выражения смещение зависит от типа данных упорядочивающего столбца. Для числовых столбцов это обычно тот же числовой тип, а для столбцов с типом дата/время – тип interval. Значение смещение при этом может так же быть отличным от NULL и неотрицательным, хотя что считать «неотрицательным», будет зависит от типа данных.



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

Дополнение исключение\_рамки позволяет исключить из рамки строки, которые окружают текущую строку, даже если они должны быть включены согласно указаниям, определяющим начало и конец рамки.

EXCLUDE CURRENT ROW исключает из рамки текущую строку.

EXCLUDE GROUP исключает из рамки текущую строку и родственные ей согласно порядку сортировки.

EXCLUDE TIES исключает из рамки все родственные строки для текущей, но не собственно текущую строку.

EXCLUDE NO OTHERS просто явно выражает поведение по умолчанию – не исключает ни текущую строку, ни родственные ей.

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

Действуют также следующие ограничения:

- в качестве начала\_рамки нельзя задать UNBOUNDED FOLLOWING;
- в качестве конца\_рамки не допускается UNBOUNDED PRECEDING и конец\_рамки не может идти в показанном выше списке указаний начало\_рамки AND конец\_рамки перед началом\_рамки.

В частности, синтаксис RANGE BETWEEN CURRENT ROW AND смещение PRECEDING не допускается. Но при этом определение ROWS BETWEEN 7 PRECEDING AND 8 PRECEDING допустимо, хотя оно и не выберет никакие строки.

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

Запись со звездочкой (\*) применяется при вызове не имеющих параметров агрегатных функций в качестве оконных count(\*) OVER (PARTITION BY x ORDER BY y). Звездочка (\*) обычно не применяется для исключительно оконных функций. Такие функции не допускают использования DISTINCT и ORDER BY в списке аргументов функции.

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

#### 4.2.9. Приведения типов

Приведение типа определяет преобразование данных из одного типа в другой. PostgreSQL воспринимает две равносильные записи приведения типов:

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

Запись с CAST соответствует стандарту SQL, тогда как вариант с :: – историческое наследие PostgreSQL.

Когда приведению подвергается значение выражения известного типа, происходит преобразование типа во время выполнения. Это приведение будет успешным, только если определен подходящий оператор преобразования типов.

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

Также можно записать приведение типа как вызов функции:

```
имя_типа ( выражение )
```

Это будет работать только для типов, имена которых являются также допустимыми именами функций. Имена типов `interval`, `time` и `timestamp` из-за синтаксического конфликта можно использовать в такой записи только в кавычках. Таким образом, запись приведения типа в виде вызова функции провоцирует несоответствия.



Приведение типа, представленное в виде вызова функции, на самом деле соответствует внутреннему механизму. Даже при использовании двух стандартных типов записи внутри происходит вызов зарегистрированной функции, выполняющей преобразование. По соглашению именем такой функции преобразования является имя выходного типа, и таким образом запись «в виде вызова функции» есть не что иное, как прямой вызов нижележащей функции преобразования.

#### 4.2.10. Применение правил сортировки

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

```
выражение COLLATE правило_сортировки
```

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

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

Предложение COLLATE имеет два распространенных применения:

- переопределение порядка сортировки в предложении ORDER BY:

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

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

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

Во втором случае предложение COLLATE добавлено к аргументу оператора, на действие которого необходимо повлиять. При этом не имеет значения, к какому именно аргументу оператора или функции добавляется COLLATE, так как правило сортировки, применяемое к оператору или функции, выбирается при рассмотрении всех аргументов, а явное предложение COLLATE переопределяет правила сортировки для всех других аргументов. Таким образом, эта команда выдаст тот же результат:

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

Но это будет ошибкой:

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

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

#### 4.2.11. Скалярные подзапросы

Скалярный подзапрос – это обычный запрос SELECT в скобках, который возвращает ровно одну строку и один столбец. После выполнения запроса SELECT его единственный результат используется в окружающем его выражении. В качестве скалярного подзапроса нельзя использовать запросы, возвращающие более одной строки или столбца.



Но если в результате выполнения подзапрос не вернет строк, скалярный результат считается равным NULL.

В подзапросе можно ссылаться на переменные из окружающего запроса; в процессе одного вычисления подзапроса они будут считаться константами.

#### 4.2.12. Конструкторы массивов

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

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

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

По умолчанию типом элементов массива считается общий тип для всех выражений, определенный по правилам, действующим и для конструкций UNION и CASE.

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

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

      array
-----
 {{1,2},{3,4}}
(1 row)

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

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

Многомерные массивы должны быть прямоугольными, и поэтому внутренние конструкторы одного уровня должны создавать вложенные массивы одинаковой



размерности. Любое приведение типа, примененное к внешнему конструктору ARRAY, автоматически распространяется на все внутренние.

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

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

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

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

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

Можно создать пустой массив, но так как массив не может быть нетипизированным, необходимо привести пустой массив к нужному типу.

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

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

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

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

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

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

```

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

```

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

#### 4.2.13. Конструкторы табличных строк

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

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

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

Конструктор строки поддерживает запись составное\_значение.\*, при этом данное значение будет развернуто в список элементов, так же, как в записи .\* на верхнем уровне списка SELECT. Если таблица t содержит столбцы f1 и f2, эти записи равнозначны:

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

По умолчанию значение, созданное выражением ROW, имеет тип анонимной записи. Если необходимо, его можно привести к именованному составному типу – либо к типу строки таблицы, либо составному типу, созданному оператором CREATE TYPE AS. Явное приведение может потребоваться для достижения однозначности.

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

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

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

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

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

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

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

- Теперь приведение необходимо для однозначного выбора функции:

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

ОШИБКА: функция getf1(record) не уникальна

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

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

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

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

Используя конструктор строк, можно создавать составное значение для сохранения в столбце составного типа или для передачи функции, принимающей составной параметр.

#### 4.2.14. Правила вычисления выражений

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

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

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

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

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



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

Как следствие, в сложных выражениях не стоит использовать функции с побочными эффектами. Особенно опасно рассчитывать на порядок вычисления или побочные эффекты в предложениях `WHERE` и `HAVING`, так как эти предложения тщательно оптимизируются при построении плана выполнения. Логические выражения (сочетания `AND/OR/NOT`) в этих предложениях могут быть видоизменены любым способом, допустимым законами Булевой алгебры.

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

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

Безопасный вариант:

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

Применяемая так конструкция `CASE` защищает выражение от оптимизации, поэтому использовать ее нужно только при необходимости.

Однако `CASE` не всегда спасает в подобных случаях. Показанный выше прием плох тем, что не предотвращает раннее вычисление константных подвыражений. Функции и

операторы, помеченные как IMMUTABLE, могут вычисляться при планировании, а не выполнении запроса. Поэтому в примере

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

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

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

Еще один подобный недостаток этого подхода в том, что CASE не может предотвратить вычисление заключенного в нем агрегатного выражения, так как агрегатные выражения вычисляются перед всеми остальными в списке SELECT или предложении HAVING. В следующем запросе может возникнуть ошибка деления на ноль, несмотря на то, что он вроде бы защищен от нее:

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

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

### 4.3. Вызов функций

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

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

PostgreSQL также поддерживает смешанную передачу, когда параметры передаются и по именам, и по позиции. В этом случае позиционные параметры должны идти перед параметрами, передаваемыми по именам.

Ниже рассмотрены все три варианта записи на примере следующей функции:

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

Функция `concat_lower_or_upper` имеет два обязательных параметра: `a` и `b`. Кроме того, есть один необязательный параметр `uppercase`, который по умолчанию имеет значение `false`.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Аргументы a и b будут сложены вместе и переведены в верхний или нижний регистр, в зависимости от параметра uppercase.

#### 4.3.1. Позиционная передача

Позиционная передача – это традиционный механизм передачи аргументов функции в PostgreSQL. Пример такой записи:

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

concat_lower_or_upper
-----
HELLO WORLD
(1 row)
```

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

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

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

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

#### 4.3.2. Именная передача

При именной передаче для аргумента добавляется имя, которое отделяется от выражения значения знаками =>.

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World');

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

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

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

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

Для обратной совместимости поддерживается и старый синтаксис с «:=»:

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

#### 4.3.3. Смешанная передача

При смешанной передаче параметры передаются и по именам, и по позиции, но именованные аргументы не могут стоять перед позиционными.

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

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



## 5. ТИПЫ ДАННЫХ


PostgreSQL предоставляет пользователям богатый ассортимент встроенных типов данных. Кроме того, пользователи могут создавать свои типы в PostgreSQL, используя команду CREATE TYPE.

Таблица 5.1 содержит все встроенные типы данных общего пользования. Многие из альтернативных имен, приведенных в столбце «Псевдонимы», используются внутри PostgreSQL по историческим причинам. В этот список не включены некоторые устаревшие типы и типы для внутреннего применения.

Таблица 5.1 – Типы данных

Имя	Псевдонимы	Описание
bigint	int8	знаковое целое из 8 байт
bigserial	serial8	восьмибайтное целое с автоувеличением
bit [ (n) ]		битовая строка фиксированной длины
bit varying [ (n) ]	varbit [ (n) ]	битовая строка переменной длины
boolean	bool	логическое значение (true/false)
box		прямоугольник в плоскости
bytea		двоичные данные («массив байт»)
character [ (n) ]	char [ (n) ]	символьная строка фиксированной длины
character varying [ (n) ]	varchar [ (n) ]	символьная строка переменной длины
cidr		сетевой адрес IPv4 или IPv6
circle		круг в плоскости
date		календарная дата (год, месяц, день)
double precision	float8	число двойной точности с плавающей точкой (8 байт)
inet		адрес узла IPv4 или IPv6
integer	int, int4	знаковое четырехбайтное целое
interval [ поля ] [ (p) ]		интервал времени
json		текстовые данные JSON
jsonb		двоичные данные JSON, разобранные
line		прямая в плоскости
lseg		отрезок в плоскости
macaddr		MAC-адрес
macaddr8		адрес MAC (Media Access Control) (в формате EUI-64)
money		денежная сумма

Имя	Псевдонимы	Описание
numeric [ (p, s) ]	decimal [ (p, s) ]	вещественное число заданной точности
path		геометрический путь в плоскости
pg_lsn		последовательный номер в журнале PostgreSQL
pg_snapshot		снимок идентификатора транзакций
point		геометрическая точка в плоскости
polygon		замкнутый геометрический путь в плоскости
real	float4	число одинарной точности с плавающей точкой (4 байта)
smallint	int2	знаковое двухбайтное целое
smallserial	serial2	двухбайтное целое с автоувеличением
serial	serial4	четырёхбайтное целое с автоувеличением
text		символьная строка переменной длины
time [ (p) ] [ without time zone ]		время суток (без часового пояса)
time [ (p) ] with time zone	timetz	время суток с учетом часового пояса
timestamp [ (p) ] [ without time zone ]		дата и время (без часового пояса)
timestamp [ (p) ] with time zone	timestampz	дата и время с учетом часового пояса
tsquery		запрос текстового поиска
tsvector		документ для текстового поиска
txid_snapshot		снимок идентификаторов транзакций для пользовательского уровня
uuid		универсальный уникальный идентификатор
xml		XML-данные

 В стандарте SQL описаны следующие типы (или их имена): bigint, bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (с часовым поясом и без), timestamp (с часовым поясом и без), xml.

Каждый тип данных имеет внутреннее представление, скрытое функциями ввода и вывода. При этом многие встроенные типы стандартны и имеют очевидные внешние форматы. Однако есть типы, уникальные для PostgreSQL геометрические пути, и есть типы, которые могут иметь разные форматы, дату и время. Некоторые функции ввода и вывода не

являются в точности обратными друг к другу, то есть результат функции вывода может не совпадать со входным значением из-за потери точности.

## 5.1. Числовые типы

Числовые типы включают двух-, четырех- и восьмибайтные целые, четырех- и восьмибайтные числа с плавающей точкой, а также десятичные числа с задаваемой точностью. Все эти типы перечислены в таблице 5.2.

Таблица 5.2 – Числовые типы

Имя	Размер	Описание	Диапазон
smallint	2 байта	целое в небольшом диапазоне	-32768 .. +32767
integer	4 байта	типичный выбор для целых чисел	-2147483648 .. +2147483647
bigint	8 байт	целое в большом диапазоне	-9223372036854775808 .. 9223372036854775807
decimal	переменный	вещественное число с указанной точностью	до 131072 цифр до десятичной точки и до 16383 — после
numeric	переменный	вещественное число с указанной точностью	до 131072 цифр до десятичной точки и до 16383 — после
real	4 байта	вещественное число с переменной точностью	точность в пределах 6 десятичных цифр
double precision	8 байт	вещественное число с переменной точностью	точность в пределах 15 десятичных цифр
smallserial	2 байта	небольшое целое с автоувеличением	1 .. 32767
serial	4 байта	целое с автоувеличением	1 .. 2147483647
bigserial	8 байт	большое целое с автоувеличением	1 .. 9223372036854775807

Для этих типов определен полный набор соответствующих арифметических операторов и функций.

### 5.1.1. Целочисленные типы

Типы smallint, integer и bigint хранят целые числа, то есть числа без дробной части, имеющие разные допустимые диапазоны. Попытка сохранить значение, выходящее за рамки диапазона, приведет к ошибке.

Чаще всего используется тип integer, как наиболее сбалансированный выбор ширины диапазона, размера и быстродействия. Тип smallint обычно применяется только когда крайне

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

важно уменьшить размер данных на диске. Тип `bigint` предназначен для тех случаев, когда числа не умещаются в диапазон типа `integer`.

В SQL определены только типы `integer` (или `int`), `smallint` и `bigint`. Имена типов `int2`, `int4` и `int8` выходят за рамки стандарта, хотя могут работать и в некоторых других СУБД.

### 5.1.2. Числа с произвольной точностью

Тип `numeric` позволяет хранить числа с очень большим количеством цифр. Он особенно рекомендуется для хранения денежных сумм и других величин, где важна точность. Вычисления с типом `numeric` дают точные результаты, где это возможно, при сложении, вычитании и умножении. Однако операции со значениями `numeric` выполняются гораздо медленнее, чем с целыми числами или с типами с плавающей точкой, описанными в следующем разделе.

Ниже используются следующие термины: масштаб значения `numeric` определяет количество десятичных цифр в дробной части, справа от десятичной точки, а точность — общее количество значимых цифр в числе, т. е. количество цифр по обе стороны десятичной точки. Например, число 23.5141 имеет точность 6 и масштаб 4. Целочисленные значения можно считать числами с масштабом 0.

Для столбца типа `numeric` можно настроить и максимальную точность, и максимальный масштаб. Столбец типа `numeric` объявляется следующим образом:

```
NUMERIC (точность, масштаб)
```

Точность должна быть положительной, а масштаб положительным или равным нулю. Альтернативный вариант

```
NUMERIC (точность)
```

устанавливает масштаб 0. Форма:

```
NUMERIC
```

без указания точности и масштаба создает столбец типа «неограниченное число», в котором можно сохранять числовые значения любой длины до предела, обусловленного реализацией. В столбце этого типа входные значения не будут приводиться к какому-либо

масштабу, тогда как в столбцах numeric с явно заданным масштабом значения подгоняются под этот масштаб.

Если масштаб значения, которое нужно сохранить, превышает объявленный масштаб столбца, система округлит его до заданного количества цифр после точки. Если же после этого количество цифр слева в сумме с масштабом превысит объявленную точность, произойдет ошибка.

Числовые значения физически хранятся без каких-либо дополняющих нулей слева или справа. Таким образом, объявляемые точность и масштаб столбца определяют максимальный, а не фиксированный размер хранения. Действительный размер хранения такого значения складывается из двух байт для каждой группы из четырех цифр и дополнительных трех-восьми байт.

В дополнение к обычным числовым значениям типы с плавающей точкой могут содержать следующие специальные значения:

```
Infinity  
-Infinity  
NaN
```

Они представляют особые значения, описанные в IEEE 754, соответственно «бесконечность», «минус бесконечность» и «не число». Записывая эти значения в виде констант в команде SQL, их нужно заключать в апострофы, например так: UPDATE table SET x = '-Infinity'. Регистр символов в этих строках не важен. В качестве альтернативы значения бесконечности могут быть записаны как inf и -inf.

Значения бесконечности соответствуют ожиданиям с точки зрения математики. Infinity плюс любое конечное значение равно Infinity, как и Infinity плюс Infinity; но Infinity минус Infinity дает NaN (не число), потому что в результате получается неопределенность. Бесконечность может быть сохранена только в столбце типа «неограниченный numeric», потому что она теоретически превышает любой конечный предел точности.

Значение NaN (не число) используется для представления неопределенных результатов вычислений. Вообще, любая операция с NaN выдает в результате тоже NaN. Единственное исключение составляют операции, которые дают один и тот же результат для

любого числового значения, конечного или бесконечного — этот результат будет выдаваться и для NaN. Пример этого принципа: результатом NaN в нулевую степень будет единица.



В большинстве реализаций «не число» (NaN) считается не равным любому другому значению (в том числе и самому NaN). Чтобы значения numeric можно было сортировать и использовать в древовидных индексах, PostgreSQL считает, что значения NaN равны друг другу и при этом больше любых числовых значений (не NaN).

Типы decimal и numeric равнозначны. Оба эти типа описаны в стандарте SQL.

При округлении значений тип numeric выдает число, большее по модулю, тогда как на большинстве платформ типы real и double precision выдают ближайшее четное число.


### 5.1.3. Типы с плавающей точкой

Типы данных real и double precision хранят приближенные числовые значения с переменной точностью. На всех поддерживаемых в настоящее время платформах эти типы реализуют стандарт IEEE 754 для двоичной арифметики с плавающей точкой (с одинарной и двойной точностью соответственно), в той мере, в какой его поддерживают процессор, операционная система и компилятор.

Неточность здесь выражается в том, что некоторые значения, которые нельзя преобразовать во внутренний формат, сохраняются приближенно, так что полученное значение может несколько отличаться от записанного. Управление подобными ошибками и их распространение в процессе вычислений является предметом изучения целого раздела математики и компьютерной науки.


На всех поддерживаемых сейчас платформах тип real может сохранить значения примерно от  $1E-37$  до  $1E+37$  с точностью не меньше 6 десятичных цифр. Тип double precision предлагает значения в диапазоне приблизительно от  $1E-307$  до  $1E+308$  и с точностью не меньше 15 цифр. Попытка сохранить слишком большие или слишком маленькие значения приведет к ошибке. Если точность вводимого числа слишком велика, оно будет округлено. При попытке сохранить число, близкое к 0, но непредставимое как отличное от 0, произойдет ошибка антипереполнения.

По умолчанию числа с плавающей точкой выводятся в текстовом виде в кратчайшем точном десятичном представлении; выводимое десятичное значение оказывается более близким к изначальному двоичному числу, чем любое другое значение, представимое с той же двоичной точностью. (Однако выводимое значение в текущей реализации никогда не находится точно посередине между двумя представимыми двоичными значениями, во избежание распространенной ошибки с функциями ввода, не учитывающими корректно правило округления до ближайшего четного.) Выводимое значение может занимать не больше 17 значащих десятичных цифр для типа float8 и не больше 9 цифр для типа float4.

 Преобразование в кратчайший точный вид производится гораздо быстрее, чем в традиционное представление с округлением.

Для совместимости с результатами, выдаваемыми старыми версиями PostgreSQL, и уменьшения точности выводимых чисел, когда это требуется, в параметре `extra_float_digits` можно выбрать также вариант округленного десятичного вывода. Со значением 0 восстанавливается действовавшее ранее по умолчанию округление числа до 6 (для типа float4) или 15 (для float8) значащих десятичных цифр. При отрицательных значениях число значащих цифр уменьшается дополнительно; при -2 результат будет округлен до 4 или 13 цифр, соответственно.

При любом значении `extra_float_digits`, большем 0, выбирается кратчайшее точное представление.

 Приложения, которым были нужны точные числовые значения, раньше задавали для параметра `extra_float_digits` значение 3, чтобы получить их. Они могут продолжать использовать это значение для максимальной совместимости с разными версиями.

В дополнение к обычным числовым значениям типы с плавающей точкой могут содержать следующие специальные значения:

```
Infinity
-Infinity
NaN
```

Они представляют особые значения, описанные в IEEE 754, соответственно «бесконечность», «минус бесконечность» и «не число». Записывая эти значения в виде констант в команде SQL, их нужно заключать в апострофы, например так: UPDATE table SET x = '-Infinity'. Регистр символов в этих строках не важен. В качестве альтернативы значения бесконечности могут быть записаны как inf и -inf.



Согласно IEEE 754, NaN не должно считаться равным любому другому значению с плавающей точкой (в том числе и самому NaN). Чтобы значения с плавающей точкой можно было сортировать и использовать в древовидных индексах, PostgreSQL считает, что значения NaN равны друг другу, и при этом больше любых числовых значений (не NaN).

PostgreSQL также поддерживает форматы float и float(p), оговоренные в стандарте SQL, для указания неточных числовых типов. Здесь p определяет минимально допустимую точность в двоичных цифрах. PostgreSQL воспринимает запись от float(1) до float(24) как выбор типа real, а запись от float(25) до float(53) как выбор типа double precision. Значения p вне допустимого диапазона вызывают ошибку. Если float указывается без точности, подразумевается тип double precision.

#### 5.1.4. Последовательные типы



В этом разделе описывается специфичный для PostgreSQL способ создания столбца с автоувеличением. Другой способ, соответствующий стандарту SQL, заключается в использовании столбцов идентификации и рассматривается в описании CREATE TABLE.

Типы данных smallserial, serial и bigserial не являются настоящими типами, а представляют собой просто удобное средство для создания столбцов с уникальными идентификаторами (подобное свойству AUTO\_INCREMENT в некоторых СУБД). В текущей реализации запись:

```
CREATE TABLE имя_таблицы (
    имя_столбца SERIAL
);
```

равнозначна следующим командам:

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------



```
CREATE SEQUENCE имя_таблицы_имя_столбца_seq AS integer;
CREATE TABLE имя_таблицы (
    имя_столбца integer NOT NULL DEFAULT
    nextval('имя_таблицы_имя_столбца_seq')
);
ALTER SEQUENCE имя_таблицы_имя_столбца_seq OWNED BY
имя_таблицы.имя_столбца;
```

То есть при определении такого типа создается целочисленный столбец со значением по умолчанию, извлекаемым из генератора последовательности. Чтобы в столбец нельзя было вставить NULL, в его определение добавляется ограничение NOT NULL. (Во многих случаях также имеет смысл добавить для этого столбца ограничения UNIQUE или PRIMARY KEY для защиты от ошибочного добавления дублирующихся значений, но автоматически это не происходит.) Последняя команда определяет, что последовательность «принадлежит» столбцу, так что она будет удалена при удалении столбца или таблицы.



Так как типы smallserial, serial и bigserial реализованы через последовательности, в числовом ряду значений столбца могут образовываться пропуски, даже если никакие строки не удалялись. Значение, выделенное из последовательности, считается «задействованным», даже если строку с этим значением не удалось вставить в таблицу. Это может произойти при откате транзакции, добавляющей данные.

Чтобы вставить в столбец serial следующее значение последовательности, ему нужно присвоить значение по умолчанию. Это можно сделать, либо исключив его из списка столбцов в операторе INSERT, либо с помощью ключевого слова DEFAULT.

Имена типов serial и serial4 равнозначны: они создают столбцы integer. Так же являются синонимами имена bigserial и serial8, но они создают столбцы bigint. Тип bigserial следует использовать, если за все время жизни таблицы планируется использовать больше чем 231 значений. И наконец, синонимами являются имена типов smallserial и serial2, но они создают столбец smallint.

Последовательность, созданная для столбца `serial`, автоматически удаляется при удалении связанного столбца. Последовательность можно удалить и отдельно от столбца, но при этом также будет удалено определение значения по умолчанию.

## 5.2. Денежные типы

Тип `money` хранит денежную сумму с фиксированной дробной частью. Точность дробной части определяется на уровне базы данных параметром `lc_monetary`. Для диапазона, показанного в таблице, предполагается, что число содержит два знака после запятой. Входные данные могут быть записаны по-разному, в том числе в виде целых и дробных чисел, а также в виде строки в денежном формате, например `$1,000.00'`. Выводятся эти значения обычно в денежном формате, зависящем от региональных стандартов.

Таблица 5.3 – Денежные типы

Имя	Размер	Описание	Диапазон
<code>money</code>	8 байт	денежная сумма	-92233720368547758.08 .. +92233720368547758.07

Так как выводимые значения этого типа зависят от региональных стандартов, попытка загрузить данные типа `money` в базу данных с другим параметром `lc_monetary` может быть неудачной. Во избежание подобных проблем, прежде чем восстанавливать копию в новую базу данных, убедитесь в том, что параметр `lc_monetary` в этой базе данных имеет то же значение, что и в исходной.

Значения типов `numeric`, `int` и `bigint` можно привести к типу `money`. Преобразования типов `real` и `double precision` так же возможны через тип `numeric`,

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

Однако использовать числа с плавающей точкой для денежных сумм не рекомендуется из-за возможных ошибок округления.

Значение `money` можно привести к типу `numeric` без потери точности. Преобразование в другие типы может быть неточным и также должно выполняться в два этапа:

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

При делении значения типа `money` на целое число выполняется отбрасывание дробной части и получается целое, ближайшее к нулю. Чтобы получить результат с округлением,

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

выполните деление значения с плавающей точкой или приведите значение типа money к numeric до деления, а затем приведите результат к типу money. Последний вариант предпочтительнее, так как исключает риск потери точности.

Когда значение money делится на другое значение money, результатом будет значение типа double precision (то есть просто число, не денежная величина); денежные единицы измерения при делении сокращаются.

### 5.3. Символьные типы

Таблица 5.4 – Символьные типы

Имя	Описание
character varying( <i>n</i> ), varchar( <i>n</i> )	строка ограниченной переменной длины
character( <i>n</i> ), char( <i>n</i> )	строка фиксированной длины, дополненная пробелами
text	строка неограниченной переменной длины

В таблице 5.4 перечислены символьные типы общего назначения, доступные в PostgreSQL.

SQL определяет два основных символьных типа: character varying(*n*) и character(*n*), где *n* — положительное число. Оба эти типа могут хранить текстовые строки длиной до *n* символов (не байт). Попытка сохранить в столбце такого типа более длинную строку приведет к ошибке, если только все лишние символы не являются пробелами (тогда они будут усечены до максимально допустимой длины). Если длина сохраняемой строки оказывается меньше объявленной, значения типа character будут дополняться пробелами; а тип character varying просто сохранит короткую строку.

При попытке явно привести значение к типу character varying(*n*) или character(*n*), часть строки, выходящая за границу в *n* символов, удаляется, не вызывая ошибки.

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

Помимо этого, PostgreSQL предлагает тип text, в котором можно хранить строки произвольной длины. Хотя тип text не описан в стандарте SQL, его поддерживают и некоторые другие СУБД SQL.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Значения типа `character` физически дополняются пробелами до `n` символов и хранятся, а затем отображаются в таком виде. Однако при сравнении двух значений типа `character` дополняющие пробелы считаются незначащими и игнорируются. С правилами сортировки, где пробельные символы являются значащими, это поведение может приводить к неожиданным результатам, `SELECT 'a '::CHAR(2) collate "C" < E'a\n'::CHAR(2)` вернет `true` (условие будет истинным), хотя в локали `C` символ пробела считается больше символа новой строки. При приведении значения `character` к другому символьному типу дополняющие пробелы отбрасываются. Эти пробелы несут смысловую нагрузку в типах `character varying` и `text` и в проверках по шаблонам, то есть в `LIKE` и регулярных выражениях.

Какие именно символы можно сохранить в этих типах данных зависит от того, какой набор символов был выбран при создании базы данных. Однако символ с кодом 0 (иногда называемый `NUL`) сохранить нельзя, вне зависимости от выбранного набора символов.

Для хранения короткой строки (до 126 байт) требуется дополнительный 1 байт плюс размер самой строки, включая дополняющие пробелы для типа `character`. Для строк длиннее требуется не 1, а 4 дополнительных байта. Система может автоматически сжимать длинные строки, так что физический размер на диске может быть меньше. Очень длинные текстовые строки переносятся в отдельные таблицы, чтобы они не замедляли работу с другими столбцами. В любом случае максимально возможный размер строки составляет около 1 Гб. (Допустимое значение `n` в объявлении типа данных меньше этого числа. Это объясняется тем, что в зависимости от кодировки каждый символ может занимать несколько байт. Если требуется сохранять строки без определенного предела длины, нужно использовать типы `text` или `character varying` без указания длины, а не задавать какое-либо большое максимальное значение.)

По быстродействию эти три типа практически не отличаются друг от друга, не считая большего размера хранения для типа с дополняющими пробелами и нескольких машинных операций для проверки длины при сохранении строк в столбце с ограниченной длиной. Хотя в некоторых СУБД тип `character(n)` работает быстрее других, в PostgreSQL это не так; на деле `character(n)` обычно оказывается медленнее остальных типов из-за большего размера данных и более медленной сортировки. В большинстве случаев вместо него лучше применять `text` или `character varying`.

В PostgreSQL есть еще два символьных типа фиксированной длины, приведенные в таблице 5.5. Тип `name` создан только для хранения идентификаторов во внутренних системных таблицах и не предназначен для обычного применения пользователями. В настоящее время его длина составляет 64 байта (63 ASCII-символа плюс конечный знак), но в исходном коде C она задается константой `NAMEDATALEN`. Эта константа определяется во время компиляции (и ее можно менять в особых случаях), а кроме того, максимальная длина по умолчанию может быть увеличена в следующих версиях. Тип `"char"` отличается от `char(1)` тем, что он фактически хранится в одном байте. Он используется во внутренних системных таблицах для простых перечислений.

Таблица 5.5 – Специальные символьные типы

Имя	Размер	Описание
"char"	1 байт	внутренний однобайтный тип
name	64 байта	внутренний тип для имен объектов

#### 5.4. Двоичные типы данных

Для хранения двоичных данных предназначен тип `bytea`.

Таблица 5.6 – Двоичные типы данных

Имя	Размер	Описание
bytea	1 или 4 байта плюс сама двоичная строка	двоичная строка переменной длины

Двоичные строки представляют собой последовательность октетов (байт) и имеют два отличия от текстовых строк. Во-первых, в двоичных строках можно хранить байты с кодом 0 и другими «непечатаемыми» значениями (обычно это значения вне десятичного диапазона 32..126). В текстовых строках нельзя сохранять нулевые байты, а также значения и последовательности значений, не соответствующие выбранной кодировке базы данных. Во-вторых, в операциях с двоичными строками обрабатываются байты в чистом виде, тогда как текстовые строки обрабатываются в зависимости от языковых стандартов. То есть, двоичные строки больше подходят для данных, которые программист видит как «просто байты», а символьные строки — для хранения текста.

Тип `bytea` поддерживает два формата ввода и вывода: «шестнадцатеричный» и традиционный для PostgreSQL формат «спецпоследовательностей». Входные данные

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

принимаются в обоих форматах, а формат выходных данных зависит от параметра конфигурации `bytea_output`; по умолчанию выбран шестнадцатеричный.

Стандарт SQL определяет другой тип двоичных данных, BLOB (BINARY LARGE OBJECT, большой двоичный объект). Его входной формат отличается от форматов `bytea`, но функции и операторы в основном те же.

#### 5.4.1. Шестнадцатеричный формат `bytea`

В «шестнадцатеричном» формате двоичные данные кодируются двумя шестнадцатеричными цифрами на байт, при этом первая цифра соответствует старшим 4 битам. К полученной строке добавляется префикс `\x` (чтобы она отличалась от формата спецпоследовательности). В некоторых контекстах обратную косую черту нужно экранировать, продублировав ее. Вводимые шестнадцатеричные цифры могут быть в любом регистре, а между парами цифр допускаются пробельные символы (но не внутри пары и не в начале последовательности `\x`). Этот формат совместим со множеством внешних приложений и протоколов, к тому же обычно преобразуется быстрее, поэтому предпочтительнее использовать его.

Пример:

```
SELECT '\xDEADBEEF';
```

#### 5.4.2. Формат спецпоследовательностей `bytea`

Формат «спецпоследовательностей» традиционно использовался в PostgreSQL для значений типа `bytea`. В нем двоичная строка представляется в виде последовательности ASCII-символов, а байты, непредставимые в виде ASCII-символов, передаются в виде спецпоследовательностей. Этот формат может быть удобен, если с точки зрения приложения представление байт в виде символов имеет смысл. Но на практике это обычно создает путаницу, так как двоичные и символьные строки могут выглядеть одинаково, а кроме того выбранный механизм спецпоследовательностей довольно неуклюж. Поэтому в новых приложениях этот формат обычно не стоит использовать.

Передавая значения `bytea` в формате спецпоследовательности, байты с определенными значениями необходимо записывать специальным образом, хотя так можно записывать и все значения. В общем виде для этого значение байта нужно преобразовать в

трехзначное восьмеричное число и добавить перед ним обратную косую черту. Саму обратную косую черту (символ с десятичным кодом 92) можно записать в виде двух таких символов. В таблице 5.7 перечислены символы, которые нужно записывать спецпоследовательностями, и приведены альтернативные варианты записи, если они возможны.

Таблица 5.7 – Спецпоследовательности записи значений bytea

Десятичное значение байта	Описание	Спецпоследовательность ввода	Пример	Шестнадцатеричное представление
0	нулевой байт	'\000'	'\000'::bytea	\x00
39	апостроф	'"' или '\047'	'"'::bytea	\x27
92	обратная косая черта	'\' или '\134'	'\'::bytea	\x5c
от 0 до 31 и от 127 до 255	«непечатаемые» байты	E'\xxx' (восьмеричное значение)	'\001'::bytea	\x01

Требования экранирования непечатаемых символов определяются языковыми стандартами. Иногда такие символы могут восприниматься и без спецпоследовательностей.

Апострофы должны дублироваться, как показано в таблице 5.7, потому что это обязательно для любой текстовой строки в команде SQL. При общем разборе текстовой строки внешние апострофы убираются, а каждая пара внутренних сводится к одному символу. Таким образом, функция ввода bytea видит всего один апостроф, который она обрабатывает как обычный символ в данных. Дублировать же обратную косую черту при вводе bytea не требуется: этот символ считается особым и меняет поведение функции ввода, как показано в таблице 5.7.

В некоторых контекстах обратная косая черта должна дублироваться (относительно примеров выше), так как при общем разборе строковых констант пара таких символов будет сведена к одному.

Данные bytea по умолчанию выводятся в шестнадцатеричном формате (hex). Если поменять значение bytea\_output на escape, «непечатаемые» байты представляются в виде соответствующих трехзначных восьмеричных значений, которые предваряются одной обратной косой чертой. Большинство «печатаемых» байтов представляются обычными символами из клиентского набора символов,

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

```
SET bytea_output = 'escape';

SELECT 'abc \153\154\155 \052\251\124'::bytea;

      bytea
-----
abc klm *\251T
```

Байт с десятичным кодом 92 (обратная косая черта) при выводе дублируется. Это иллюстрирует таблица 5.8.

Таблица 5.8 – Спецпоследовательности выходных значений bytea

Десятичное значение байта	Описание	Спецпоследовательность вывода	Пример	Выводимый результат
92	обратная косая черта	\\	'\134'::bytea	\\
от 0 до 31 и от 127 до 255	«непечатаемые» байты	\xxx (значение байта)	'\001'::bytea	\001
от 32 до 126	«печатаемые» байты	представление из клиентского набора символов	'\176'::bytea	~

В зависимости от применяемой клиентской библиотеки PostgreSQL, для преобразования значений bytea в спецстроки и обратно могут потребоваться дополнительные действия. Если приложение сохраняет в строках символы перевода строк, возможно их также нужно будет представить спецпоследовательностями.

## 5.5. Типы даты/времени

PostgreSQL поддерживает полный набор типов даты и времени SQL, показанный в таблице 5.9. Все даты считаются по Григорианскому календарю, даже для времени до его введения.

Таблица 5.9 – Типы даты/времени

Имя	Размер	Описание	Наименьшее значение	Наибольшее значение	Точность
timestamp [ (p) ] [ without time zone ]	8 байт	дата и время (без часового пояса)	4713 до н. э.	294276 н. э.	1 микросекунда
timestamp [ (p) ] with time zone	8 байт	дата и время (с часовым поясом)	4713 до н. э.	294276 н. э.	1 микросекунда
№ изменения: _____		Подпись отв. лица: _____		Дата внесения изм: _____	



Имя	Размер	Описание	Наименьшее значение	Наибольшее значение	Точность
date	4 байта	дата (без времени суток)	4713 до н. э.	5874897 н. э.	1 день
time [ (p) ] [ without time zone ]	8 байт	время суток (без даты)	00:00:00	24:00:00	1 микросекунда
time [ (p) ] with time zone	12 байт	время дня (без даты), с часовым поясом	00:00:00+1559	24:00:00-1559	1 микросекунда
interval [ поля ] [ (p) ]	16 байт	временной интервал	-178000000 лет	178000000 лет	1 микросекунда



Стандарт SQL требует, чтобы тип timestamp подразумевал timestamp without time zone (время без часового пояса), и PostgreSQL следует этому. Для краткости timestamp with time zone можно записать как timestamptz; это расширение PostgreSQL.

Типы time, timestamp и interval принимают необязательное значение точности p, определяющее, сколько знаков после запятой должно сохраняться в секундах. По умолчанию точность не ограничивается. Допустимые значения p лежат в интервале от 0 до 6.

Тип interval дополнительно позволяет ограничить набор сохраняемых полей следующими фразами:

YEAR  
MONTH  
DAY  
HOUR  
MINUTE  
SECOND  
YEAR TO MONTH  
DAY TO HOUR  
DAY TO MINUTE  
DAY TO SECOND

HOUR TO MINUTE  
 HOUR TO SECOND  
 MINUTE TO SECOND

Если указаны и поля, и точность *p*, указание поля должно включать **SECOND**, так как точность применима только к секундам.

Тип *time with time zone* определен стандартом SQL, но в его определении описаны свойства сомнительной ценности. В большинстве случаев сочетание типов *date*, *time*, *timestamp without time zone* и *timestamp with time zone* удовлетворяет все потребности в функционале дат/времени, возникающие в приложениях.

### 5.5.1. Ввод даты/времени

Значения даты и времени принимаются практически в любом разумном формате, включая ISO 8601, SQL-совместимый, традиционный формат POSTGRES и другие. В некоторых форматах порядок даты, месяца и года во вводимой дате неоднозначен и поэтому поддерживается явное определение формата. Для этого предназначен параметр *DateStyle*. Когда он имеет значение *MDY*, выбирается интерпретация месяц-день-год, значению *DMY* соответствует день-месяц-год, а *YMD* — год-месяц-день.

PostgreSQL обрабатывает вводимые значения даты/времени более гибко, чем того требует стандарт SQL.

Помните, что любые вводимые значения даты и времени нужно заключать в апострофы, как текстовые строки. SQL предусматривает следующий синтаксис:

тип [ (*p*) ] 'значение'

Здесь *p* — необязательное указание точности, определяющее число знаков после точки в секундах. Точность может быть определена для типов *time*, *timestamp* и *interval* в интервале от 0 до 6. Если в определении константы точность не указана, она считается равной точности значения в строке (но не больше 6 цифр).

#### 5.5.1.1 Даты

В таблице 5.10 приведены некоторые допустимые значения типа *date*.

Таблица 5.10 – Вводимые даты

Пример	Описание
1999-01-08	ISO 8601; 8 января в любом режиме (рекомендуемый формат)
January 8, 1999	воспринимается однозначно в любом режиме datestyle
1/8/1999	8 января в режиме MDY и 1 августа в режиме DMY
1/18/1999	18 января в режиме MDY; недопустимая дата в других режимах
01/02/03	2 января 2003 г. в режиме MDY; 1 февраля 2003 г. в режиме DMY и 3 февраля 2001 г. в режиме YMD
1999-Jan-08	8 января в любом режиме
Jan-08-1999	8 января в любом режиме
08-Jan-1999	8 января в любом режиме
99-Jan-08	8 января в режиме YMD; ошибка в других режимах
08-Jan-99	8 января; ошибка в режиме YMD
Jan-08-99	8 января; ошибка в режиме YMD
19990108	ISO 8601; 8 января 1999 в любом режиме
990108	ISO 8601; 8 января 1999 в любом режиме
1999.008	год и день года
J2451187	юлианский день
January 8, 99 BC	99 до н. э.

### 5.5.1.2 Время

Для хранения времени суток без даты предназначены типы time [ (p) ] without time zone и time [ (p) ] with time zone. Тип time без уточнения эквивалентен типу time without time zone.

Допустимые вводимые значения этих типов состоят из записи времени суток и необязательного указания часового пояса. Если в значении для типа time without time zone указывается часовой пояс, он просто игнорируется. Так же будет игнорироваться дата, если ее указать, за исключением случаев, когда в указанном часовом поясе принят переход на летнее время. В данном случае указать дату необходимо, чтобы система могла определить, применяется ли обычное или летнее время. Соответствующее смещение часового пояса записывается в значении time with time zone.

Таблица 5.11 – Вводимое время

Пример	Описание
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Пример	Описание
040506	ISO 8601
04:05 AM	то же, что и 04:05; AM не меняет значение времени
04:05 PM	то же, что и 16:05; часы должны быть $\leq 12$
04:05:06.789-8	ISO 8601, с часовым поясом в виде смещения от UTC
04:05:06-08:00	ISO 8601, с часовым поясом в виде смещения от UTC
04:05-08:00	ISO 8601, с часовым поясом в виде смещения от UTC
040506-08	ISO 8601, с часовым поясом в виде смещения от UTC
040506+0730	ISO 8601, с часовым поясом, задаваемым нецелочисленным смещением от UTC
040506+07:30:00	смещение от UTC, заданное до секунд (не допускается в ISO 8601)
04:05:06 PST	часовой пояс задается аббревиатурой
2003-04-12 04:05:06 America/New_York	часовой пояс задается полным названием

Таблица 5.12 – Вводимый часовой пояс

Пример	Описание
PST	аббревиатура (Pacific Standard Time, Стандартное тихоокеанское время)
America/New_York	полное название часового пояса
PST8PDT	указание часового пояса в стиле POSIX
-8:00:00	смещение часового пояса PST от UTC
-8:00	смещение часового пояса PST от UTC (расширенный формат ISO 8601)
-800	смещение часового пояса PST от UTC (стандартный формат ISO 8601)
-8	смещение часового пояса PST от UTC (стандартный формат ISO 8601)
zulu	принятое у военных сокращение UTC
z	краткая форма zulu (также определена в ISO 8601)

### 5.5.1.3 Даты и время

Допустимые значения типов timestamp состоят из записи даты и времени, после которого может указываться часовой пояс и необязательное уточнение AD или BC, определяющее эпоху до нашей эры и нашу эру соответственно. (AD/BC можно указать и перед часовым поясом, но предпочтительнее первый вариант.) Допустимые варианты, соответствующие стандарту ISO 8601:

1999-01-08 04:05:06

```
1999-01-08 04:05:06 -8:00
```

В дополнение к этому поддерживается распространенный формат:

```
January 8 04:05:06 1999 PST
```

Стандарт SQL различает константы типов `timestamp without time zone` и `timestamp with time zone` по знаку «+» или «-» и смещению часового пояса, добавленному после времени. Следовательно, согласно стандарту, записи

```
TIMESTAMP '2004-10-19 10:23:54'
```

должен соответствовать тип `timestamp without time zone`, а

```
TIMESTAMP '2004-10-19 10:23:54+02'
```

тип `timestamp with time zone`. PostgreSQL никогда не анализирует содержимое текстовой строки, чтобы определить тип значения, и поэтому обе записи будут обработаны как значения типа `timestamp without time zone`. Чтобы текстовая константа обрабатывалась как `timestamp with time zone`, укажите этот тип явно:

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

В константе типа `timestamp without time zone` PostgreSQL просто игнорирует часовой пояс. То есть результирующее значение вычисляется только из полей даты/времени и не подстраивается под указанный часовой пояс.

Значения `timestamp with time zone` внутри всегда хранятся в UTC (Universal Coordinated Time, Всемирное скоординированное время или время по Гринвичу, GMT). Вводимое значение, в котором явно указан часовой пояс, переводится в UTC с учетом смещения данного часового пояса. Если во входной строке не указан часовой пояс, подразумевается часовой пояс, заданный системным параметром `TimeZone`, и время также пересчитывается в UTC со смещением `timezone`.

Когда значение `timestamp with time zone` выводится, оно всегда преобразуется из UTC в текущий часовой пояс `timezone` и отображается как локальное время. Чтобы получить

время для другого часового пояса, нужно либо изменить `timezone`, либо воспользоваться конструкцией `AT TIME ZONE`.

В преобразованиях между `timestamp without time zone` и `timestamp with time zone` обычно предполагается, что значение `timestamp without time zone` содержит местное время (для часового пояса `timezone`). Другой часовой пояс для преобразования можно задать с помощью `AT TIME ZONE`.

#### 5.5.1.4 Специальные значения

PostgreSQL для удобства поддерживает несколько специальных значений даты/времени, перечисленных в таблице 5.13. Значения `infinity` и `-infinity` имеют особое представление в системе, и они отображаются в том же виде, тогда как другие варианты при чтении преобразуются в значения даты/времени. Чтобы использовать эти значения в качестве констант в командах SQL, их нужно заключать в апострофы.

Таблица 5.13 – Специальные значения даты/времени

Вводимая строка	Допустимые типы	Описание
<code>epoch</code>	<code>date, timestamp</code>	1970-01-01 00:00:00+00 (точка отсчета времени в Unix)
<code>infinity</code>	<code>date, timestamp</code>	время после максимальной допустимой даты
<code>-infinity</code>	<code>date, timestamp</code>	время до минимальной допустимой даты
<code>now</code>	<code>date, time, timestamp</code>	время начала текущей транзакции
<code>today</code>	<code>date, timestamp</code>	время начала текущих суток (00:00)
<code>tomorrow</code>	<code>date, timestamp</code>	время начала следующих суток (00:00)
<code>yesterday</code>	<code>date, timestamp</code>	время начала предыдущих суток (00:00)
<code>allballs</code>	<code>time</code>	00:00:00.00 UTC

Для получения текущей даты/времени соответствующего типа можно также использовать следующие SQL-совместимые функции: `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME` и `LOCALTIMESTAMP`. Во входных строках эти SQL-функции не распознаются.



Входные значения `now`, `today`, `tomorrow` и `yesterday` вполне корректно работают в интерактивных SQL-командах, но, когда команды сохраняются для последующего выполнения, например, в подготовленных операторах, представлениях или определениях функций, их поведение может быть

неожиданным. Такая строка может преобразоваться в конкретное значение времени, которое затем будет использоваться гораздо позже момента, когда оно было получено. В таких случаях следует использовать одну из SQL-функций. CURRENT\_DATE + 1 будет работать надежнее, чем 'tomorrow'::date.

### 5.5.2. Вывод даты/времени

В качестве выходного формата типов даты/времени можно использовать один из четырех стилей: ISO 8601, SQL (Ingres), традиционный формат POSTGRES (формат date в Unix) или German. По умолчанию выбран формат ISO. (Стандарт SQL требует, чтобы использовался именно ISO 8601. Другой формат называется «SQL» исключительно по историческим причинам.) Примеры всех стилей вывода перечислены в таблице 5.14. Вообще со значениями типов date и time выводилась бы только часть даты или времени из показанных примеров, но со стилем POSTGRES значение даты без времени выводится в формате ISO.

Таблица 5.14 – Стили вывода даты/время

Стиль	Описание	Пример
ISO	ISO 8601, стандарт SQL	1997-12-17 07:37:16-08
SQL	традиционный стиль	12/17/1997 07:37:16.00 PST
Postgres	изначальный стиль	Wed Dec 17 07:37:16 1997 PST
German	региональный стиль	17.12.1997 07:37:16.00 PST



ISO 8601 указывает, что дата должна отделяться от времени буквой Т в верхнем регистре. PostgreSQL принимает этот формат при вводе, но при выводе вставляет вместо Т пробел, как показано выше. Это сделано для улучшения читаемости и для совместимости с RFC 3339 и другими СУБД.

В стилях SQL и POSTGRES день выводится перед месяцем, если установлен порядок DMY, а в противном случае месяц выводится перед днем. Соответствующие примеры показаны в таблице 5.15.

Таблица 5.15 – Соглашения о порядке компонентов даты

Параметр datestyle	Порядок при вводе	Пример вывода
SQL, DMY	<i>день/месяц/год</i>	17/12/1997 15:37:16.00 CET
SQL, MDY	месяц/день/год	12/17/1997 07:37:16.00 PST
Postgres, DMY	день/месяц/год	Wed 17 Dec 07:37:16 1997 PST

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

В формате ISO часовой пояс всегда отображается в виде числового смещения со знаком относительно всемирного координированного времени (UTC); для зон к востоку от Гринвича знак смещения положительный. Смещение будет отображаться как hh (только часы), если оно задается целым числом часов, как hh:mm, если оно задается целым числом минут, или как hh:mm:ss. (Третий вариант невозможен в современных стандартах часовых поясов, но он может понадобиться при работе с отметками времени, предшествующими принятию стандартизированных часовых поясов.) В других форматах даты часовой пояс отображается как буквенное сокращение, если оно принято в текущем поясе. В противном случае он отображается как числовое смещение со знаком в стандартном формате ISO 8601 (hh или hhmm).

Стиль даты/времени пользователь может выбрать с помощью команды SET datestyle, параметра DateStyle в файле конфигурации postgresql.conf или переменной окружения PGDATESTYLE на сервере или клиенте.

Для большей гибкости при форматировании выводимой даты/времени можно использовать функцию to\_char.

### 5.5.3. Часовые пояса

Часовые пояса и правила их применения определяются не только по географическим, но и по политическим соображениям. Часовые пояса во всем мире были более-менее стандартизированы в начале прошлого века, но они продолжают претерпевать изменения, в частности это касается перехода на летнее время. Для расчета времени в прошлом PostgreSQL получает исторические сведения о правилах часовых поясов из распространенной базы данных IANA (Olson). Для будущего времени предполагается, что в заданном часовом поясе будут продолжать действовать последние принятые правила.

PostgreSQL стремится к совместимости со стандартом SQL в наиболее типичных случаях. Однако стандарт SQL допускает некоторые странности при смешивании типов даты и времени. Две очевидные проблемы:

- Хотя для типа date часовой пояс указать нельзя, это можно сделать для типа time. В реальности это не очень полезно, так как без даты нельзя точно определить смещение при переходе на летнее время.



- По умолчанию часовой пояс задается постоянным смещением от UTC. Это также не позволяет учесть летнее время при арифметических операциях с датами, пересекающими границы летнего времени.

Поэтому рекомендуется использовать часовой пояс с типами, включающими и время, и дату. Не рекомендуется использовать тип `time with time zone` (хотя PostgreSQL поддерживает его для старых приложений и совместимости со стандартом SQL). Для типов, включающих только дату или только время, в PostgreSQL предполагается местный часовой пояс.

Все значения даты и времени с часовым поясом представляются внутри в UTC, а при передаче клиентскому приложению они переводятся в местное время, при этом часовой пояс по умолчанию определяется параметром конфигурации `TimeZone`.

PostgreSQL позволяет задать часовой пояс тремя способами:

- Полное название часового пояса, например, `America/New_York`. Все допустимые названия перечислены в представлении `pg_timezone_names`. Определения часовых поясов PostgreSQL берет из широко распространенной базы IANA, так что имена часовых поясов PostgreSQL будут воспринимать и другие приложения.

- Аббревиатура часового пояса, например, `PST`. Такое определение просто задает смещение от UTC, в отличие от полных названий поясов, которые кроме того подразумевают и правила перехода на летнее время. Распознаваемые аббревиатуры перечислены в представлении `pg_timezone_abbrevs`. Аббревиатуры можно использовать во вводимых значениях даты/времени и в операторе `AT TIME ZONE`, но не в параметрах конфигурации `TimeZone` и `log_timezone`.

- Помимо аббревиатур и названий часовых поясов PostgreSQL принимает указания часовых поясов в стиле POSIX. Этот вариант обычно менее предпочтителен, чем использование именованного часового пояса, но он может быть единственным возможным, если для нужного часового пояса нет записи в базе данных IANA.

Вкратце различие между аббревиатурами и полными названиями заключается в следующем: аббревиатуры представляют определенный сдвиг от UTC, а полное название подразумевает еще местное правило по переходу на летнее время, то есть, возможно, два сдвига от UTC. Например, `2014-06-04 12:00 America/New_York` представляет полдень по

местному времени в Нью-Йорк, что для данного дня было бы летним восточным временем (EDT или UTC-4). Так что 2014-06-04 12:00 EDT обозначает тот же момент времени. Но 2014-06-04 12:00 EST задает стандартное восточное время (UTC-5), не зависящее от того, действовало ли летнее время в этот день.

В некоторых юрисдикциях одна и та же аббревиатура часового пояса означала разные сдвиги UTC в разное время; аббревиатура московского времени MSK несколько лет означала UTC+3, а затем стала означать UTC+4. PostgreSQL обрабатывает такие аббревиатуры в соответствии с их значениями на заданную дату, но, как и с примером выше EST, это не обязательно будет соответствовать местному гражданскому времени в этот день.

Независимо от формы, регистр в названиях и аббревиатурах часовых поясов не важен.

Ни названия, ни аббревиатуры часовых поясов, не зашиты в самом сервере; они считываются из файлов конфигурации, находящихся в путях `.../share/timezone/` и `.../share/timezonesets/` относительно каталога установки.

Параметр конфигурации `TimeZone` можно установить в `postgresql.conf` или любым другим стандартным способом. Часовой пояс может быть также определен следующими специальными способами:

- Часовой пояс для текущего сеанса можно установить с помощью SQL-команды `SET TIME ZONE`. Это альтернативная запись команды `SET TIMEZONE TO`, более соответствующая SQL-стандарту.
- Если установлена переменная окружения `PGTZ`, клиенты `libpq` используют ее значение, выполняя при подключении к серверу команду `SET TIME ZONE`.

#### 5.5.4. Ввод интервалов

Значения типа `interval` могут быть записаны в следующей расширенной форме:

[@] количество единица [количество единица...] [направление]

где количество — это число (возможно, со знаком); единица — одно из значений: `microsecond`, `millisecond`, `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, `millennium` (которые обозначают соответственно микросекунды, миллисекунды, секунды, минуты, часы, дни, недели, месяцы, годы, десятилетия, века и тысячелетия), либо эти же слова во множественном числе, либо их сокращения; направление может принимать

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

значение ago (назад) или быть пустым. Знак @ является необязательным. Все заданные величины различных единиц суммируются вместе с учетом знака чисел. Указание ago меняет знак всех полей на противоположный. Этот синтаксис также используется при выводе интервала, если параметр IntervalStyle имеет значение postgres\_verbose.

Количество дней, часов, минут и секунд можно определить, не указывая явно соответствующие единицы. Например, запись '1 12:59:10' равнозначна '1 day 12 hours 59 min 10 sec'. Сочетание года и месяца также можно записать через минус; '200-10' означает то, же что и '200 years 10 months'.

Интервалы можно также записывать в виде, определенном в ISO 8601, либо в «формате с кодами», в разделе 4.4.3.2 этого стандарта, либо в «альтернативном формате», описанном в разделе 4.4.3.3. Формат с кодами выглядит так:

P количество единица [ количество единица ... ] [ T [ количество единица ... ] ]

Строка должна начинаться с символа P и может включать также T перед временем суток. Допустимые коды единиц перечислены в таблице 5.16. Коды единиц можно опустить или указать в любом порядке, но компоненты времени суток должны идти после символа T. В частности, значение кода M зависит от того, располагается ли он до или после T.

Таблица 5.16 – Коды единиц временных интервалов ISO 8601

Код	Значение
Y	годы
M	месяцы (в дате)
W	недели
D	дни
H	часы
M	минуты (во времени)
S	секунды

В альтернативном формате:

P [ год-месяц-день ] [ T часы:минуты:секунды ]

строка должна начинаться с P, а T разделяет компоненты даты и времени. Значения выражаются числами так же, как и в датах ISO 8601.

При записи интервальной константы с указанием полей или присваивании столбцу типа `interval` строки с полями, интерпретация непомеченных величин зависит от полей. `INTERVAL '1' YEAR` воспринимается как 1 год, а `INTERVAL '1'` — как 1 секунда. Кроме того, значения «справа» от меньшего значащего поля, заданного в определении полей, просто отбрасываются. В записи `INTERVAL '1 day 2:03:04' HOUR TO MINUTE` будут отброшены секунды, но не день.

Согласно стандарту SQL, все компоненты значения `interval` должны быть одного знака, и ведущий минус применяется ко всем компонентам; минус в записи `'-1 2:03:04'` применяется и к дню, и к часам/минутам/секундам. PostgreSQL позволяет задавать для разных компонентов разные знаки и традиционно обрабатывает знак каждого компонента в текстовом представлении отдельно от других, так что в данном случае часы/минуты/секунды будут считаться положительными. Если параметр `IntervalStyle` имеет значение `sql_standard`, ведущий знак применяется ко всем компонентам (но только если они не содержат знаки явно). В противном случае действуют традиционные правила PostgreSQL. Во избежание неоднозначности рекомендуется добавлять знак к каждому компоненту с отрицательным значением.

Значения полей могут содержать дробную часть: `'1.5 weeks'` или `'01: 02: 03.45'`. Однако поскольку интервал содержит только три целых единицы (месяцы, дни, микросекунды), дробные единицы должны быть разделены на более мелкие единицы. Дробные части единиц, превышающих месяцы, усекаются до целого числа месяцев, `'1.5 years'` преобразуется в `'1 year 6 mons'`. Дробные части недель и дней пересчитываются в целое число дней и микросекунд, из расчета, что в месяце 30 дней, а в сутках – 24 часа, `'1.75 months'` становится `1 mon 22 days 12:00:00`. На выходе только секунды могут иметь дробную часть.

В таблице 5.17 показано несколько примеров допустимых вводимых значений типа `interval`.

Таблица 5.17 – Ввод интервалов

Пример		Описание	
1-2		Стандартный формат SQL: 1 год и 2 месяца	
3 4:05:06		Стандартный формат SQL: 3 дня 4 часа 5 минут 6 секунд	
1 year 2 months 3 days 4 hours 5 minutes 6 seconds		Традиционный формат Postgres: 1 год 2 месяца 3 дня 4 часа 5 минут 6 секунд	
№ изменения: _____		Подпись отв. лица: _____	Дата внесения изм: _____

Пример	Описание
P1Y2M3DT4H5M6S	«Формат с кодами» ISO 8601: то же значение, что и выше
P0001-02-03T04:05:06	«Альтернативный формат» ISO 8601: то же значение, что и выше

Тип `interval` представлен внутри в виде отдельных значений месяцев, дней и микросекунд. Это объясняется тем, что число дней в месяце может быть разным, а в сутках может быть и 23, и 25 часов в дни перехода на летнее/зимнее время. Значения месяцев и дней представлены целыми числами, а в микросекундах могут представляться секунды с дробной частью. Так как интервалы обычно получаются из строковых констант или при вычитании типов `timestamp`, этот способ хранения эффективен в большинстве случаев, но может давать неожиданные результаты:

```
SELECT EXTRACT(hours from '80 minutes'::interval);
date_part
-----
1

SELECT EXTRACT(days from '80 hours'::interval);
date_part
-----
0
```

Для корректировки числа дней и часов, когда они выходят за обычные границы, есть специальные функции `justify_days` и `justify_hours`.

### 5.5.5. Вывод интервалов

Формат вывода типа `interval` может определяться одним из четырех стилей: `sql_standard`, `postgres`, `postgres_verbose` и `iso_8601`. Выбрать нужный стиль позволяет команда `SET intervalstyle` (по умолчанию выбран `postgres`). Примеры форматов разных стилей показаны в таблице 5.18.

Стиль `sql_standard` выдает результат, соответствующий стандарту SQL, если значение интервала удовлетворяет ограничениям стандарта (и содержит либо только год и месяц, либо только день и время, и при этом все его компоненты одного знака). В противном случае

выводится год-месяц, за которым идет дата-время, а в компоненты для однозначности явно добавляются знаки.

Вывод в стиле postgres соответствует формату, который был принят в PostgreSQL, когда параметр DateStyle имел значение ISO.

Вывод в стиле postgres\_verbose соответствует формату, который был принят в PostgreSQL, когда значением параметром DateStyle было не ISO.

Вывод в стиле iso\_8601 соответствует «формату с кодами» описанному в разделе 4.4.3.2 формата ISO 8601.

Таблица 5.18 – Примеры стилей вывода интервалов

Стиль	Интервал год-месяц	Интервал день-время	Смешанный интервал
sql_standard	1-2	3 4:05:06	-1-2 +3 -4:05:06
postgres	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
postgres_verbose	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
iso_8601	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

## 5.6. Логический тип

В PostgreSQL есть стандартный SQL-тип boolean. Тип boolean может иметь следующие состояния: «true», «false» и третье состояние, «unknown», которое представляется SQL-значением NULL.

Таблица 5.19 – Логический тип данных

Имя	Размер	Описание
boolean	1 байт	состояние: истина или ложь

Логические константы могут представляться в SQL-запросах следующими ключевыми словами SQL: TRUE, FALSE и NULL.

Функция ввода данных типа boolean воспринимает следующие строковые представления состояния «true»:

```
true
yes
on
```

1

и следующие представления состояния «false»:

```
false  
no  
off  
0
```

Также воспринимаются уникальные префиксы этих строк, t или n. Регистр символов не имеет значения, а пробельные символы в начале и в конце строки игнорируются.

Ключевые слова TRUE и FALSE являются предпочтительными (соответствующими стандарту SQL) для записи логических констант в SQL-запросах.

При анализе запроса TRUE и FALSE автоматически считаются значениями типа boolean, но для NULL это не так, потому что ему может соответствовать любой тип. Поэтому в некоторых контекстах может потребоваться привести NULL к типу boolean явно, NULL::boolean. С другой стороны, приведение строковой константы к логическому типу можно опустить в тех контекстах, где анализатор запроса может понять, что буквальное значение должно иметь тип boolean.

## 5.7. Типы перечислений

Типы перечислений (enum) определяют статический упорядоченный набор значений, так же, как и типы enum, существующие в ряде языков программирования. В качестве перечисления можно привести дни недели или набор состояний.

### 5.7.1. Объявление перечислений

Тип перечислений создается с помощью команды CREATE TYPE,

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

Созданные типы enum можно использовать в определениях таблиц и функций, как и любые другие:

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

```
CREATE TABLE person (  
    name text,  
    current_mood mood  
);  
  
INSERT INTO person VALUES ('Moe', 'happy');  
  
SELECT * FROM person WHERE current_mood = 'happy';  
  
name | current_mood  
-----+-----  
Moe  | happy  
(1 row)
```

### 5.7.2. Порядок

Порядок значений в перечислении определяется последовательностью, в которой были указаны значения при создании типа. Перечисления поддерживаются всеми стандартными операторами сравнения и связанными агрегатными функциями.

```
INSERT INTO person VALUES ('Larry', 'sad');  
INSERT INTO person VALUES ('Curly', 'ok');  
  
SELECT * FROM person WHERE current_mood > 'sad';  
  
name  | current_mood  
-----+-----  
Moe   | happy  
Curly | ok  
(2 rows)  
  
SELECT * FROM person WHERE current_mood > 'sad' ORDER BY  
current_mood;  
  
name  | current_mood  
-----+-----  
Curly | ok  
Moe    | happy  
(2 rows)  
  
SELECT name  
  
FROM person
```



```
WHERE current_mood = (SELECT MIN(current_mood) FROM person);

name
-----
Larry
(1 row)
```

### 5.7.3. Безопасность типа

Все типы перечислений считаются уникальными и поэтому значения разных типов нельзя сравнивать. Пример:

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy',
'ecstatic');

CREATE TABLE holidays (
    num_weeks integer,
    happiness happiness
);

INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very
happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8,
'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ОШИБКА: неверное значение для перечисления happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
    WHERE person.current_mood = holidays.happiness;
ОШИБКА: оператор не существует: mood = happiness
```

Если нужно сделать что-то подобное, можно либо реализовать собственный оператор, либо явно преобразовать типы в запросе:

```
SELECT person.name, holidays.num_weeks FROM person, holidays
    WHERE person.current_mood::text = holidays.happiness::text;

name | num_weeks
-----+-----
```

```

Мое | 4
(1 row)

```

#### 5.7.4. Тонкости реализации

В метках значений регистр имеет значение, т. е. 'happy' и 'HAPPY' — не одно и то же. Также в метках имеют значение пробелы.

Хотя типы-перечисления предназначены прежде всего для статических наборов значений, имеется возможность добавлять новые значения в существующий тип-перечисление и переименовывать значения. Однако удалять существующие значения из перечисления, а также изменять их порядок, нельзя — для получения нужного результата придется удалить и воссоздать это перечисление.

Значение enum занимает на диске 4 байта. Длина текстовой метки значения ограничена параметром компиляции NAMEDATALEN; в стандартных сборках PostgreSQL он ограничивает длину 63 байтами.

Сопоставления внутренних значений enum с текстовыми метками хранятся в системном каталоге pg\_enum. Он может быть полезен в ряде случаев.

### 5.8. Геометрические типы

Геометрические типы данных представляют объекты в двумерном пространстве. Все существующие в PostgreSQL геометрические типы перечислены в таблице 5.20.

Таблица 5.20 – Геометрические типы

Имя	Размер	Описание	Представление
point	16 байт	Точка на плоскости	(x,y)
line	32 байта	Бесконечная прямая	{A,B,C}
lseg	32 байта	Отрезок	((x1,y1),(x2,y2))
box	32 байта	Прямоугольник	((x1,y1),(x2,y2))
path	16+16n байт	Закрытый путь (подобный многоугольнику)	((x1,y1),...)
path	16+16n байт	Открытый путь	[(x1,y1),...]
polygon	40+16n байт	Многоугольник (подобный закрытому пути)	((x1,y1),...)
circle	24 байта	Окружность	<(x,y),r> (центр окружности и радиус)

Для выполнения различных геометрических операций, в частности масштабирования, вращения и определения пересечений, PostgreSQL предлагает богатый набор функций и операторов.

### 5.8.1. Точки

Точки — это основной элемент, на базе которого строятся все остальные геометрические типы. Значения типа `point` записываются в одном из двух форматов:

```
( x , y )  
x , y
```

где  $x$  и  $y$  — координаты точки на плоскости, выраженные числами с плавающей точкой.

Выводятся точки в первом формате.

### 5.8.2. Прямые

Прямые представляются линейным уравнением  $Ax + By + C = 0$ , где  $A$  и  $B$  не равны 0. Значения типа `line` вводятся и выводятся в следующем виде:

```
{ A, B, C }
```

Кроме того, для ввода может использоваться любая из этих форм:

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

где  $(x1,y1)$  и  $(x2,y2)$  — две различные точки на данной прямой.

### 5.8.3. Отрезки

Отрезок представляется парой точек, определяющих концы отрезка. Значения типа `lseg` записываются в одной из следующих форм:

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

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

где (x1,y1) и (x2,y2) — концы отрезка.

Выводятся отрезки в первом формате.

#### 5.8.4. Прямоугольники

Прямоугольник представляется двумя точками, находящимися в противоположных его углах. Значения типа box записываются в одной из следующих форм:

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

где (x1,y1) и (x2,y2) — противоположные углы прямоугольника.

Выводятся прямоугольники во второй форме.

Во вводимом значении могут быть указаны любые два противоположных угла, но затем они будут упорядочены, так что внутри сохранятся правый верхний и левый нижний углы, в таком порядке.

#### 5.8.5. Пути

Пути представляют собой списки соединенных точек. Пути могут быть закрытыми, когда подразумевается, что первая и последняя точка в списке соединены, или открытыми, в противном случае.

Значения типа path записываются в одной из следующих форм:

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]  
( ( x1 , y1 ) , ... , ( xn , yn ) )  
( x1 , y1 ) , ... , ( xn , yn )  
( x1 , y1      , ... ,      xn , yn )  
x1 , y1      , ... ,      xn , yn
```

где точки задают узлы сегментов, составляющих путь. Квадратные скобки ([]) указывают, что путь открытый, а круглые (()) — закрытый. Когда внешние скобки опускаются, как в показанных выше последних трех формах, считается, что путь закрытый.

Пути выводятся в первой или второй форме, в соответствии с типом.

### 5.8.6. Многоугольники

Многоугольники представляются списками точек (вершин). Многоугольники похожи на закрытые пути, но хранятся в другом виде и для работы с ними предназначен отдельный набор функций.

Значения типа `polygon` записываются в одной из следующих форм:

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

где точки задают узлы сегментов, образующих границу многоугольника.

Выводятся многоугольники в первом формате.

### 5.8.7. Окружности

Окружности задаются координатами центра и радиусом. Значения типа `circle` записываются в одном из следующих форматов:

```
< ( x , y ) , r >
( ( x , y ) , r )
( x , y ) , r
x , y , r
```

где (x,y) — центр окружности, а r — ее радиус.

Выводятся окружности в первом формате.

## 5.9. Типы, описывающие сетевые адреса

PostgreSQL предлагает типы данных для хранения адресов IPv4, IPv6 и MAC, показанные в таблице 5.21. Для хранения сетевых адресов лучше использовать эти типы, а

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

не простые текстовые строки, так как PostgreSQL проверяет вводимые значения данных типов и предоставляет специализированные операторы и функции для работы с ними.

Таблица 5.21 – Типы, описывающие сетевые адреса

Имя	Размер	Описание
cidr	7 или 19 байт	Сети IPv4 и IPv6
inet	7 или 19 байт	Узлы и сети IPv4 и IPv6
macaddr	6 байт	MAC-адреса
macaddr8	8 байт	MAC-адреса (в формате EUI-64)

При сортировке типов `inet` и `cidr`, адреса IPv4 всегда идут до адресов IPv6, в том числе адреса IPv4, включенные в IPv6 или сопоставленные с ними, `::10.2.3.4` или `::ffff:10.4.3.2`.

### 5.9.1. `inet`

Тип `inet` содержит IPv4- или IPv6-адрес узла и может также содержать его подсеть, все в одном поле. Подсеть представляется числом бит, определяющих адрес сети в адресе узла (или «маску сети»). Если маска сети равна 32 для адреса IPv4, такое значение представляет не подсеть, а определенный узел. Адреса IPv6 имеют длину 128 бит, поэтому уникальный адрес узла задается с маской 128 бит.

Вводимые значения такого типа должны иметь формат `адрес/у`, где `адрес` — адрес IPv4 или IPv6, а `у` — число бит в маске сети. Если компонент `/у` опущен, маска сети считается равной 32 для IPv4 и 128 для IPv6, так что это значение будет представлять один узел. При выводе компонент `/у` опускается, если сетевой адрес определяет адрес одного узла.

### 5.9.2. `cidr`

Тип `cidr` содержит определение сети IPv4 или IPv6. Входные и выходные форматы соответствуют соглашениям CIDR (Classless Internet Domain Routing, Бесклассовая межсетевая адресация). Определение сети записывается в формате `адрес/у`, где `адрес` — минимальный адрес в сети, представленный в виде адреса IPv4 или IPv6, а `у` — число бит в маске сети. Если `у` не указывается, это значение вычисляется по старой классовой схеме нумерации сетей, но при этом оно может быть увеличено, чтобы в него вошли все байты введенного адреса. Если в сетевом адресе справа от маски сети окажутся биты со значением 1, он будет считаться ошибочным.

В таблице 5.22 показаны несколько примеров адресов.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Таблица 5.22 – Примеры допустимых значений типа cidr

Вводимое значение cidr	Выводимое значение cidr	abbrev(cidr)
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba/64
2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

### 5.9.3. Различия inet и cidr

Существенным различием типов данных inet и cidr является то, что inet принимает значения с ненулевыми битами справа от маски сети, а cidr — нет. Например, значение 192.168.0.1/24 является допустимым для типа inet, но не для cidr.



Если не устраивает выходной формат значений inet или cidr, попробуйте функции host, text и abbrev.

### 5.9.4. macaddr

Тип macaddr предназначен для хранения MAC-адреса, примером которого является адрес сетевой платы Ethernet (хотя MAC-адреса применяются и для других целей). Вводимые значения могут задаваться в следующих форматах:

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
```

```
'0800.2b01.0203'
'0800-2b01-0203'
'08002b010203'
```

Все эти примеры определяют один и тот же адрес. Шестнадцатеричные цифры от а до f могут быть и в нижнем, и в верхнем регистре. Выводятся MAC-адреса всегда в первой форме.

Стандарт IEEE 802-2001 считает канонической формой MAC-адресов вторую (с минусами), а в первой (с двоеточиями) предполагает обратный порядок бит, так что 08-00-2b-01-02-03 = 01:00:4D:08:04:0C. В настоящее время этому соглашению практически никто не следует, и уместно оно было только для устаревших сетевых протоколов (таких как Token Ring). PostgreSQL не меняет порядок бит и во всех принимаемых форматах подразумевается традиционный порядок LSB.

Последние пять входных форматов не описаны ни в одном стандарте.

### 5.9.5. macaddr8

Тип `macaddr8` хранит MAC-адреса в формате EUI-64, применяющийся для аппаратных адресов плат Ethernet (хотя MAC-адреса используются и для других целей). Этот тип может принять и 6-байтовые, и 8-байтовые адреса MAC, и сохраняет их в 8 байтах. MAC-адреса, заданные в 6-байтовом формате, хранятся в формате 8 байт, а 4-ый и 5-ый байт содержат FF и FE соответственно. Для IPv6 используется модифицированный формат EUI-64, в котором 7-ой бит должен быть установлен в 1 после преобразования из EUI-48. Для выполнения этого изменения предоставляется функция `macaddr8_set7bit`. Этот тип принимает любые строки, состоящие из пар шестнадцатеричных цифр (выровненных по границам байт), которые могут согласованно разделяться одинаковыми символами ':', '-' или ' '. Шестнадцатеричных цифр должно быть либо 16 (для 8 байт), либо 12 (для 6 байт). Начальные и конечные пробелы игнорируются. Ниже показаны примеры допустимых входных строк:

```
'08:00:2b:01:02:03:04:05'
'08-00-2b-01-02-03-04-05'
'08002b:0102030405'
```



```
'08002b-0102030405'
'0800.2b01.0203.0405'
'0800-2b01-0203-0405'
'08002b01:02030405'
'08002b0102030405'
```

Во всех этих примерах задается один и тот же адрес. Для цифр с a по f принимаются буквы и в верхнем, и в нижнем регистре. Вывод всегда представляется в первом из показанных форматов.

Последние шесть входных форматов из показанных выше не являются стандартизированными.

Чтобы преобразовать традиционный 48-битный MAC-адрес в формате EUI-48 в модифицированный формат EUI-64 для включения в состав адреса IPv6 в качестве адреса узла, используйте функцию `macaddr8_set7bit` следующим образом:

```
SELECT macaddr8_set7bit('08:00:2b:01:02:03');

      macaddr8_set7bit
-----
0a:00:2b:ff:fe:01:02:03
(1 row)
```

## 5.10. Битовые строки

Битовые строки представляют собой последовательности из 1 и 0. Их можно использовать для хранения или отображения битовых масок. В SQL есть два битовых типа: `bit(n)` и `bit varying(n)`, где `n` — положительное целое число.

Длина значения типа `bit` должна в точности равняться `n`; при попытке сохранить данные длиннее или короче произойдет ошибка. Данные типа `bit varying` могут иметь переменную длину, но не превышающую `n`; строки большей длины не будут приняты. Запись `bit` без указания длины равнозначна записи `bit(1)`, тогда как `bit varying` без указания длины подразумевает строку неограниченной длины.



При попытке привести значение битовой строки к типу `bit(n)`, оно будет усечено или дополнено нулями справа до длины ровно `n` бит, ошибки при этом не будет.

Подобным образом, если явно привести значение битовой строки к типу `bit varying(n)`, она будет усечена справа, если ее длина превышает `n` бит.

Для хранения битовой строки используется по 1 байту для каждой группы из 8 бит, плюс 5 или 8 байт дополнительно в зависимости от длины строки (но длинные строки могут быть сжаты или вынесены отдельно применительно к символьным строкам).

## 5.11. Типы, предназначенные для текстового поиска

PostgreSQL предоставляет два типа данных для поддержки полнотекстового поиска. Текстовым поиском называется операция анализа набора документов с текстом на естественном языке, в результате которой находятся фрагменты, наиболее соответствующие запросу. Тип `tsvector` представляет документ в виде, оптимизированном для текстового поиска, а `tsquery` представляет запрос текстового поиска в подобном виде.

### 5.11.1. `tsvector`

Значение типа `tsvector` содержит отсортированный список неповторяющихся лексем, т. е. слов, нормализованных так, что все словоформы сводятся к одной. Сортировка и исключение повторяющихся слов производится автоматически при вводе значения, как показано в этом примере:

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

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

Для представления в виде лексем пробелов или знаков препинания их нужно заключить в апострофы:

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

               tsvector
-----
' ' 'contains' 'lexeme' 'spaces' 'the'
```

При этом включаемый апостроф или обратную косую черту нужно продублировать:

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

               tsvector
```

```
-----  
'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

Также для лексем можно указать их целочисленные позиции:

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

tsvector

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

Позиция обычно указывает положение исходного слова в документе. Информация о расположении слов затем может использоваться для оценки близости. Позиция может задаваться числом от 1 до 16383; большие значения просто заменяются на 16383. Если для одной лексемы дважды указывается одно положение, такое повторение отбрасывается.

Лексемам, для которых заданы позиции, также можно назначить вес, выраженный буквами A, B, C или D. Вес D подразумевается по умолчанию и поэтому он не показывается при выводе:

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

tsvector

```
-----  
'a':1A 'cat':5 'fat':2B,4C
```

Веса обычно применяются для отражения структуры документа для придания особого значения словам в заголовке по сравнению со словами в обычном тексте. Назначенным весам можно сопоставить числовые приоритеты в функциях ранжирования результатов.

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

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

tsvector

```
-----  
'Fat' 'Rats' 'The'
```

Для большинства англоязычных приложений приведенные выше слова будут считаться ненормализованными, но для `tsvector` это не важно. Поэтому исходный документ обычно следует обработать функцией `to_tsvector`, нормализующей слова для поиска:

```
SELECT to_tsvector('english', 'The Fat Rats');

   to_tsvector
-----
'fat':2 'rat':3
```

### 5.11.2. tsquery

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

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

Лексемам в `tsquery` можно дополнительно сопоставить 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'
```

Кроме того, в лексемах `tsquery` можно использовать знак `*` для поиска по префиксу:

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

   tsquery
-----
'super':*
```

Этот запрос найдет все слова в `tsvector`, начинающиеся с приставки «super».

Апострофы в лексемах этого типа можно использовать так же, как и в лексемах в `tsvector`; и так же, как и для типа `tsvector`, необходимая нормализация слова должна выполняться до приведения значения к типу `tsquery`. Для такой нормализации удобно использовать функцию `to_tsquery`:

```
SELECT to_tsquery('Fat:ab & Cats');

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

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

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

?column?
-----
t
```

так как `postgres` преобразуется стеммером в `postgr`:

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

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

и эта приставка находится в преобразованной форме слова `postgraduate`.

## 5.12. Тип UUID

Тип данных `uuid` сохраняет универсальные уникальные идентификаторы (Universally Unique Identifiers, UUID), определенные в RFC 4122, ISO/IEC 9834-8:2005 и связанных стандартах. (В некоторых системах это называется GUID, глобальным уникальным идентификатором.) Этот идентификатор представляет собой 128-битное значение, генерируемое специальным алгоритмом, практически гарантирующим, что этим же алгоритмом оно не будет получено больше нигде в мире. Таким образом, эти идентификаторы будут уникальными и в распределенных системах, а не только в единственной базе данных, как значения генераторов последовательностей.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

UUID записывается в виде последовательности шестнадцатеричных цифр в нижнем регистре, разделенных знаками минуса на несколько групп, в таком порядке: группа из 8 цифр, за ней три группы из 4 цифр и, наконец, группа из 12 цифр, что в сумме составляет 32 цифры и представляет 128 бит. Пример UUID в этом стандартном виде:

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

PostgreSQL также принимает альтернативные варианты: цифры в верхнем регистре, стандартную запись, заключенную в фигурные скобки, запись без минусов или с минусами, разделяющими любые группы из четырех цифр.

```
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.

### 5.13. Тип XML

Тип xml предназначен для хранения XML-данных. Его преимущество по сравнению с обычным типом text в том, что он проверяет вводимые значения на допустимость по правилам XML и для работы с ним есть типобезопасные функции. Для использования этого типа дистрибутив должен быть скомпилирован в конфигурации configure --with-libxml.

Тип xml может сохранять правильно оформленные «документы», в соответствии со стандартом XML, а также фрагменты «содержимого», определяемые как менее ограниченные «узлы документа» в модели данных XQuery и XPath. Другими словами, это означает, что во фрагментах содержимого может быть несколько элементов верхнего уровня или текстовых узлов. Является ли некоторое значение типа xml полным документом или фрагментом содержимого, позволяет определить выражение xml-значение IS DOCUMENT.

#### 5.13.1. Создание XML-значений

Чтобы получить значение типа xml из текстовой строки, используйте функцию xmlparse:

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

```
XMLPARSE ( { DOCUMENT | CONTENT } value)
```

Примеры:

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

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

Хотя в стандарте SQL описан только один способ преобразования текстовых строк в XML-значения, также допустим специфический синтаксис PostgreSQL:

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

Тип xml не проверяет вводимые значения по схеме DTD (Document Type Declaration, объявления типа документа), даже если в них присутствуют ссылка на DTD. В настоящее время в PostgreSQL также нет встроенной поддержки других разновидностей схем.

Обратная операция, получение текстовой строки из xml, выполняется с помощью функции xmlserialize:

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } значение AS тип )
```

Здесь допустимый тип — character, character varying или text (или их псевдонимы). И в данном случае стандарт SQL предусматривает только один способ преобразования xml в тип текстовых строк, но PostgreSQL позволяет просто привести значение к нужному типу.

При преобразовании текстовой строки в тип xml или наоборот без использования функций XMLPARSE и XMLSERIALIZE, выбор режима DOCUMENT или CONTENT определяется параметром конфигурации сеанса «XML option», установить который можно следующей стандартной командой:

```
SET XML OPTION { DOCUMENT | CONTENT };
```

или такой командой в духе PostgreSQL:

```
SET xmloption TO { DOCUMENT | CONTENT };
```

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

### 5.13.2. Обработка кодировки

Если на стороне сервера и клиента и в XML-данных используются разные кодировки символов, с этим могут возникать проблемы. Когда запросы передаются на сервер, а их результаты возвращаются клиенту в обычном текстовом режиме, PostgreSQL преобразует все передаваемые текстовые данные в кодировку для соответствующей стороны. В том числе это происходит и со строковыми представлениями XML-данных, подобными тем, что показаны в предыдущих примерах. Обычно это означает, что объявления кодировки, содержащиеся в XML-данных, могут не соответствовать действительности, когда текстовая строка преобразуется из одной кодировки в другую при передаче данных между клиентом и сервером, так как подобные включенные в данные объявления не будут изменены автоматически. Для решения этой проблемы объявления кодировки, содержащиеся в текстовых строках, вводимых в тип xml, просто игнорируются и предполагается, что XML-содержимое представлено в текущей кодировке сервера. Как следствие, для правильной обработки таких строк с XML-данными клиент должен передавать их в своей текущей кодировке. Для сервера не важно, будет ли клиент для этого преобразовывать документы в свою кодировку, или изменит ее, прежде чем передавать ему данные. При выводе значения типа xml не содержат объявления кодировки, а клиент должен предполагать, что все данные поступают в его текущей кодировке.

Если параметры запроса передаются на сервер, и он возвращает результаты клиенту в двоичном режиме, кодировка символов не преобразуется, так что возникает другая ситуация. В этом случае объявление кодировки в XML принимается во внимание, а если его нет, то предполагается, что данные закодированы в UTF-8 (это соответствует стандарту XML). При выводе в данные будет добавлено объявление кодировки, выбранной на стороне клиента (но если это UTF-8, объявление будет опущено).

Само собой, XML-данные в PostgreSQL будут обрабатываться гораздо эффективнее, когда и в XML-данных, и на стороне клиента, и на стороне сервера используется одна



кодировка. Так как внутри XML-данные представляются в UTF-8, оптимальный вариант, когда на сервере также выбрана кодировка UTF-8.



Некоторые XML-функции способны работать исключительно с ASCII-данными, если кодировка сервера не UTF-8. В частности, это известная особенность функций `xmltable()` и `xpath()`.

### 5.13.3. Обращение к XML-значениям

Тип `xml` отличается от других тем, что для него не определены никакие операторы сравнения, так как четко определенного и универсального алгоритма сравнения XML-данных не существует. Одно из следствий этого — нельзя отфильтровать строки таблицы, сравнив столбец `xml` с искомым значением. Поэтому обычно XML-значения должны дополняться отдельным ключевым полем.

Из-за отсутствия операторов сравнения для типа `xml`, для столбца этого типа также нельзя создать индекс. Поэтому, когда требуется быстрый поиск в XML данных, обойти это ограничение можно, приведя данные к типу текстовой строки и проиндексировав эти строки, либо проиндексировав выражение `XPath`. Конечно сам запрос при этом следует изменить, чтобы поиск выполнялся по индексированному выражению.

Для ускорения поиска в XML-данных также можно использовать функции полнотекстового поиска в PostgreSQL. Однако это требует определенной подготовки данных, что дистрибутив PostgreSQL пока не поддерживает.

### 5.14. Типы JSON

Типы JSON предназначены для хранения данных JSON (JavaScript Object Notation, запись объекта JavaScript) согласно стандарту RFC 7159. Такие данные можно хранить и в типе `text`, но типы JSON лучше тем, что проверяют, соответствует ли вводимое значение формату JSON. Для работы с ними есть также несколько специальных функций и операторов.

В PostgreSQL имеются два типа для хранения данных JSON: `json` и `jsonb`. Для реализации эффективного механизма запросов к этим типам данных в PostgreSQL также имеется тип `jsonpath`.

Типы данных `json` и `jsonb` принимают на вход почти одинаковые наборы значений, а отличаются они главным образом с точки зрения эффективности. Тип `json` сохраняет точную

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

копию введенного текста, которую функции обработки должны разбирать заново при каждом выполнении запроса, тогда как данные jsonb сохраняются в разобранном двоичном формате, что несколько замедляет ввод из-за преобразования, но значительно ускоряет обработку, не требуя многократного разбора текста. Кроме того, jsonb поддерживает индексацию, что тоже может быть очень полезно.

Так как тип json сохраняет точную копию введенного текста, он сохраняет семантически незначимые пробелы между элементами, а также порядок ключей в JSON-объектах. И если JSON-объект внутри содержит повторяющиеся ключи, этот тип сохранит все пары ключ/значение. (Функции обработки будут считать действительной последнюю пару.) Тип jsonb, напротив, не сохраняет пробелы, порядок ключей и значения с дублирующимися ключами. Если во входных данных оказываются дублирующиеся ключи, сохраняется только последнее значение.

Для большинства приложений предпочтительнее хранить данные JSON в типе jsonb.

В RFC 7159 говорится, что строки JSON должны быть представлены в кодировке UTF-8. Поэтому данные JSON не будут полностью соответствовать спецификации, если кодировка базы данных не UTF-8. При этом нельзя будет вставить в JSON символы, непредставимые в кодировке сервера, и наоборот, допустимыми будут символы, представимые в кодировке сервера, но не в UTF-8.

RFC 7159 разрешает включать в строки JSON спецпоследовательности Unicode в виде \uXXXX. В функцию ввода для типа json эти спецпоследовательности допускаются вне зависимости от кодировки базы данных, и проверяется только правильность их синтаксиса (за \u должны следовать четыре шестнадцатеричных цифры). Однако функция ввода для типа jsonb более строгая: она не допускает спецпоследовательности Unicode для символов, которые не могут быть представлены в кодировке базы. Тип jsonb также не принимает \u0000 (так как это значение не может быть представлено в типе text PostgreSQL) и требует, чтобы суррогатные пары Unicode использовались для представления символов вне основной многоязыковой плоскости (BMP) правильно. Корректные спецпоследовательности Unicode преобразуются для хранения в один соответствующий символ (это подразумевает сворачивание суррогатных пар в один символ).



Многие из функций обработки JSON преобразуют спецпоследовательности Unicode в обычные символы, поэтому могут выдавать подобные ошибки, даже если им на вход поступает тип `json`, а не `jsonb`. То, что функция ввода в тип `json` не производит этих проверок, можно считать историческим артефактом, хотя это и позволяет просто сохранять (но не обрабатывать) в JSON спецкоды Unicode в базе данных с кодировкой, в которой представленные таким образом символы отсутствуют.

При преобразовании вводимого текста JSON в тип `jsonb`, примитивные типы, описанные в RFC 7159, по сути отображаются в собственные типы PostgreSQL как показано в таблице 5.23. Таким образом, к содержимому типа `jsonb` предъявляются некоторые дополнительные требования, продиктованные ограничениями представления нижележащего типа данных, которые не распространяются ни на тип `json`, ни на формат JSON вообще. В частности, тип `jsonb` не принимает числа, выходящие за диапазон типа данных PostgreSQL `numeric`, тогда как с `json` такого ограничения нет. Такие ограничения, накладываемые реализацией, допускаются согласно RFC 7159. Однако на практике такие проблемы более вероятны в других реализациях, так как обычно примитивный тип JSON `number` представляется в виде числа с плавающей точкой двойной точности IEEE 754 (что RFC 7159 явно признает и допускает). При использовании JSON в качестве формата обмена данными с такими системами следует учитывать риски потери точности чисел, хранившихся в PostgreSQL.

И напротив, как показано в таблице, есть некоторые ограничения в формате ввода примитивных типов JSON, не актуальные для соответствующих типов PostgreSQL.

Таблица 5.23 – Примитивные типы JSON и соответствующие им типы PostgreSQL

Примитивный тип JSON	Тип PostgreSQL	Замечания
string	text	\u0000 не допускается как спецпоследовательность Unicode, представляющая символ, который отсутствует в кодировке базы
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

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

### 5.14.1. Проектирование документов JSON

Представлять данные в JSON можно гораздо более гибко, чем в традиционной реляционной модели данных, что очень привлекательно там, где нет жестких условий. И оба этих подхода вполне могут сосуществовать и дополнять друг друга в одном приложении. Однако даже для приложений, которым нужна максимальная гибкость, рекомендуется, чтобы документы JSON имели некоторую фиксированную структуру. Эта структура обычно не навязывается жестко (хотя можно декларативно диктовать некоторые бизнес-правила), но когда она предсказуема, становится гораздо проще писать запросы, которые извлекают полезные данные из набора «документов» (информации) в таблице.

Данные JSON, как и данные любых других типов, хранящиеся в таблицах, находятся под контролем механизма параллельного доступа. Хотя хранить большие документы вполне возможно, не забывайте, что при любом изменении устанавливается блокировка всей строки (на уровне строки). Поэтому для оптимизации блокировок транзакций, изменяющих данные, стоит ограничить размер документов JSON разумными пределами. В идеале каждый документ JSON должен собой представлять атомарный информационный блок, который, согласно бизнес-логике, нельзя разделить на меньшие, индивидуально изменяемые блоки.

### 5.14.2. Проверки на вхождение и существование jsonb

Проверка вхождения — важная особенность типа jsonb, не имеющая аналога для типа json. Эта проверка определяет, входит ли один документ jsonb в другой. В следующих примерах возвращается истинное значение (кроме упомянутых исключений):

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

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

-- Массив с правой стороны входит в массив слева:

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

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

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

-- Повторяющиеся элементы массива не имеют значения:

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

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

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

-- Массив справа не считается входящим в массив слева, хотя в последний и вложен подобный массив (команда выдает false):

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

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

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

-- Аналогично, это вхождением не считается (команда выдает false):

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

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

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

Общий принцип этой проверки в том, что входящий объект должен соответствовать объекту, содержащему его, по структуре и данным, возможно, после исключения из содержащего объекта лишних элементов массива или пар ключ/значение. Но помните, что порядок элементов массива для проверки на вхождение не имеет значения, а повторяющиеся элементы массива считаются только один раз.

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

-- В этот массив входит примитивное строковое значение:

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

-- Это исключение действует только в одну сторону -- здесь вхождения нет (команда выдает false):

```
SELECT '"bar"'::jsonb @> '["bar"]'::jsonb;
```

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

-- Строка существует в качестве элемента массива:

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

-- Строка существует в качестве ключа объекта:

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

-- Значения объектов не рассматриваются (команда выдает false):

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

-- Как и вхождение, существование определяется на верхнем уровне (команда выдает false):

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

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

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

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



Так как вхождение в JSON проверяется с учетом вложенности, правильно написанный запрос может заменить явную выборку внутренних объектов. Предположим, что есть столбец `doc`, содержащий объекты на верхнем уровне, и большинство этих объектов содержит поля `tags` с массивами вложенных объектов. Данный запрос найдет 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:

```
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.

### 5.14.3. Индексация jsonb

Для эффективного поиска ключей или пар ключ/значение в большом количестве документов `jsonb` можно успешно применять индексы GIN. Для этого предоставляются два «класса операторов» GIN, предлагающие выбор между производительностью и гибкостью.

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

```
CREATE INDEX idxgin ON api USING GIN (jdoc) ;
```

Дополнительный класс операторов GIN jsonb\_path\_ops поддерживает индексацию только для оператора @>. Пример создания индекса с этим классом операторов:

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

Приведен 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"
  ]
}
```

Документы сохраняются в таблице api, в столбце jdoc типа jsonb. Если по этому столбцу создается 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';
```

И все же, правильно применяя индексы выражений, в этом запросе можно задействовать индекс. Если запрос определенных элементов в ключе "tags" выполняется часто, вероятно стоит определить такой индекс:

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

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

Также индексы GIN поддерживают операторы @@ и @?, которые сопоставляют jsonpath с данными.

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @@
'$ .tags[*] == "qui"';

SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @?
'$ .tags[*] ? (@ == "qui")';
```

Индекс GIN извлекает из jsonpath конструкции следующего вида: цепочка\_обращения = константа. Цепочка обращения может включать указания обращения .ключ, [\*] и [индекс]. Класс операторов jsonb\_ops дополнительно поддерживает указания .\* и .\*\*.

Еще 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 `jdosc`. Такой индекс будет хранить копии всех ключей и значений в поле `jdosc`, тогда как индекс выражения из предыдущего примера хранит только данные внутри объекта с ключом `tags`. Хотя подход с простым индексом гораздо более гибкий (так как он поддерживает запросы по любому ключу), индексы конкретных выражений скорее всего будут меньше и быстрее, чем простые индексы.

Класс операторов `jsonb_path_ops` поддерживает только запросы с операторами `@>`, `@@` и `@?`, но он значительно производительнее класса по умолчанию `jsonb_ops`. Индекс `jsonb_path_ops` обычно гораздо меньше индекса `jsonb_ops` для тех же данных и более точен при поиске, особенно если запросы обращаются к ключам, часто встречающимся в данных. Таким образом, с ним операции поиска выполняются гораздо эффективнее, чем с классом операторов по умолчанию.

Техническое различие между GIN-индексами `jsonb_ops` и `jsonb_path_ops` состоит в том, что для первых создаются независимые элементы индекса для каждого ключа/значения в данных, тогда как для вторых создаются элементы только для значений. [7] По сути, каждый элемент индекса `jsonb_path_ops` представляет собой хеш значения и ключа(ей), приводящего к нему; при индексации `{"foo": {"bar": "baz"}}` будет создан один элемент индекса с хешем, рассчитанным по всем трем значениям: `foo`, `bar` и `baz`. Таким образом, проверка на вхождение этой структуры будет использовать крайне точный поиск по индексу, но определить, является ли `foo` ключом, с помощью такого индекса нельзя. С другой стороны, индекс `jsonb_ops` создаст три отдельных элемента индекса, представляющих `foo`, `bar` и `baz` по отдельности; для выполнения проверки на вхождение будут проверены строки таблицы, содержащие все эти три значения. Хотя GIN-индексы позволяют вычислить AND довольно эффективно, такой поиск все же будет менее точным и более медленным, чем равнозначный поиск с `jsonb_path_ops`, особенно если любое одно из этих трех значений содержится в большом количестве строк.

Недостаток класса `jsonb_path_ops` заключается в том, что он не учитывает в индексе структуры JSON, не содержащие никаких значений `{"a": {}}`. Для поиска по документам, содержащих такие структуры, потребуется выполнить полное сканирование индекса, что довольно долго, поэтому `jsonb_path_ops` не очень подходит для приложений, часто выполняющих такие запросы.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Тип `jsonb` также поддерживает индексы `btree` и `hash`. Они полезны только если требуется проверять равенство JSON-документов в целом. Порядок сортировки `btree` для типа `jsonb` редко имеет большое значение, но для полноты он приводится ниже:

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

Объект с  $n$  парами > Объект с  $n - 1$  парами

Массив с  $n$  элементами > Массив с  $n - 1$  элементами

Объекты с равным количеством пар сравниваются в таком порядке:

ключ-1, значение-1, ключ-2 ...

Ключи объектов сравниваются согласно порядку при хранении; в частности, из-за того, что короткие ключи хранятся перед длинными, результаты могут оказаться несколько не интуитивными:

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

Массивы с равным числом элементов упорядочиваются аналогично:

элемент-1, элемент-2 ...

Примитивные значения JSON сравниваются по тем же правилам сравнения, что и нижележащие типы данных PostgreSQL. Строки сравниваются с учетом порядка сортировки по умолчанию в текущей базе данных.

#### 5.14.4. Обращение по индексу к элементам `jsonb`

Тип данных `jsonb` поддерживает извлечение и изменение элементов в стиле обращения к элементам массива. Указывать на вложенные значения можно, задавая цепочку обращений к элементам, при этом будут действовать правила использования аргумента `path` функции `jsonb_set`. Если значение `jsonb` является массивом, числовые индексы начинаются с нуля, а отрицательные целые числа отсчитывают элементы с конца массива к началу.

Обращения к срезам массивов не поддерживаются. Результат обращения по индексу всегда имеет тип `jsonb`.

Используя операцию обращения по индексу в предложении `SET` оператора `UPDATE`, можно изменять значения `jsonb`. Путь такого обращения должен быть «проходимым» для всех указанных значений, если они существуют. Путь `val['a']['b']['c']` можно пройти полностью до `c`, если `val`, `val['a']` и `val['a']['b']` — объекты. Если же значение `val['a']` или `val['a']['b']` не определено, будет создан пустой объект, заполняемый по мере необходимости. Однако, если значение собственно `val` или любое из промежуточных значений существует и является не объектом, а строкой, числом или сущностью `jsonb null`, пройти этот путь невозможно, поэтому возникает ошибка и транзакция прерывается.

Пример синтаксиса обращения по индексу:

-- Извлечь значение объекта по ключу:

```
SELECT ('{"a": 1} '::jsonb) ['a'];
```

-- Извлечь значение вложенного объекта по пути ключа:

```
SELECT ('{"a": {"b": {"c": 1}}' '::jsonb) ['a'] ['b'] ['c'];
```

-- Извлечь элемент массива по индексу:

```
SELECT ('[1, "2", null]' '::jsonb) [1];
```

-- Изменить значение объекта по ключу. Апострофы вокруг `'1'`: присваиваемое значение также должно быть типа `jsonb`:

```
UPDATE table_name SET jsonb_field['key'] = '1';
```

-- Это вызовет ошибку, если `jsonb_field['a']['b']` в какой-либо записи является не объектом. В `{ "a": 1 }` ключу `'a'` соответствует числовое значение ключа `'a'`:

```
UPDATE table_name SET jsonb_field['a'] ['b'] ['c'] = '1';
```

-- Отфильтровать записи предложением `WHERE` с обращением по ключу. Поскольку результат обращения по ключу будет иметь тип `jsonb`, такой же тип должно иметь

сравниваемое с ним значение. Двойные кавычки добавлены, чтобы строка "value" стала допустимой строкой jsonb:

```
SELECT * FROM table_name WHERE jsonb_field['key'] = '"value"';
```

Присваивание jsonb при обращении по индексу в некоторых особых случаях работает не так, как с функцией jsonb\_set. Когда исходное значение jsonb — NULL, присваивание при обращении по ключу будет работать, как будто это значение — пустое значение JSON (типа массив или объект, в зависимости от типа ключа):

-- Там, где поле jsonb\_field было NULL, оно станет {"a": 1}

```
UPDATE table_name SET jsonb_field['a'] = '1';
```

-- Там, где поле jsonb\_field было NULL, оно станет [1]

```
UPDATE table_name SET jsonb_field[0] = '1';
```

Если индекс указан для массива, содержащего недостаточно элементов, в него будут добавляться элементы со значением NULL до тех пор, пока индекс не станет достижимым и пока не станет возможным задать значение.

-- Там, где поле jsonb\_field было [], оно станет [null, null, 2]; там, где поле jsonb\_field было [0], оно станет [0, null, 2]:

```
UPDATE table_name SET jsonb_field[2] = '2';
```

Значение jsonb будет принимать присваивания по несуществующим путям, если последний существующий элемент, который нужно пройти — объект или массив, в зависимости от соответствующего ключа (элемент, на который указывает последний компонент пути, не проходится и может быть любым). Вложенные структуры массивов и объектов будут созданы и для первых дополнятся элементами null до нужной позиции (заданной в пути), в которую может быть помещено присваиваемое значение.

-- Там, где поле jsonb\_field было {}, оно станет {'a': [{'b': 1}]}:

```
UPDATE table_name SET jsonb_field['a'][0]['b'] = '1';
```

-- Там, где поле `jsonb_field` было [], оно станет [null, {'a': 1}]:

```
UPDATE table_name SET jsonb_field[1]['a'] = '1';
```

#### 5.14.5. Трансформации

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

Расширения для PL/Perl называются `jsonb_plperl` и `jsonb_plperlu`. Когда они используются, значения `jsonb` отображаются в соответствующие структуры Perl: массивы, хеши или скаляры.

Расширения для PL/Python называются `jsonb_plpythonu`, `jsonb_plpython2u` и `jsonb_plpython3u`. Когда они используются, значения `jsonb` отображаются в соответствующие структуры Python: массивы, хеши или скаляры.

Из этих расширений «доверенным» считается `jsonb_plperl`, то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных. Остальные расширения могут устанавливать только суперпользователи.

#### 5.14.6. Тип `jsonpath`

Тип `jsonpath` предназначен для реализации поддержки языка путей SQL/JSON в PostgreSQL, позволяющего эффективно выполнять запросы к данным JSON. Он обеспечивает двоичное представление разобранного выражения пути SQL/JSON, определяющего, какие элементы должны извлекаться из данных JSON для дальнейшей обработки в функциях SQL/JSON.

Семантика предикатов и операторов языка путей SQL/JSON в целом соответствует SQL. В то же время, чтобы с данными JSON можно было оперировать естественным образом, в синтаксисе путей SQL/JSON приняты некоторые соглашения JavaScript:

- Точка (.) применяется для доступа к члену объекта.
- Квадратные скобки ([]) применяются для обращения к массиву.
- Элементы массивов в SQL/JSON нумеруются с 0, тогда как обычные массивы SQL — с 1.

Выражение пути SQL/JSON обычно записывается в SQL-запросе в виде символьной константы SQL, и поэтому должно заключаться в апострофы, а любой апостроф, который нужно заключить в это значение, должен дублироваться. Нередко строковые константы требуется использовать и внутри выражений путей. На такие константы распространяются соглашения JavaScript/ECMAScript: они должны заключаться в двойные кавычки, а для представления символов, которые сложно ввести иначе, используются спецпоследовательности с обратной косой чертой. В частности, символ двойных кавычек внутри строковой константы записывается как \", а собственно обратная косая черта как \\. В число других спецпоследовательностей, воспринимаемых в строках JSON, входят: \b, \f, \n, \r, \t, \v, выражающие различные управляющие символы ASCII, а также \uNNNN, выражающая символ Unicode кодом в виде четырех шестнадцатеричных цифр. Синтаксис спецпоследовательностей допускает также две записи, выходящие за рамки JSON: \xNN, выражающая символ кодом в виде только двух шестнадцатеричных цифр, и \u{N...}, позволяющая для записи кода символа использовать от 1 до 6 шестнадцатеричных цифр.

Выражение пути состоит из последовательности элементов пути, которые могут быть следующими:

- Константы примитивных типов JSON: текст Unicode, числа и значения true, false и null.
- Переменные пути.
- Операторы обращения.
- Операторы и методы jsonpath.
- Скобки, применяющиеся для образования выражений фильтра и изменения порядка вычисления пути.

Таблица 5.24 – Переменные jsonpath

Переменная	Описание
\$	Переменная, представляющая значение JSON, фигурирующее в запросе ( <i>элемент контекста</i> ).
\$varname	Именованная переменная. Ее значение может быть задано в параметре <i>vars</i> , который принимают различные функции обработки JSON.
@	Переменная, представляющая результат вычисления пути в выражениях фильтров.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Таблица 5.25 – Операторы обращения в jsonpath

Оператор обращения	Описание
.ключ ."\$имя_переменной"	Оператор обращения к члену объекта, выбираемому по заданному ключу. Если имя ключа совпадает с именем какой-либо переменной, начинающимся с \$, или не соответствует действующим в JavaScript требованиям к идентификаторам, оно должно заключаться в двойные кавычки и таким образом представляться как строковая константа.
.*	Оператор обращения по звездочке, который возвращает значения всех членов, находящихся на верхнем уровне объекта.
.**	Рекурсивный оператор обращения по звездочке, который проходит по всем уровням иерархии JSON текущего объекта и возвращает все значения членов, вне зависимости от их уровня вложенности. Это реализованное в PostgreSQL расширение стандарта SQL/JSON.
.**{уровень} .**{начальный_уровень to конечный_уровень}	Этот оператор подобен **, но выбирает только указанные уровни иерархии JSON. Уровни вложенности задаются целыми числами, при этом нулевой уровень соответствует текущему объекту. Для обращения к самому нижнему уровню вложенности можно использовать ключевое слово last. Это реализованное в PostgreSQL расширение стандарта SQL/JSON.
[селектор, ...]	Оператор обращения к элементу массива. Селектор может задаваться в двух формах: индекс или начальный_индекс to конечный_индекс. Первая форма выбирает единственный элемент по индексу. Вторая форма выбирает срез массива по двум индексам, включающий крайние элементы, соответствующие значениям начальный_индекс и конечный_индекс. Задаваемый индекс может быть целочисленным значением или выражением, возвращающим единственное число, которое автоматически приводится к целому. Индекс 0 соответствует первому элементу массива. Также в качестве индекса принимается ключевое слово last, обозначающее индекс последнего элемента массива, что полезно при обработке массивов неизвестной длины.
[*]	Оператор обращения к элементам массива по звездочке, возвращающий все элементы массива.



## 5.15. Массивы

PostgreSQL позволяет определять столбцы таблицы как многомерные массивы переменной длины. Элементами массивов могут быть любые встроенные или определенные пользователями базовые типы, перечисления, составные типы, типы-диапазоны или домены.

### 5.15.1. Объявления типов массивов

Чтобы проиллюстрировать использование массивов, создается таблица:

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

Как показано, для объявления типа массива к названию типа элементов добавляются квадратные скобки ([]). Показанная выше команда создаст таблицу `sal_emp` со столбцами типов `text` (`name`), одномерный массив с элементами `integer` (`pay_by_quarter`), представляющий квартальную зарплату работников, и двухмерный массив с элементами `text` (`schedule`), представляющий недельный график работника.

Команда `CREATE TABLE` позволяет также указать точный размер массивов:

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

Однако текущая реализация игнорирует все указанные размеры, т. е. фактически размер массива остается неопределенным.

Текущая реализация также не ограничивает число размерностей. Все элементы массивов считаются одного типа, вне зависимости от его размера и числа размерностей. Поэтому явно указывать число элементов или размерностей в команде `CREATE TABLE` имеет смысл только для документирования, на механизм работы с массивом это не влияет.

Для объявления одномерных массивов можно применять альтернативную запись с ключевым словом `ARRAY`, соответствующую стандарту SQL. 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,
```

В этом случае PostgreSQL не накладывает ограничения на фактический размер массива.

### 5.15.2. Ввод значения массива

Чтобы записать значение массива в виде буквальной константы, заключите значения элементов в фигурные скобки и разделите их запятыми. Можно заключить значение любого элемента в двойные кавычки, а если он содержит запятые или фигурные скобки, это обязательно нужно сделать. Таким образом, общий формат константы массива выглядит так:

```
'{ значение1 разделитель значение2 разделитель ... }'
```

где разделитель — символ, указанный в качестве разделителя в соответствующей записи в таблице `pg_type`. Для стандартных типов данных, существующих в дистрибутиве PostgreSQL, разделителем является запятая (,), за исключением лишь типа `box`, в котором разделитель — точка с запятой (;). Каждое значение здесь — это либо константа типа элемента массива, либо вложенный массив. Константа массива может быть такой:

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

Эта константа определяет двухмерный массив 3x3, состоящий из трех вложенных массивов целых чисел.

Чтобы присвоить элементу массива значение `NULL`, достаточно просто написать `NULL` (регистр символов при этом не имеет значения). Если же требуется добавить в массив строку, содержащую «NULL», это слово нужно заключить в двойные кавычки.

Такого рода константы массивов на самом деле представляют собой всего лишь частный случай констант. Константа изначально воспринимается как строка и передается процедуре преобразования вводимого массива. При этом может потребоваться явно указать целевой тип.

Теперь можно 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},{meetin
g,lunch}}
(2 rows)
```

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

```
INSERT INTO sal_emp
VALUES ('Bill',
```

```
'{10000, 10000, 10000, 10000}',  
'{{"meeting", "lunch"}, {"meeting"}}}');
```

ОШИБКА: для многомерных массивов должны задаваться выражения с соответствующими размерностями

Также 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']]);
```

Элементы массива здесь — это простые SQL-константы или выражения; и поэтому строки будут заключаться в одинарные апострофы, а не в двойные, как в буквальной константе массива.

### 5.15.3. Обращение к массивам

Добавив данные в таблицу, можно перейти к выборкам. Сначала покажем, как получить один элемент массива. Этот запрос получает имена сотрудников, зарплата которых изменилась во втором квартале:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <>  
pay_by_quarter[2];  
  
name  
-----  
Carol  
(1 row)
```

Индексы элементов массива записываются в квадратных скобках. По умолчанию в PostgreSQL действует соглашение о нумерации элементов массива с 1, то есть в массиве из  $n$  элементов первым считается `array[1]`, а последним — `array[n]`.

Этот запрос выдает зарплату всех сотрудников в третьем квартале:

```
SELECT pay_by_quarter[3] FROM sal_emp;
```

```
pay_by_quarter
-----
          10000
          25000
(2 rows)
```

Можно получать обычные прямоугольные срезы массива, то есть подмассивы. Срез массива обозначается как нижняя-граница:верхняя-граница для одной или нескольких размерностей. Этот запрос получает первые пункты в графике «Bill» в первые два дня недели:

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

```
schedule
-----
{{meeting},{training}}
(1 row)
```

Если одна из размерностей записана в виде среза, то есть содержит двоеточие, тогда срез распространяется на все размерности. Если при этом для размерности указывается только одно число (без двоеточия), в срез войдут элемент от 1 до заданного номера. В этом примере `[2]` будет равнозначно `[1:2]`:

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

```
schedule
-----
{{meeting,lunch},{training,presentation}}
(1 row)
```

Во избежание путаницы с обращением к одному элементу, срезы лучше всегда записывать явно для всех измерений, [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.

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

```
-----
schedule
-----
{{lunch},{presentation}}
(1 row)
```

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

```
-----
schedule
-----
{{meeting},{training}}
(1 row)
```

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

Аналогично, NULL возвращается при обращении к срезу массива, если сам массив или одно из выражений, определяющих индексы элементов, равны NULL. Однако в других случаях, 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 срез, сокращенный до области пересечения.

Текущие размеры значения массива можно получить с помощью функции array\_dims:

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

```
array_dims
-----
[1:2][1:2]
(1 row)
```

array\_dims выдает результат типа text, что удобно скорее для людей, чем для программ. Размеры массива также можно получить с помощью функций array\_upper и array\_lower, которые возвращают соответственно верхнюю и нижнюю границу для указанной размерности:

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

```
array_upper
-----
                2
(1 row)
```

array\_length возвращает число элементов в указанной размерности массива:

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

```
array_length
-----
                2
(1 row)
```

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

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

```
cardinality
-----
                4
(1 row)
```

#### 5.15.4. Изменение массивов

Значение массива можно заменить полностью так:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
```

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

```
WHERE name = 'Carol';
```

или используя синтаксис ARRAY:

```
UPDATE sal_emp SET pay_by_quarter =  
ARRAY[25000,25000,27000,27000]  
  
WHERE name = 'Carol';
```

Также можно изменить один элемент массива:

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

или срез:

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

При этом в указании среза может быть опущена нижняя-граница и/или верхняя-граница, но только для массива, отличного от NULL, и имеющего ненулевую размерность (иначе неизвестно, какие граничные значения должны подставляться вместо опущенных).

Сохраненный массив можно расширить, определив значения ранее отсутствовавших в нем элементов. При этом все элементы, располагающиеся между существовавшими ранее и новыми, принимают значения NULL. Если массив myarray содержит 4 элемента, после присваивания 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. Можно присвоить значение выражению myarray[-2:7] и таким образом создать массив, в котором будут элементы с индексами от -2 до 7.

Значения массива также можно сконструировать с помощью оператора конкатенации ||:

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



```
?column?
-----
{1,2,3,4}
(1 row)

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

?column?
-----
{{5,6},{1,2},{3,4}}
(1 row)
```

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

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

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

array_dims
-----
[0:2]
(1 row)

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

array_dims
-----
[1:3]
(1 row)
```

Когда складываются два массива одинаковых размерностей, в результате сохраняется нижняя граница внешней размерности левого операнда. Выходной массив включает все элементы левого операнда, после которых добавляются все элементы правого.

```
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.

```
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}}
```

В простых случаях описанный выше оператор конкатенации предпочтительнее непосредственного вызова этих функций. Однако так как оператор конкатенации перегружен для решения всех трех задач, возможны ситуации, когда лучше применить одну из этих функций во избежание неоднозначности.

В показанных примерах анализатор запроса видит целочисленный массив с одной стороны оператора конкатенации и константу неопределенного типа с другой. Согласно своим правилам разрешения типа констант, он полагает, что она имеет тот же тип, что и другой операнд — в данном случае целочисленный массив. Поэтому предполагается, что оператор конкатенации здесь представляет функцию `array_cat`, а не `array_append`. Если это решение оказывается неверным, его можно скорректировать, приведя константу к типу элемента массива; однако может быть лучше явно использовать функцию `array_append`.

#### 5.15.5. Поиск значений в массивах

Чтобы найти значение в массиве, необходимо проверить все его элементы. Это можно сделать вручную, если знать размер массива.

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

Однако с большим массивами этот метод становится утомительным, и к тому же он не работает, когда размер массива неизвестен. Показанный выше запрос можно было переписать так:

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

```
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`.

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

Также искать в массиве значения можно, используя оператор `&&`, который проверяет, перекрывается ли левый операнд с правым.

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

Можно искать определенные 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.

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

array_position
-----
                2
(1 row)

SELECT array_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1], 1);

array_positions
-----
```

```
{1, 4, 8}
(1 row)
```



Массивы — это не множества; необходимость поиска определенных элементов в массиве может быть признаком неудачно сконструированной базы данных. Возможно, вместо массива лучше использовать отдельную таблицу, строки которой будут содержать данные элементов массива. Это может быть лучше и для поиска, и для работы с большим количеством элементов.

#### 5.15.6. Синтаксис вводимых и выводимых значений массива

Внешнее текстовое представление значения массива состоит из записи элементов, интерпретируемых по правилам ввода/вывода для типа элемента массива, и оформления структуры массива. Оформление состоит из фигурных скобок (`{` и `}`), окружающих значение массива, и знаков-разделителей между его элементами. В качестве знака-разделителя обычно используется запятая (`,`), но это может быть и другой символ; он определяется параметром `typdelim` для типа элемента массива. Для стандартных типов данных, существующих в дистрибутиве PostgreSQL, разделителем является запятая (`,`), за исключением лишь типа `box`, в котором разделитель — точка с запятой (`;`). В многомерном массиве у каждой размерности (ряд, плоскость, куб и т. д.) есть свой уровень фигурных скобок, а соседние значения в фигурных скобках на одном уровне должны отделяться разделителями.

Функция вывода массива заключает значение элемента в кавычки, если это пустая строка или оно содержит фигурные скобки, знаки-разделители, кавычки, обратную косую черту, пробельный символ или это текст `NULL`. Кавычки и обратная косая черта, включенные в такие значения, преобразуются в спецпоследовательность с обратной косой чертой. Для числовых типов данных можно рассчитывать на то, что значения никогда не будут выводиться в кавычках, но для текстовых типов следует быть готовым к тому, что выводимое значение массива может содержать кавычки.

По умолчанию нижняя граница всех размерностей массива равна одному. Чтобы представить массивы с другими нижними границами, перед содержимым массива можно указать диапазоны индексов. Такое оформление массива будет содержать квадратные

скобки ([]) вокруг нижней и верхней границ каждой размерности с двоеточием (:) между ними. За таким указанием размерности следует знак равно (=).

Процедура вывода массива включает в результат явное указание размерностей, только если нижняя граница в одной или нескольких размерностях отличается от 1.

Если в качестве значения элемента задается NULL (в любом регистре), этот элемент считается равным непосредственно NULL. Если же оно включает кавычки или обратную косую черту, элементу присваивается текстовая строка «NULL». Кроме того, для обратной совместимости с версиями PostgreSQL до 8.2, параметр конфигурации `array_nulls` можно выключить (присвоив ему `off`), чтобы строки NULL не воспринимались как значения NULL.

Как было показано ранее, записывая значение массива, любой его элемент можно заключить в кавычки. Это нужно делать, если при разборе значения массива без кавычек возможна неоднозначность. В кавычки необходимо заключать элементы, содержащие фигурные скобки, запятую (или разделитель, определенный для данного типа), кавычки, обратную косую черту, а также пробельные символы в начале или конце строки. Пустые строки и строки, содержащие одно слово NULL, также нужно заключать в кавычки. Чтобы включить кавычки или обратную косую черту в значение, заключенное в кавычки, добавьте обратную косую черту перед таким символом. С другой стороны, чтобы обойтись без кавычек, таким экранированием можно защитить все символы в данных, которые могут быть восприняты как часть синтаксиса массива.

Перед открывающей и после закрывающей скобки можно добавлять пробельные символы. Пробелы также могут окружать каждую отдельную строку значения. Во всех случаях такие пробельные символы игнорируются. Однако все пробелы в строках, заключенных в кавычки, или окруженные не пробельными символами, напротив, учитываются.



Записывать значения массивов в командах SQL часто бывает удобнее с помощью конструктора `ARRAY`. В `ARRAY` отдельные значения элементов записываются так же, как если бы они не были членами массива.

## 5.16. Составные типы

Составной тип представляет структуру табличной строки или записи; по сути это просто список имен полей и соответствующих типов данных. PostgreSQL позволяет

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

использовать составные типы во многом так же, как и простые типы. В определении таблицы можно объявить столбец составного типа.

### 5.16.1. Объявление составных типов

Ниже приведены два простых примера определения составных типов:

```
CREATE TYPE complex AS (  
    r      double precision,  
    i      double precision  
);  
  
CREATE TYPE inventory_item AS (  
    name          text,  
    supplier_id   integer,  
    price         numeric  
);
```

Синтаксис очень похож на CREATE TABLE, за исключением того, что он допускает только названия полей и их типы, какие-либо ограничения (такие как NOT NULL) в настоящее время не поддерживаются. Ключевое слово AS здесь имеет значение; без него система будет считать, что подразумевается другой тип команды CREATE TYPE, и выдаст неожиданную синтаксическую ошибку.

Определив такие типы, можно использовать их в таблицах:

```
CREATE TABLE on_hand (  
    item      inventory_item,  
    count     integer  
);  
  
INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

или функциях:

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

SELECT price_extension(item, 10) FROM on_hand;
```

Всякий раз, когда создается таблица, вместе с ней автоматически создается составной тип. Этот тип представляет тип строки таблицы, и его именем становится имя таблицы. 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. В текущей реализации есть один недостаток: так как с составным типом не могут быть связаны ограничения, то описанные в определении таблицы ограничения не применяются к значениям составного типа вне таблицы.

### 5.16.2. Конструирование составных значений

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

```
'( значение1 , значение2 , ... )'
```

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. Эта константа задает значение `NULL` для третьего поля:

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

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

```
' ( "", 42, ) '
```

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

Такого рода константы массивов на самом деле представляют собой частный случай констант. Константа изначально воспринимается как строка и передается процедуре преобразования составного типа. При этом может потребоваться явно указать тип, к которому будет приведена константа.

Значения составных типов также можно конструировать, используя синтаксис выражения `ROW`. В большинстве случаев это значительно проще, чем записывать значения в строке, так как при этом не нужно беспокоиться о вложенности кавычек.

```
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)
```

### 5.16.3. Обращение к составным типам

Чтобы обратиться к полю столбца составного типа, после имени столбца нужно добавить точку и имя поля, подобно тому, как указывается столбец после имени таблицы. На самом деле, эти обращения неотличимы, так что часто бывает необходимо использовать

скобки, чтобы команда была разобрана правильно. Можно попытаться выбрать поле столбца из тестовой таблицы `on_hand` таким образом:

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

Но это не будет работать, так как согласно правилам SQL имя `item` здесь воспринимается как имя таблицы, а не 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;
```

В результате объект в скобках будет правильно интерпретирован как ссылка на столбец `item`, из которого выбирается поле.

При выборке поля из значения составного типа также возможны подобные синтаксические казусы. Чтобы выбрать одно поле из результата функции, возвращающей составное значение, потребуется написать что-то подобное:

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

Без дополнительных скобок в этом запросе произойдет синтаксическая ошибка.

#### 5.16.4. Изменение составных типов

Ниже приведены примеры правильных команд добавления и изменения значений составных столбцов. Данные 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 опущено ключевое слово `ROW`, а во втором оно есть; присутствовать или отсутствовать оно может в обоих случаях.

Можно изменить также отдельное поле составного столбца:

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

При этом не нужно 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.

И также можем указать поля в качестве цели команды INSERT:

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

Если при этом не указываются значения для всех полей столбца, оставшиеся поля будут заполнены значениями NULL.

#### 5.16.5. Использование составных типов в запросах

С составными типами в запросах связаны особые правила синтаксиса и поведение. Эти правила образуют полезные конструкции, но они могут быть неочевидными, если не понимать стоящую за ними логику.

В PostgreSQL ссылка на имя таблицы (или ее псевдоним) в запросе по сути является ссылкой на составное значение текущей строки в этой таблице. Имея таблицу `inventory_item`, показанную выше, можно написать:

```
SELECT c FROM inventory_item c;
```

Этот запрос выдает один столбец с составным значением, и его результат может быть таким:

```

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

Простые имена сопоставляются сначала с именами столбцов, и только потом с именами таблиц, так что такой результат получается только потому, что в таблицах запроса не оказалось столбца с именем `c`.

Обычную запись полного имени столбца вида имя\_таблицы.имя\_столбца можно понимать как применение выбора поля к составному значению текущей строки таблицы.

Когда пишется:

```
SELECT c.* FROM inventory_item c;
```

то, согласно стандарту SQL, должно получиться содержимое таблицы, развернутое 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;
```

PostgreSQL применяет такое развертывание для любых выражений с составными значениями, хотя как показано выше, необходимо заключить в скобки значение, к которому применяется `.*`, если только это не простое имя таблицы. 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;
```



PostgreSQL осуществляет развертывание столбцов фактически переводя первую форму во вторую. Таким образом, в данном примере `myfunc()` будет вызываться три раза для каждой строки и с одним, и с другим синтаксисом. Если это дорогостоящая функция, можно использовать такой запрос:

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

Размещение вызова функции в элементе `FROM LATERAL` гарантирует, что она будет вызываться для строки не более одного раза. Конструкция `m.*` так же

разворачивается в m.a, m.b, m.c, но теперь эти переменные просто ссылаются на выходные значения FROM.

Запись `составное_значение.*` приводит к такому разворачиванию столбцов, когда она фигурирует на верхнем уровне выходного списка SELECT, в списке RETURNING команд INSERT/UPDATE/DELETE, в предложении VALUES или в конструкторе строки. Во всех других контекстах (включая вложенные в одну из этих конструкций), добавление `.*` к составному значению не меняет это значение, так как это воспринимается как «все столбцы» и поэтому выдается то же составное значение. Если функция `somefunc()` принимает в качестве аргумента составное значение, эти запросы равносильны:

```
SELECT somefunc(c.*) FROM inventory_item c;  
SELECT somefunc(c) FROM inventory_item c;
```

В обоих случаях текущая строка таблицы `inventory_item` передается функции как один аргумент с составным значением. И хотя дополнение `.*` в этих случаях не играет роли, использовать его считается хорошим стилем, так как это ясно указывает на использование составного значения. В частности анализатор запроса воспримет «с» в записи `с.*` как ссылку на имя или псевдоним таблицы, а не имя столбца, что избавляет от неоднозначности; тогда как без `.*` неясно, означает ли `с` имя таблицы или имя столбца, и на самом деле при наличии столбца с именем `с` будет выбрано второе прочтение.

Эту концепцию демонстрирует и следующий пример, все запросы в котором действуют одинаково:

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

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

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

В последнем случае используется конструктор строки, в котором опущено ключевое слово ROW.

Другая особенность синтаксиса, связанная с составными значениями, состоит в том, что можно использовать функциональную запись для извлечения поля составного значения. Это легко можно объяснить тем, что записи поле(таблица) и таблица.поле взаимозаменяемы. Следующие запросы равнозначны:

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

Более того, если есть функция, принимающая один аргумент составного типа, можно вызвать ее в любой записи. Все эти запросы равносильны:

```
SELECT somefunc(c) FROM inventory_item c;  
SELECT somefunc(c.*) FROM inventory_item c;  
SELECT c.somefunc FROM inventory_item c;
```

Эта равнозначность записи с полем и функциональной записи позволяет использовать с составными типами функции, реализующие «вычисляемые поля». При этом приложению, использующему последний из предыдущих запросов, не нужно знать, что фактически somefunc — не настоящий столбец таблицы.



Учитывая такое поведение, будет неразумно давать функции, принимающей один аргумент составного типа, то же имя, что и одному из полей данного составного типа. В случае неоднозначности прочтение имени поля будет выбрано при использовании синтаксиса обращения к полю, а прочтение имени функции — если используется синтаксис вызова функции.

### 5.16.6. Синтаксис вводимых и выводимых значений составного типа

Внешнее текстовое представление составного значения состоит из записи элементов, интерпретируемых по правилам ввода/вывода для соответствующих типов полей, и оформления структуры составного типа. Оформление состоит из круглых скобок (( и )) окружающих все значение, и запятых (,) между его элементами. Пробельные символы вне скобок игнорируются, но внутри они считаются частью соответствующего элемента и могут учитываться или не учитываться в зависимости от правил преобразования вводимых данных для типа этого элемента. В записи:

```
' ( 42 ) '
```

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

Записывая составное значение, любой его элемент можно заключить в кавычки. Это нужно делать, если при разборе этого значения без кавычек возможна неоднозначность. В кавычки нужно заключать элементы, содержащие скобки, кавычки, запятую или обратную косую черту. Чтобы включить в поле составного значения, заключенное в кавычки, такие символы, как кавычки или обратная косая черта, перед ними нужно добавить обратную косую черту. С другой стороны, можно обойтись без кавычек, защитив все символы в данных, которые могут быть восприняты как часть синтаксиса составного значения, с помощью спецпоследовательностей.

Значение NULL в этой записи представляется пустым местом (когда между запятыми или скобками нет никаких символов). Чтобы ввести именно пустую строку, а не NULL, нужно написать "".

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



Помните, что написанная SQL-команда прежде всего интерпретируется как текстовая строка, а затем как составное значение. Вследствие этого число символов обратной косой черты удваивается (если используются

спецпоследовательности). 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 (`"\"\\\""`). В свою очередь, он передает эту строку процедуре ввода значения типа `text`, где она преобразуется в `"\"`. Во избежание такого дублирования спецсимволов строки можно заключать в доллары.

### 5.17. Диапазонные типы

Диапазонные типы представляют диапазоны значений некоторого типа данных (он также называется подтипом диапазона). Диапазон типа `timestamp` может представлять временной интервал, когда зарезервирован зал заседаний. В данном случае типом данных будет `tsrange` (сокращение от «`timestamp range`»), а подтипом — `timestamp`. Подтип должен быть полностью упорядочиваемым, чтобы можно было однозначно определить, где находится значение по отношению к диапазону: внутри, до или после него.

Диапазонные типы полезны тем, что позволяют представить множество возможных значений в одной структуре данных и четко выразить такие понятия, как пересечение диапазонов. Наиболее очевидный вариант их использования — применять диапазоны даты и времени для составления расписания, но также полезными могут оказаться диапазоны цен, интервалы измерений и т. д.

Для каждого диапазонного типа есть соответствующий мультидиапазонный тип. Мультидиапазон представляет собой упорядоченный список несмежных, непустых и отличных от `NULL` диапазонов. Большинство диапазонных операторов работают и с мультидиапазонами. Кроме того, есть функции для работы именно с мультидиапазонными типами.



### 5.17.1. Встроенные диапазонные и мультидиапазонные типы

PostgreSQL имеет следующие встроенные диапазонные типы:

- `int4range` — диапазон подтипа `integer`, `int4multirange` — соответствующий мультидиапазон;
- `int8range` — диапазон подтипа `bigint`, `int8multirange` — соответствующий мультидиапазон;
- `numrange` — диапазон подтипа `numeric`, `nummultirange` — соответствующий мультидиапазон;
- `tsrange` — диапазон подтипа `timestamp without time zone`, `tsmultirange` — соответствующий мультидиапазон;
- `tstzrange` — диапазон подтипа `timestamp with time zone`, `tstzmultirange` — соответствующий мультидиапазон;
- `daterange` — диапазон подтипа `date`, `datemultirange` — соответствующий мультидиапазон.

### 5.17.2. Включение и исключение границ

Любой непустой диапазон имеет две границы, верхнюю и нижнюю, и включает все точки между этими значениями. В него также может входить точка, лежащая на границе, если диапазон включает эту границу. И наоборот, если диапазон не включает границу, считается, что точка, лежащая на этой границе, в него не входит.

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

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

### 5.17.3. Неограниченные (бесконечные) диапазоны

Нижнюю границу диапазона можно опустить и определить тем самым диапазон, включающий все значения, лежащие ниже верхней границы, например: `(,3]`. Подобным образом, если не определить верхнюю границу, в диапазон войдут все значения, лежащие выше нижней границы. Если же опущена и нижняя, и верхняя границы, такой диапазон

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

будет включать все возможные значения своего подтипа. Указание отсутствующей границы как включаемой в диапазон автоматически преобразуется в исключающее; например, [,] преобразуется в (,). Можно воспринимать отсутствующие значения как плюс/минус бесконечность, но все же это особые значения диапазонного типа, которые охватывают 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 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,].

Проверить, определена ли верхняя или нижняя граница, можно с помощью функций lower\_inf и upper\_inf, соответственно.

#### 5.17.4. Ввод/вывод диапазонов

Вводимое значение диапазона должно записываться в одной из следующих форм:

```
(нижняя-граница, верхняя-граница)
(нижняя-граница, верхняя-граница]
[нижняя-граница, верхняя-граница)
[нижняя-граница, верхняя-граница]
empty
```

Тип скобок (квадратные или круглые) определяет, включаются ли в диапазон соответствующие границы, как описано выше. Последняя форма содержит только слово empty и определяет пустой диапазон (диапазон, не содержащий точек).

Здесь нижняя-граница может быть строкой с допустимым значением подтипа или быть пустой (тогда диапазон будет без нижней границы). Аналогично, верхняя-граница может задаваться одним из значений подтипа или быть неопределенной (пустой).

Любое значение границы диапазона можно заключить в кавычки ("). А если значение содержит круглые или квадратные скобки, запятые, кавычки или обратную косую черту, использовать кавычки необходимо, чтобы эти символы не рассматривались как часть синтаксиса диапазона. Чтобы включить в значение границы диапазона, заключенное в

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

кавычки, такие символы, как кавычки или обратная косая черта, перед ними нужно добавить обратную косую черту. Кроме того, продублированные кавычки в значении диапазона, заключенного в кавычки, воспринимаются как одинарные, подобно апострофам в строках SQL. С другой стороны, можно обойтись без кавычек, защитив все символы в данных, которые могут быть восприняты как часть синтаксиса диапазона, с помощью спецпоследовательностей. Чтобы задать в качестве границы пустую строку, нужно ввести "", так как пустая строка без кавычек будет означать отсутствие границы.

Пробельные символы до и после определения диапазона игнорируются, но когда они присутствуют внутри скобок, они воспринимаются как часть значения верхней или нижней границы. (Хотя они могут также игнорироваться в зависимости от подтипа диапазона.)



Эти правила очень похожи на правила записи значений для полей составных типов.

Вводимое значение мультидиапазона заключается в фигурные скобки ({ и }) и содержит ноль или более диапазонов, разделенных запятыми. До и после скобок и запятых допускаются пробельные символы. Синтаксис сделан похожим на синтаксис массивов, но мультидиапазоны намного проще: они имеют только одну размерность, а их содержимое не нужно заключать в кавычки.

#### 5.17.5. Конструирование диапазонов и мультидиапазонов

Для каждого диапазонного типа определена функция конструктора, имеющая то же имя, что и данный тип. Использовать этот конструктор обычно удобнее, чем записывать текстовую константу диапазона, так как это избавляет от потребности в дополнительных кавычках. Функция конструктора может принимать два или три параметра. Вариант с двумя параметрами создает диапазон в стандартной форме (нижняя граница включается, верхняя исключается), тогда как для варианта с тремя параметрами включение границ определяется третьим параметром. Третий параметр должен содержать одну из строк: «()», «()», «[]» или «[]».

-- 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.

```
SELECT int8range(1, 14, '[');
```

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

```
SELECT numrange(NULL, 2.2);
```

Для каждого диапазонного типа также есть конструктор мультидиапозона с тем же именем, что и у мультидиапазонного типа. Функция-конструктор принимает ноль или более аргументов, представляющих собой диапазоны соответствующего типа.

```
SELECT nummultirange();  
SELECT nummultirange(numrange(1.0, 14.0));  
SELECT nummultirange(numrange(1.0, 14.0), numrange(20.0,  
25.0));
```

#### 5.17.6. Типы дискретных диапазонов

Дискретным диапазоном считается диапазон, для подтипа которого однозначно определен «шаг», как например для типов `integer` и `date`. Значения этих двух типов можно назвать соседними, когда между ними нет никаких других значений. В непрерывных диапазонах, напротив, всегда (или почти всегда) можно найти еще одно значение между двумя данными. Непрерывным диапазоном будет диапазон с подтипами `numeric` и `timestamp`.

Можно также считать дискретным подтип диапазона, в котором четко определены понятия «следующего» и «предыдущего» элемента для каждого значения. Такие определения позволяют преобразовывать границы диапазона из включаемых в исключаемые, выбирая следующий или предыдущий элемент вместо заданного значения.

Диапазоны целочисленного типа [4,8] и (3,9) описывают одно и то же множество значений; но для диапазона подтипа `numeric` это не так.

Для типа дискретного диапазона определяется функция канонизации, учитывающая размер шага для данного подтипа. Задача этой функции — преобразовать равнозначные диапазоны к единственному представлению, в частности нормализовать включаемые и исключаемые границы. Если функция канонизации не определена, диапазоны с различным определением будут всегда считаться разными, даже когда они на самом деле представляют одно множество значений.

Для встроенных типов `int4range`, `int8range` и `daterange` каноническое представление включает нижнюю границу и не включает верхнюю; то есть диапазон приводится к виду `[]`. Однако для нестандартных типов можно использовать и другие соглашения.

#### 5.17.7. Определение новых диапазонных типов

Пользователи могут определять собственные диапазонные типы. Это может быть полезно, когда нужно использовать диапазоны с подтипами, для которых нет встроенных диапазонных типов. Можно определить новый тип диапазона для подтипа `float8`:

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

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

Так как для `float8` осмысленное значение «шага» не определено, функция канонизации в данном примере не задается.

При определении собственного диапазона, система автоматически создает мультидиапазонный тип.

Определяя собственный диапазонный тип, можно выбрать другие правила сортировки или класс оператора В-дерева для его подтипа, что позволит изменить порядок значений, от которого зависит, какие значения попадают в заданный диапазон.

Если подтип можно рассматривать как дискретный, а не непрерывный, в команде CREATE TYPE следует также задать функцию канонизации. Этой функции будет передаваться значение диапазона, а она должна вернуть равнозначное значение, но, возможно, с другими границами и форматированием. Для двух диапазонов, представляющих одно множество значений целочисленных диапазонов [1, 7] и [1, 8), функция канонизации должна выдавать один результат. Какое именно представление будет считаться каноническим, не имеет значения — главное, чтобы два равнозначных диапазона, отформатированных по-разному, всегда преобразовывались в одно значение с одинаковым форматированием. Помимо исправления формата включаемых/исключаемых границ, функция канонизации может округлять значения границ, если размер шага превышает точность хранения подтипа. В типе диапазона для подтипа timestamp можно определить размер шага, равный часу, тогда функция канонизации должна будет округлить границы, заданные с точностью до минут, либо вместо этого выдать ошибку.

Помимо этого, для любого диапазонного типа, ориентированного на использование с индексами GiST или SP-GiST, должна быть определена разница значений подтипов, функция subtype\_diff. (Индекс сможет работать и без subtype\_diff, но в большинстве случаев это будет не так эффективно.) Эта функция принимает на вход два значения подтипа и возвращает их разницу (т. е. X минус Y) в значении типа float8. В показанном выше примере может использоваться функция float8mi, определяющая нижележащую реализацию обычного оператора «минус» для типа float8, но для другого подтипа могут потребоваться дополнительные преобразования. Иногда для представления разницы в числовом виде требуется еще и творческий подход. Функция subtype\_diff, насколько это возможно, должна быть согласована с порядком сортировки, вытекающим из выбранных правил сортировки и класса оператора; то есть, ее результат должен быть положительным, если согласно порядку сортировки первый ее аргумент больше второго.

Еще один, не столь тривиальный пример функции subtype\_diff:

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

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

```
CREATE TYPE timerange AS RANGE (
    subtype = time,
    subtype_diff = time_subtype_diff
);

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

Дополнительные сведения о создании диапазонных типов можно найти в описании CREATE TYPE.

### 5.17.8. Индексация

Для столбцов, имеющих диапазонный тип, можно создать индексы GiST и SP-GiST. Также индексы GiST можно создавать для столбцов мультидиапазонных типов. Так создается индекс GiST:

```
CREATE INDEX reservation_idx ON reservation USING GIST
(during);
```

Индекс GiST или SP-GiST для диапазонов помогает ускорить запросы со следующими операторами: =, &&, <@, @>, <<, >>, -|-, &< и &>. Индекс GiST для мультидиапазонов может ускорить запросы, действующие один набор мультидиапазонных операторов. Индекс GiST для диапазонов и индекс GiST для мультидиапазонов могут также соответственно ускорить запросы, действующие следующие межтиповые операторы диапазон-мультидиапазон и мультидиапазон-диапазон: &&, <@, @>, <<, >>, -|-, &< и &>.

Кроме того, для таких столбцов можно 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 этот порядок может быть произвольным и он не очень важен в реальном мире. Поддержка B-деревьев и хешей диапазонными типами нужна в основном для сортировки и хеширования при выполнении запросов, но не для создания самих индексов.

### 5.17.9. Ограничения для диапазонов

Тогда как для скалярных значений естественным ограничением является UNIQUE, оно обычно не подходит для диапазонных типов. Вместо этого чаще оказываются полезнее ограничения-исключения. Такие ограничения позволяют определить условие «непересечения» диапазонов.

```
CREATE TABLE reservation (
    during tsrange,
    EXCLUDE USING GIST (during WITH &&)
);
```

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

```
INSERT INTO reservation VALUES
    ('[2010-01-01 11:30, 2010-01-01 15:00)');
INSERT 0 1
```

```
INSERT INTO reservation VALUES
    ('[2010-01-01 14:45, 2010-01-01 15:45)');
```

ОШИБКА: конфликтующее значение ключа нарушает ограничение-исключение "reservation\_during\_excl"

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

Для максимальной гибкости в ограничении-исключении можно сочетать простые скалярные типы данных с диапазонами, используя расширение btree\_gist. Если btree\_gist установлено, следующее ограничение не будет допускать пересекающиеся диапазоны.

### 5.18. Типы доменов

Домен — пользовательский тип данных, основанный на другом нижележащем типе. Он может быть определен с условиями, ограничивающими множество допустимых значений подмножеством значений нижележащего типа. В остальном он ведет себя как

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------



нижележащий тип — с доменным типом будут работать любые операторы или функции, применимые к нижележащему типу. Нижележащим типом может быть любой встроенный или пользовательский базовый тип, тип-перечисление, массив, составной тип, диапазон или другой домен.

Можно создать домен поверх целых чисел, принимающий только положительные числа:

```
CREATE DOMAIN posint AS integer CHECK (VALUE > 0);
CREATE TABLE mytable (id posint);
INSERT INTO mytable VALUES(1);    -- работает
INSERT INTO mytable VALUES(-1);  -- ошибка
```

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

## 5.19. Идентификаторы объектов

Идентификатор объекта (Object Identifier, OID) используется внутри PostgreSQL в качестве первичного ключа различных системных таблиц. Идентификатор объекта представляется в типе `oid`. Также существуют различные типы-псевдонимы для `oid`, с именами `regсущность`. Обзор этих типов приведен в таблице 5.26.

В настоящее время тип `oid` реализован как четырехбайтное целое. Таким образом, значение этого типа может быть недостаточно большим для обеспечения уникальности в базе данных или даже в отдельных больших таблицах.

Для самого типа `oid` помимо сравнения определены всего несколько операторов. Однако его можно привести к целому и затем задействовать в обычных целочисленных вычислениях.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Типы-псевдонимы OID сами по себе не вводят новых операций и отличаются только специализированными функциями ввода/вывода. Эти функции могут принимать и выводить не просто числовые значения, как тип oid, а символические имена системных объектов. Эти типы позволяют упростить поиск объектов по значениям OID. Чтобы выбрать из pg\_attribute строки, относящиеся к таблице mytable, можно написать:

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

вместо:

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

Хотя второй вариант выглядит не таким уж плохим, но это лишь очень простой запрос. Если же потребуется выбрать правильный OID, когда таблица mytable есть в нескольких схемах, вложенный подзапрос будет гораздо сложнее. Преобразователь вводимого значения типа regclass находит таблицу согласно заданному пути поиска схем, так что он делает «все правильно» автоматически. Аналогично, приведя идентификатор таблицы к типу regclass, можно получить символическое представление числового кода.

Таблица 5.26 – Идентификаторы объектов

Имя	Ссылки	Описание	Пример значения
oid	any	числовой идентификатор объекта	564182
regclass	pg_class	имя отношения	pg_type
regcollation	pg_collation	имя правила сортировки	"POSIX"
regconfig	pg_ts_config	конфигурация текстового поиска	english
regdictionary	pg_ts_dict	словарь текстового поиска	simple
regnamespace	pg_namespace	пространство имен	pg_catalog
regoper	pg_operator	имя оператора	+
regoperator	pg_operator	оператор с типами аргументов	*(integer, integer) или - (NONE, integer)
regproc	pg_proc	имя функции	sum
regprocedure	pg_proc	функция с типами аргументов	sum(int4)
regrole	pg_authid	имя роли	smithee

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Имя	Ссылки	Описание	Пример значения
regtype	pg_type	имя типа данных	integer

Все типы псевдонимов OID для объектов, сгруппированных в пространство имен, принимают имена, дополненные именем схемы, и выводят имена со схемой, если данный объект нельзя будет найти в текущем пути поиска без имени схемы. `myschema.mytable` является приемлемым входным значением для `regclass` (если существует такая таблица). Это значение может выводиться как `myschema.mytable` или просто `mytable`, в зависимости от текущего пути поиска. Типы `regproc` и `regoper` принимают только уникальные вводимые имена (не перегруженные), что ограничивает их применимость; в большинстве случаев лучше использовать `regprocedure` или `regoperator`. Для типа `regoperator` в записи унарного оператора неиспользуемый операнд заменяется словом `NONE`.

Функции ввода для данных типов допускают пробелы между компонентами и приводят буквы верхнего регистра к нижнему, за исключением строки в двойных кавычках; это сделано для того, чтобы правила записи были похожи на принятые для записи имен объектов в SQL. И наоборот, функции вывода будут добавлять двойные кавычки, если это необходимо, чтобы выводимая строка была допустимым идентификатором SQL. OID функции с именем `Foo` (с `F` в верхнем регистре), принимающей два целочисленных аргумента, можно ввести как `'"Foo" (int, integer) '::regprocedure`. Результат будет выглядеть как `"Foo"(integer,integer)`. И имя функции, и имена типов аргументов также могут быть дополнены схемой.

Многие встроенные функции PostgreSQL принимают OID таблицы или другого типа объекта БД и для удобства объявляются как принимающие `regclass` (или соответствующий тип-псевдоним OID). Это означает, что вам не нужно искать OID объекта вручную, а можно просто ввести его имя в виде строки.



Когда аргумент такой функции записывается как текстовая строка в чистом виде, она становится константой типа `regclass`. Так как фактически это будет просто значение OID, оно будет привязано к изначально идентифицированной последовательности, несмотря на то, что она может быть переименована, перенесена в другую схему и т. д. Такое «раннее связывание» обычно желательно для ссылок на последовательности в значениях столбцов по умолчанию и представлениях. Но иногда возникает необходимость в «позднем

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

связывании», когда ссылки на последовательности распознаются в процессе выполнения. Чтобы получить такое поведение, нужно принудительно изменить тип константы с `regclass` на `text`:

```
nextval('foo'::text)      foo распознается во время
выполнения
```

Для поиска во время выполнения также может использоваться функция `to_regclass()` и подобные.

Другой практический пример использования `regclass` — поиск OID таблицы, отображенной в представлениях `information_schema`, которые не предоставляют такие OID напрямую. Можно вызвать функцию `pg_relation_size()`, для которой требуется OID таблицы. С учетом указанных выше правил, далее представлен корректный способ вызова данной функции:

```
SELECT table_schema, table_name,
       pg_relation_size((quote_ident(table_schema) || '.' ||
                           quote_ident(table_name))::regclass)
FROM information_schema.tables
WHERE ...
```

Функция `quote_ident()` заключит идентификатор в двойные кавычки, когда это необходимо. Более простым способом кажется:

```
SELECT pg_relation_size(table_name)
FROM information_schema.tables
WHERE ...
```

Использовать этот способ не рекомендуется, потому что он не работает для таблиц, которые не входят в заданный путь поиска или имена которых нужно заключать в кавычки.

Дополнительным свойством большинства типов псевдонимов OID является образование зависимостей. Когда в сохраненном выражении фигурирует константа одного из этих типов (в представлении или в значении столбца по умолчанию), это создает

зависимость от целевого объекта. Если значение по умолчанию определяется выражением `nextval('my_seq'::regclass)`, PostgreSQL понимает, что это выражение зависит от последовательности `my_seq`, и не позволит удалить последовательность раньше, чем будет удалено это выражение. Альтернативная запись `nextval('my_seq'::text)` не создает зависимость.

Есть еще один тип системных идентификаторов, `xid`, представляющий идентификатор транзакции (сокращенно `xact`). Этот тип имеют системные столбцы `xmin` и `xmax`. Идентификаторы транзакций определяются 32-битными числами. В некоторых контекстах используется 64-битный вариант `xid8`. В отличие от `xid`, значения `xid8` увеличиваются строго монотонно и никогда не повторяются на протяжении всего существования кластера баз данных.

Третий тип идентификаторов, используемых в системе, — `cid`, идентификатор команды (`command identifier`). Этот тип данных имеют системные столбцы `ctid` и `stid`. Идентификаторы команд — это тоже 32-битные числа.

И наконец, последний тип системных идентификаторов — `tid`, идентификатор строки/кортежа (`tuple identifier`). Этот тип данных имеет системный столбец `ctid`. Идентификатор кортежа представляет собой пару (из номера блока и индекса записи в блоке), идентифицирующую физическое расположение строки в таблице.

## 5.20. Тип `pg_lsn`

Тип данных `pg_lsn` может применяться для хранения значения LSN (последовательный номер в журнале, Log Sequence Number), которое представляет собой указатель на позицию в журнале WAL. Этот тип содержит `XLogRecPtr` и является внутренним системным типом PostgreSQL.

Технически LSN — это 64-битное целое число, представляющее байтовое смещение в потоке журнала предзаписи. Он выводится в виде двух шестнадцатеричных чисел до 8 цифр каждое, через косую черту, например: `16/B374D848`. Тип `pg_lsn` поддерживает стандартные операторы сравнения, такие как `=` и `>`. Можно также вычесть один LSN из другого с помощью оператора `-`; результатом будет число байт между этими двумя позициями в журнале предзаписи. Также число байт можно добавлять и вычитать из LSN с помощью операторов `+(pg_lsn, numeric)` и `-(pg_lsn, numeric)` соответственно. Вычисленный

LSN должен находиться в допустимом для типа pg\_lsn диапазоне, то есть между 0/0 и FFFFFFFF/FFFFFFF.

## 5.21. Псевдотипы

В систему типов PostgreSQL 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 определенный тип данных SQL. Все существующие псевдотипы перечислены в таблице 5.27.

Таблица 5.27 – Псевдотипы

Имя	Описание
any	Указывает, что функция принимает любой вводимый тип данных
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
anymultirange	Указывает, что функция принимает любой мультидиапазонный тип данных
anycompatible	Указывает, что функция принимает любой тип данных и может автоматически приводить различные аргументы к общему типу данных
anycompatiblearray	Указывает, что функция принимает любой тип массива и может автоматически приводить различные аргументы к общему типу данных
anycompatiblenonarray	Указывает, что функция принимает любой тип, отличный от массива, и может автоматически приводить различные аргументы к общему типу данных
anycompatiblerange	Указывает, что функция принимает любой тип диапазонный данных и может автоматически приводить различные аргументы к общему типу данных
anycompatiblemultirange	Указывает, что функция принимает любой мультидиапазонный тип данных и может автоматически приводить различные аргументы к общему типу данных
cstring	Указывает, что функция принимает или возвращает строку в стиле C
internal	Указывает, что функция принимает или возвращает внутренний серверный тип данных
№ изменения: _____ Подпись отв. лица: _____ Дата внесения изм: _____	

Имя	Описание
language_handler	Обработчик процедурного языка объявляется как возвращающий тип language_handler
fdw_handler	Обработчик обертки сторонних данных объявляется как возвращающий тип fdw_handler
table_am_handler	Обработчик табличного метода доступа объявляется как возвращающий тип table_am_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 неопределенный 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 еще не распознанный тип

Функции, написанные на языке C (встроенные или динамически загружаемые), могут быть объявлены с параметрами или результатами любого из этих псевдотипов. Ответственность за безопасное поведение функции с аргументами таких типов ложится на разработчика функции.

Функции, написанные на процедурных языках, могут использовать псевдотипы, только если это позволяет соответствующий язык. В настоящее время большинство процедурных языков запрещают использовать псевдотипы в качестве типа аргумента и позволяют использовать для результатов только типы void и record (и trigger или event\_trigger, когда функция реализует триггер или событийный триггер). Некоторые языки также поддерживают полиморфные функции с полиморфными псевдотипами.

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, ее нельзя будет вызывать из SQL. Чтобы сохранить типобезопасность при таком ограничении, следуйте важному правилу: не создавайте функцию, возвращающую результат типа internal, если у нее нет ни одного аргумента internal.

## ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

ASCII	–	American standard code for information interchange
CIDR	–	Classless Internet Domain Routing
DDL	–	Data Definition Language
DML	–	Data Manipulation Language
DTD	–	Document Type Declaration
GIN	–	Generalized Inverted Index
IEEE	–	Institute of Electrical and Electronics Engineers
IP	–	Internet Protocol
ISO	–	International Organization for Standardization
JSON	–	JavaScript Object Notation
LSN	–	Log Sequence Number
MAC	–	Media Access Control
NaN	–	Not-a-Number
OID	–	Object Identifier
SQL	–	Structured Query Language
UTC	–	Universal Coordinated Time
XML	–	eXtensible Markup Language
АРМ	–	Автоматизированное рабочее место
БД	–	Базы данных
ОЗУ	–	Оперативное запоминающее устройство
ОС	–	Операционная система
СУБД	–	Система управления базами данных
ЭВМ	–	Электронно-вычислительная машина



## Лист регистрации изменений

[illegible]

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------