

Сборник заметок по СУБД PostgreSQL

Содержание

1	Установка PostgreSQL	4
1.1	Установка на Centos 7	4
1.2	Установка на MacOS X	6
2	Учебные наборы данных	7
3	Индексные структуры	9
4	Алгоритмы соединения	10
4.1	Соединение хешированием	10
4.2	Сортировка слиянием	11
5	Чтение планов выполнения	11
6	Короткие запросы	14
6.1	Выбор критериев фильтрации	15
6.2	Функциональные индексы	17
6.3	Составные индексы	18
6.4	Частичные индексы	19
6.5	Индексы и порядок соединения	20
6.6	Избегаем использования индексов	20
6.7	Создавать или не создавать	20
7	Длинные запросы и последовательное сканирование	21
7.1	Полусоединение	22
7.2	Антисоединение	22
7.3	Группировка	23
7.4	Использование операций над множествами	24
7.5	Избегаем многократного сканирования	26
7.6	Длинные запросы. Дополнительные приемы	28
7.6.1	Временные таблицы	28
7.6.2	Общие табличные выражения, CTE	29
7.6.3	Преставления	30
7.6.4	Материализованные представления	30
7.6.5	Секционирование	31
8	Оптимизация модификации данных	32

9	Суррогатные ключи	34
10	Функции	34
11	Динамический SQL	40
12	Более сложная фильтрация и поиск	42
12.1	Индексирование очень больших таблиц	43
13	Замечания по использованию различных баз данных для различных проектов	44
14	Ответвления от PostgreSQL	45
14.1	Базовые сведения о Greenplum	45
15	Хранилища данных	46
16	Основные ошибки при работе с PostgreSQL	47
17	Файл подкачки	48
18	Первичные и внешние ключи	48
19	Буферный кеш в PostgreSQL	49
20	План выполнения запроса и команда EXPLAIN	51
20.1	Обнаружение проблем в планах выполнения запросов	52
21	Сброс пароля для psql и pgAdmin4	53
22	Создание резервной копии базы данных	53
23	Логический порядок обработки инструкции SELECT	54
24	Смена схемы базы данных	55
25	Секционирование	55
26	Транзакции и управление параллельным доступом	55
27	Уровня изоляции транзакций	56
28	Создание таблицы	57
28.1	Базовые синтаксис создания таблицы	57
28.2	Создать таблицу по образу другой таблицы	58
29	Подзапросы	58
29.1	Латеральные подзапросы	58
29.2	Дополнительные средства группировки	60
29.3	Дополнительные виды агрегирования	61
30	Копирование данных между файлом и таблицей	63

31 Обновление записей. Команда UPDATE	63
32 Общие табличные выражения	64
32.1 Конструкция WITH	64
32.2 Конструкция WITH RECURSIVE	66
32.3 Изменение данных в WITH	69
33 Оконные функции	69
33.1 Определение окна	69
33.2 Фраза WINDOWS	71
33.3 Использование оконных функций	71
34 Продвинутое методы работы с SQL	74
34.1 Выборка первых записей	74
35 Пользовательские типы данных	76
36 Работа со строками и регулярные выражения	77
37 Программирование на PL/pgSQL	80
37.1 Примеры работы с функциями	80
37.2 Итерирование	81
37.3 Возврат из функции	83
37.4 Обработка исключений	85
37.5 Динамический SQL	86
38 Массивы	88
38.1 Функции и операторы массивов	90
38.2 Доступ к элементам массива и их модификация	90
38.3 Индексирование массивов	92
39 Хранилище ключей и значений	92
39.1 Индексирование hstore	94
40 Структура данных JSON	95
40.1 JSON и XML	95
40.2 Типы данных JSON в PostgreSQL	95
40.3 Доступ к объектам типа JSON и их модификация	95
40.4 Индексирование JSON-документов	96
41 PostgreSQL и Python	97
41.1 Простой пример работы с PostgreSQL через библиотеки SQLAlchemy и psycopg2	97
42 Приемы работы в pgAdmin 4	101
43 Приемы работы в psql	101
43.1 Конфигурационный файл	101
43.2 Метакоманды psql	101

43.3 Примеры использования	102
44 Специальные функции и операторы	104
44.1 Предикаты ANY, ALL	104
44.2 Оператор конкатенации	104
44.3 Диапазонные операторы	104
44.4 Функции, генерирующие ряды значений	105
44.5 Функции для работы с массивами	107
44.6 Операторы и функции даты/времени	107
44.7 Выполнение динамически формируемых команд	107
44.8 Функции и операторы JSON	109
45 Приемы работы с JSONPATH	112
45.1 Элементы языка путей SQL/JSON	112
45.2 Элементы языка JQuery	114
45.3 Примеры конструкций с JSONPATH	114
46 Наиболее полезные команды PostgreSQL	116
46.1 Команда VACUUM	116
47 Обработка временных рядов в TimescaleDB	116
47.1 Установка TimescaleDB	117
47.1.1 На Linux	118
47.1.2 На MacOS	118
47.1.3 Сборка из исходников	118
Список литературы	118
Список листингов	118

1. Установка PostgreSQL

1.1. Установка на Centos 7

На ОС Centos Postgres может быть установлен с использованием репозитория «из коробки», однако доступная версия Postgres (для Centos 7 это Postgres 9.2) считается устаревшей.

Прежде чем приступить к настройке нового репозитория, следует исключить из поиска пакетов `postgresql` из репозитория Centos-Base. Иначе зависимости могут разрешаться для `postgresql`, предоставляемого базовым репозиторием. Для этого нужно в файле `/etc/yum.repos.d/CentOS-Base.repo` в разделы `[base]` и `[updates]` добавить строку `exclude=postgresql*`. В итоге должно получиться так

`/etc/yum.repos.d/CentOS-Base.repo`

```
[base]
name=CentOS-$releasever - Base
mirrorlist=http://mirrorlist...
gpgcheck=1
gpgkey=file:///...
exclude=postgresql*  # <- NB
```

```
[updates]
name=CentOS-$releasever - Updates
mirrorlist=...
gpgcheck=1
gpgkey=file:///...
exclude=postgresql* # <- NB
...
```

Устанавливаем пакет конфигурации репозитория, используя официальный репозиторий PostgreSQL для CentOS

```
$ sudo yum install https://download.postgresql.org/pub/repos/yum/reposms/EL-7-x86_64/pgdg-redhat-repo-latest.noarch.rpm
```

Устанавливаем PostgreSQL-сервер (несколько раз нужно будет подтвердить, что мы разрешаем импорт GPG-ключей)

```
# PostgreSQL 12
$ sudo yum install postgresql12-server
```

Прежде чем можно будет создать свою базу данных, нужно создать кластер баз данных PostgreSQL. Кластер баз данных – это набор баз данных, управляемых одним экземпляром сервера. Создание кластера баз данных сводится к созданию каталогов, в которые будут помещены данные БД и баз данных `template1` и `postgres`.

База данных `template1` нужна для создания новой базы данных. Все, что в ней находится будет помещено в новую базу данных при ее создании. А `postgres` это база данных по умолчанию.

Создать PostgreSQL-кластер базы данных можно так

```
$ sudo /usr/pgsql-12/bin/postgresql-12-setup initdb
Initializing database ... OK
```

Запускаем PostgreSQL с помощью утилиты `systemctl`

```
$ sudo systemctl start postgresql-12
$ sudo systemctl enable postgresql-12
```

Во время установки была создана роль `postgres`. Чтобы использовать ее для запуска утилиты `psql` следует набрать следующее

```
$ sudo -i -u postgres
```

Чтобы не переключаться между ролями можно сделать так

```
$ sudo -u postgres psql
```

На текущий момент у нас есть только роль `postgres`. Чтобы создать новую роль нужно набрать в командной оболочке

```
$ sudo -u postgres creatuser --interactive # sammy
```

PostgreSQL предполагает, что у роли есть база данных с тем же именем. То есть если роль пользователя называется `sammy`, то эта роль попытается подключиться к базе данных с тем же именем `sammy`

```
$ sudo -u postgres createdb sammy
```

Если имя роли PostgreSQL не совпадает с именем пользователя Linux, его можно добавить так

```
$ sudo adduser sammy
```

а потом

```
$ sudo -u sammy psql
```

Если при создании базы данных из sql-файла возникают ошибки доступа к данным, то для начала нужно проверить права роли, из-под которой создается база данных

```
# Запускаем psql из-под роли, у которой есть права на изменение ролей
$ sudo -u postgres psql
postgres# \du -- выведет список ролей
```

Если окажется, что интересующая роль не обладает нужными правами, то следует их добавить

```
# alter user role_name superuser;
# alter user role_name createrole;
# alter user role_name createdb;
```

После этого можно запускать создание базы данных из sql-файла

```
# Если файл большой можно запустить создание в фоне (Ъ в конце команды)
$ psql -f database_maker.sql > psql.log 2>&1 &
# Следим за ходом выполнения
$ tail -f psql.log
```

Проверить статус сервера можно так

```
$ sudo systemctl status postgresql-12
```

а остановить так

```
$ sudo systemctl stop postgresql-12
```

1.2. Установка на MacOS X

Для работы с PostgreSQL на ОС MacOS X проще всего установить специальный клиент <https://postgresapp.com/>. Нужно скачать образ приложения Postgres.app и перенести его в директорию Applications. Затем нужно двойным кликом запустить приложение. После чего появится окно приложения (рис. 1).

Нажимаем кнопку *Initialize* и запускаем сервер (рис. 2). Убедиться в том, что сервер запущен можно с помощью утилиты `pg_ctl`

```
$ pg_ctl -D ~/Library/Application\ Support/Postgres/var-15 status
pg_ctl: server is running (PID: 48717)
/Applications/Postgres.app/Contents/Versions/15/bin/postgres "-D" "/Users/leor.finkelberg/Library/Application Support/Postgres/var-15" "-p" "5432"
```

И так

```
$ ps aux | grep -inE postgres
...
581:leor.finkelberg 48717  0,0  0,2 34344392 15460  ??  Ss   1:42   0:00.08 /Applications
/Postgres.app/Contents/Versions/15/bin/postgres -D /Users/leor.finkelberg/Library/
Application Support/Postgres/var-15 -p 5432
...
```

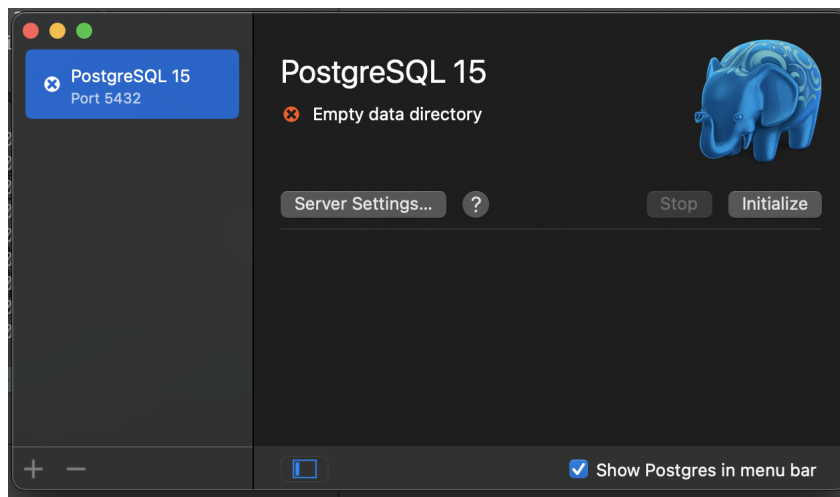


Рис. 1. Окно приложения Postgres для MacOS X

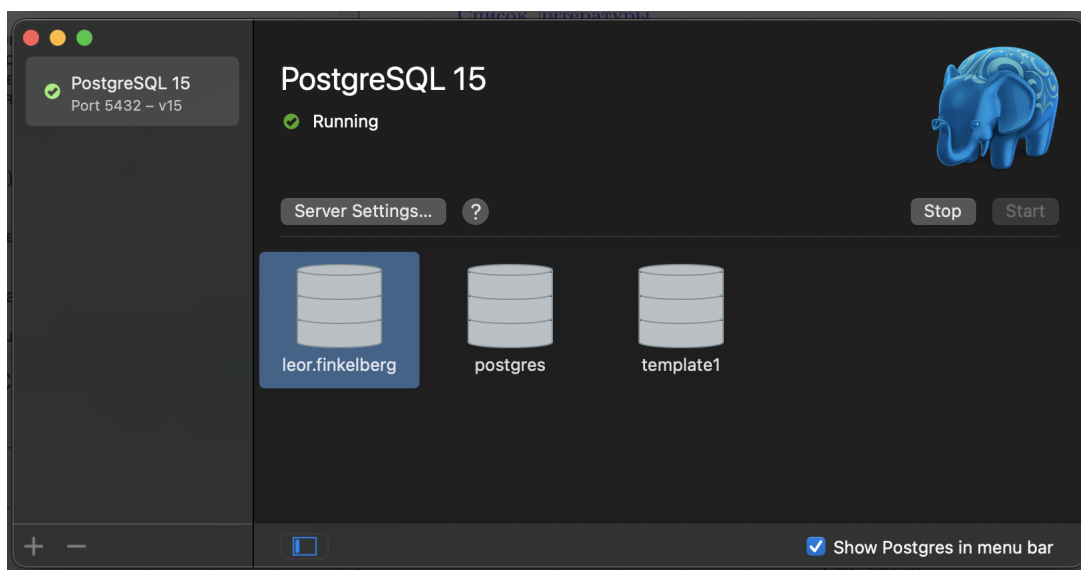


Рис. 2. Запуск сервера PostgreSQL на MacOS X

2. Учебные наборы данных

Очень приличных объемов базу данных можно скачать здесь https://drive.google.com/drive/folders/13F7M80Kf_somnjb-mTYAnh1hW1Y_g4kJ

Например можно скачать sql-файл создания базы данных

```
$ unzip postgres_air.sql.zip
# запускаем в фоне
$ psql -f postgres_air.sql > psql.log 2>&1 &
# следим за ходом выполнения
$ tail -f psql.log
```

Если создание базы данных прошло успешно, то потом нужно добавить путь к схеме базы данных

```
SET search_path TO postgres_air;
```

Проверить, что все таблицы схемы видны можно сделать так

```
# \dt+
```

При работе с этой базой данных полезно сразу создать кое-какие индексы

```
CREATE INDEX flight_departure_airport ON flight(departure_airport);
CREATE INDEX flight_scheduled_departure ON flight(scheduled_departure);
CREATE INDEX flight_update_ts ON flight(update_ts);
CREATE INDEX booking_leg_booking_id ON booking_leg(booking_id);
CREATE INDEX booking_leg_update_ts ON booking_leg(update_ts);
CREATE INDEX account_last_name ON account(last_name);
```

Чтобы получить результаты запросов, PostgreSQL выполняет следующие шаги [2, стр. 31]:

- компилирует и преобразует инструкцию SQL в выражение, состоящее из логических операций высокого уровня, называемое логический план,
- оптимизирует логический план и превращает его в план выполнения,
- выполняет (интерпретирует) план и возвращает результаты.

Замечание

Разные серверы баз данных могут интерпретировать один и тот же запрос по-разному

Для оценки эффективности запроса оптимизатор использует внутренние показатели, основанные на объеме вычислительных ресурсов, необходимых для выполнения запроса или отдельной физической операции в рамках плана. Наиболее важными ресурсами являются те, что влияют на время выполнения, а именно циклы процессора и количество операций ввода-вывода (чтение и запись дисковых блоков).

Другие ресурсы, такие как память или дисковое пространство, тоже косвенно влияют на время выполнения; например, количество доступной памяти будет влиять на соотношение циклов процессора и количество операций ввода-вывода.

Эти два основных показателя, *циклы процессора* и *количество операций ввода-вывода*, напрямую не сопоставимы. Однако для сравнения планов выполнения запросов оптимизатор объединяет их в одну функцию стоимости: чем ниже стоимость, тем лучше план.

Любой файл, используемый для объектов базы данных, делится на блоки одинаковой длины: по умолчанию PostgreSQL использует блоки по 8192 байта каждый. *Блок* – это единица, которая передается между жестким диском и оперативной памятью, а количество операций ввода-вывода, необходимое для выполнения любого доступа к данным, равно количеству блоков, которые читаются и записываются.

Строка может быть вставлена в любой блок, в котором достаточно свободного места, без специального упорядочивания.

Все реляционные базы данных, включая PostgreSQL, позволяют создавать дополнительные, избыточные структуры, значительно ускоряя доступ к данным по сравнению с простым последовательным чтением.

Эти дополнительные структуры называются *индексами*. Во-первых, индекс – «избыточные» объекты базы данных; они не хранят никакой дополнительной информации, которую нельзя найти в исходной таблице. Во-вторых, индексы предоставляют дополнительные пути доступа к данным; они позволяют определить, какие значения хранятся в строках таблицы, без необходимости чтения самой таблицы – так работает доступ на основе индексов.

Если условие (или условия) фильтрации поддерживаются индексом в таблице, индекс можно использовать для доступа к данным из этой таблицы. Алгоритм извлекает список указателей на блоки, содержащие строки со значениями, удовлетворяющими условию фильтрации, и только эти блоки читаются из таблицы.

Основная структура данных таблицы – это куча, то есть строки хранятся неупорядоченно. Есть две отдельные *физические операции*, используемые PostgreSQL для получения *строк с помощью индексов*:

- *индексное сканирование* (**index scan**); при индексном сканировании движок базы данных считывает одну за другой все записи индекса, которые удовлетворяют условию фильтрации, и в этом же порядке извлекает блоки.
- *сканирование по битовой карте* (**bitmap heap scan**): поскольку базовая таблица представляет собой кучу, несколько записей индекса могут указывать на один и тот же блок; чтобы избежать многократного чтения одного и того же блока, в PostgreSQL реализована операция сканирования по битовой карте, которая создает битовую карту блоков, содержащих необходимые строки. Потом все строки в этих блоках фильтруются.

Стоимостная модель этого алгоритма намного сложнее. Неформально ее можно описать так: при малых значениях селективности, скорее всего, все строки, удовлетворяющие условиям фильтрации, будут располагаться в разных блоках, и, следовательно, стоимость будет пропорциональна количеству возвращаемых строк. Для больших значений селективности количество обрабатываемых блоков приближается к общему количеству блоков. В последнем случае стоимость становится выше, чем стоимость полного сканирования, поскольку для доступа к индексу необходимы ресурсы [2, стр. 43].

Если сканируется только индекс, то для малых значений селективности стоимость примерно пропорциональна количеству возвращаемых строк. При больших значениях селективности алгоритм выполняет (почти) полный просмотр индекса. Стоимость просмотра индекса обычно ниже, чем стоимость полного просмотра таблицы, потому что *индекс содержит меньше данных*.

3. Индексные структуры

Для *меньших значений селективности* предпочтительнее доступ на основе *индексов*, а *полное сканирование* лучше подходит для *больших значений селективности* [2, стр. 44].

Замечание

Индексы эффективны, когда подмножество строк таблицы составляет небольшую долю от общего числа (то есть при низких значениях селективности)

Структура данных называется индексом, если:

- это избыточная структура данных: избыточность означает, что индекс можно удалить без потери данных и восстановить по информации, хранящейся где-то еще,
- она невидима для приложения: невидимость означает, что приложение не может узнать о наличии индекса или о его отсутствии. Иначе говоря, любой запрос дает те же результаты как с индексом, так и без него,
- она предназначена для ускорения выбора данных по определенным критериям: индекс создается в надежде, что это улучшит производительность конкретного запроса или нескольких запросов.

Улучшение производительности не происходит бесплатно. Поскольку индекс избыточен, он должен обновляться при обновлении данных таблицы. Это приводит к накладным расходам для операции обновления, которыми иногда нельзя пренебречь. В частности, индексы PostgreSQL могут оказывать большое влияние на операцию очистки (**VACUUM**). Однако эти расходы как правило

переоценивают. Современные высокопроизводительные СУБД используют алгоритмы, которые снижают стоимость обновления индексов, поэтому несколько индексов в таблице – обычное дело.

Хотя индексные структуры могут значительно различаться для разных типов индексов, ускорение всегда достигается за счет *быстрой проверки некоторых условий фильтрации*, указанных в запросе. Такие условия фильтрации устанавливают определенные ограничения на атрибуты таблицы.

Ни один алгоритм поиска не может найти индексный ключ среди N различных ключей быстрее, чем за время $\log N$. Такая производительность достигается с помощью двоичного поиска по упорядоченному списку или с помощью бинарного дерева. Однако стоимость обновлений (например, вставки новых ключей) может быть очень высока как для упорядоченных списков, так и для двоичных деревьев: вставка одной записи может привести к полной реструктуризации. Это делает обе структуры непригодными для хранения.

Напротив, *B-деревья* можно изменять без значительных накладных расходов. Когда вы вставляете запись, *реструктуризация ограничена одним блоком*. Даже в худшем случае количество измененных блоков не будет превышать глубины дерева [2, стр. 49].

Если каждый блок содержит f указателей, то количество блоков на каждом уровне в f раз больше, чем в предыдущем. Следовательно, глубина дерева, содержащего N записей, равна $\log_f N = \frac{\log N}{\log f}$. Эта формула дает количество обращений к диску, необходимое для поиска по одному ключу. Размер блока в PostgreSQL составляет 8 Кбайт. В такой блок помещается десятки индексных записей. следовательно, индекс с 6-7 уровнями может вместить миллиарды записей.

4. Алгоритмы соединения

Для больших таблиц более эффективны алгоритмы соединения хешированием и сортировки слиянием.

4.1. Соединение хешированием

Результат естественного соединения состоит из пар строк из таблиц R и S , которые имеют равные значения атрибутов, по которым выполняется соединение. Идея алгоритма соединения хешированием проста: если значения равны, то равны и хеш-значения.

Базовая версия алгоритма соединения хешированием включает две фазы:

- на этапе построения хеш-таблицы все кортежи (строки) левой таблицы R сохраняются в корзинах согласно значениям хеш-функции,
- на этапе проверки каждая строка правой таблицы S направляется в соответствующую ей корзину. Если подходящие строки таблицы R находятся в этой корзине, порождаются выходные строки.

Алгоритм на основе хеширования значительно лучше вложенных циклов подходит для больших таблиц и большого количества различных значений атрибута соединения.

Базовый алгоритм соединения хешированием работает, если все корзины, созданные на этапе построения хеш-таблицы, помещаются в оперативную память.

Другой вариант, называемый *гибридным соединением хешированием*, соединяет таблицы, которые не могут поместиться в оперативной памяти. Гибридное соединение хешированием разделяет обе таблицы так, чтобы отдельные разделы помещались в память, а затем выполняет базовый алгоритм для каждой пары соответствующих разделов. Стоимость гибридного соединения

хешированием выше, потому что разделы временно хранятся на жестком диске и обе таблицы сканируются дважды.

4.2. Сортировка слиянием

На первом этапе алгоритма обе входные таблицы сортируются в порядке возрастания по атрибутам соединения.

После того как таблицы правильно упорядочены, на этапе слияния обе таблицы сканируются один раз и для каждого значения атрибута соединения вычисляется декартово произведение строк, содержащих это значение.

Алгоритм сортировки слиянием особенно эффективен, если одна или обе входные таблицы уже отсортированы.

5. Чтение планов выполнения

Рассмотрим запрос

```
SELECT
  f.flight_no, -- атрибут группировки
  f.actual_departure, -- атрибут группировки
  count(passenger_id) passengers
FROM
  flight f
  JOIN booking_leg bl ON bl.flight_id = f.flight_id
  JOIN passenger p ON p.booking_id = bl.booking_id
WHERE
  f.departure_airport = 'JFK'
  AND f.arrival_airport = 'ORD'
  AND f.actual_departure BETWEEN '2020-08-14' AND '2020-08-15'
GROUP BY f.flight_no, f.actual_departure; -- атрибуты группировки должны быть в списке выборки
```

Чтобы получить план выполнения запроса, используется команда `EXPLAIN query`

```
+-----+
|
|                                QUERY PLAN                                |
+-----+-----+
| Group  (cost=501006.78..501007.75 rows=1 width=12)                      |
|
|   Group Key: f.flight_no, f.actual_departure                            |
|
|   ->  Gather Merge  (cost=501006.78..501007.74 rows=2 width=12)         |
|
|         Workers Planned: 2                                              |
|
|       ->  Group  (cost=500006.75..500007.48 rows=1 width=12)            |
|
```

```

|
|      Group Key: f.flight_no, f.actual_departure
|
|
|      -> Sort (cost=500006.75..500007.00 rows=97 width=12)
|
|
|      Sort Key: f.flight_no, f.actual_departure
|
|
|      -> Parallel Hash Join (cost=234897.56..500003.55 rows=97 width=12)
|
|
|      Hash Cond: (p.booking_id = bl.booking_id)
|
|      -> Parallel Seq Scan on passenger p (cost=0.00..239617.15 rows
|      =6796915 width=4)
|
|      -> Parallel Hash (cost=234897.42..234897.42 rows=11 width=16)
|
|      -> Parallel Hash Join (cost=9198.22..234897.42 rows=11 width
|      =16)
|
|      Hash Cond: (bl.flight_id = f.flight_id)
|
|      -> Parallel Seq Scan on booking_leg bl (cost
|      =0.00..206127.90 rows=7455690 width=8)
|
|      -> Parallel Hash (cost=9198.20..9198.20 rows=1 width
|      =16)
|
|      -> Parallel Bitmap Heap Scan on flight f (cost
|      =112.43..9198.20 rows=1 width=16)
|
|      Recheck Cond: (departure_airport = 'JFK'::
|      bpchar)
|
|      Filter: ((actual_departure >= '2020-08-14
|      00:00:00+03'::timestamp with time zone) AND (actual_departure <= '2020-08-15 00:00:0
|      0+03'::timestamp with time zone) AND (arrival_airport = 'ORD'::bpchar)) |
|      -> Bitmap Index Scan on
|      flight_departure_airport (cost=0.00..112.43 rows=10134 width=0)
|
|      Index Cond: (departure_airport = 'JFK
|      '::bpchar)
|
+-----+
+-----+

```

План содержит:

- оценки стоимости,
- ожидаемое количество выходных строк,
- ожидаемый средний размер («ширину») строк.

Значения стоимости включают в себя совокупную стоимость:

- первая показывает стоимость, необходимую для создания первой строки вывода,
- в то время как вторая оценивает *полную стоимость получения всего результата*.

Важно подчеркнуть, что все эти цифры являются приблизительными. План выполнения представляется в виде дерева физических операций. В этом дереве узлы представляют операции, а стрелки указывают на операнды. Он показывает каждый узел дерева в отдельной строке, начинающейся с символа `->`, а глубина узла представлена отступом. Поддережья помещаются после своего родительского узла. Некоторые операции представлены двумя строками.

Выполнение плана начинается с листьев и заканчивается у корня. Это значит, что *операция*, которая выполняется *первой*, находится в строке с *наибольшим отступом* [2, стр. 60]. Как только операция создает выходную строку, эта строка передается следующей операции. Таким образом, не нужно хранить промежуточные результаты между операциями.

Итак, выполнение начинается с последней строки, а таблица `flight` читается с помощью индекса по столбцу `departure_airport`. Поскольку к таблице применено несколько фильтров, а индексом поддерживается только одно из условий фильтрации, PostgreSQL выполняет операцию `index bitmap scan`.

Движок обращается к индексу и составляет битовую карту блоков, которые могут содержать необходимые записи. Затем он считывает фактические блоки из базы данных с помощью операции `bit map heap scan`, и для каждой записи, извлеченной из базы данных, повторно проверяет (`recheck`), что найденные в индексе строки являются текущими, и применяет фильтрацию (`filter`) для дополнительных условий по столбцам `arrival_airport` и `schedule_departure`, для которых у нас нет индексов.

Результат соединяется с таблицей `booking_leg`. PostgreSQL использует последовательное сканирование для доступа к этой таблице и алгоритм соединения хешированием по условию

```
bl.flight_id = f.flight_id.
```

Затем таблица `passenger` читается последовательным сканированием (поскольку на ней нет никаких индексов), и снова используется алгоритм соединения хешированием по условию

```
p.booking_id = bl.booking_id.
```

Последняя выполняемая операция – группировка и вычисление агрегатной функции `sum`. После сортировки выясняется, что критериям поиска удовлетворяет только один рейс. Таким образом, нет необходимости использовать какие-либо алгоритмы группировки, и выполняется подсчет всех пассажиров на найденном рейсе.

Оптимизатор выполняет два вида преобразований:

- заменяет *логические операции* соответствующими *физическими алгоритмами выполнения*
- и (возможно) изменяет структуру логического выражения, изменяя порядок, в котором выполняются логические операции.

Первый шаг – это переписывание запроса. На этом этапе PostgreSQL улучшает код, раскрывая подзапросы, заменяя представления их текстовым видом и т.д.

Замечание

Учебники по SQL, рассказывая о концепции представления, часто говорят, что «представления можно использовать как таблицы», и это вводит в заблуждение. В большинстве случаев *представления* заменяют их *исходным кодом* [2, стр. 62]

Следующим шагом после переписывания запроса выполняется то, что мы обычно и называем оптимизацией. Она включает:

- определение возможных порядков операций,
- определение возможных алгоритмов выполнения для каждой операции,

- сравнение стоимости разных планов,
- выбор оптимального плана выполнения.

Замечание

Многие разработчики SQL предполагают, что PostgreSQL выполняет запросы, обращаясь к таблицам в том порядке, в котором они указаны в предложении **FROM**, и соединяя их в том же порядке. Однако в большинстве случаев порядок соединения НЕ сохраняется [2, стр. 62]

Одна инструкция SQL может выполняться разными способами, используя разные планы исполнения. На самом деле могут быть сотни, тысячи или даже миллионы возможных способов выполнения одной команды!

Планы могут отличаться:

- порядком операций,
- алгоритмами, используемыми для соединений и других операций (такими как вложенные циклы или соединения хешированием),
- методами получения данных (такими как индексный доступ или полное сканирование).

Говоря формально, оптимизатор находит лучший план, вычисляя стоимости всех возможных планов, а затем сравнивает их. К счастью PostgreSQL не проверяет все возможные планы. Оптимизатор строит оптимальный план, начиная с самых маленьких поддеревьев (то есть с доступа к отдельным таблицам), и постепенно создает более сложные поддеревья, включающие больше операций, выполняя на каждом шаге лишь несколько сравнений стоимости.

Алгоритм является исчерпывающим: он строит оптимальный план, несмотря на то что значительная часть возможных планов не рассматривается. Тем не менее количество рассмотренных планов может быть огромным. Эвристики исключают те части пространства плана, которые вряд ли будут содержать оптимальные планы, уменьшая тем самым количество проверяемых вариантов. Хотя это и помогает оптимизатору быстрее выбирать план выполнения, но может отрицательно сказаться на производительности: существует риск, что самый подходящий план выполнения будет случайно исключен перед сравнением стоимостей.

Хотя эвристики могут исключить оптимальный план, алгоритм строит лучший план из оставшихся.

Стоимости алгоритмов измеряются [2, стр. 63]:

- Количеством операций ввода-вывода
- и циклов процессора.

Стоимость каждого плана выполнения зависит от:

- формул стоимости алгоритмов, используемых в плане,
- статистических данных по таблицам и индексам, включая распределение значений,
- системных настроек (параметров и предпочтений), таких как `join_collapse_limit` или `cpu_index_tuple_cost`.

6. Короткие запросы

Во-первых, тип запроса не имеет ничего общего с длиной SQL-запроса. Пример длинного запроса

Длинный запрос

```
SELECT
  d.airport_code AS departure_airport,
  a.airport_code AS arrival_airport
FROM
  airport a,
  airport d;
```

Короткий запрос

Короткий запрос

```
SELECT
  f.fligth_no,
  f.scheduled_departure,
  boarding_time,
  p.last_name,
  p.first_name,
  bp.update_ts AS pass_issued,
  ff.level
FROM
  flight f
  JOIN booking_leb bl ON bl.flight_id = f.flight_id
  JOIN passenger p ON p.booking_id = bl.booking_id
  JOIN account a ON a.account_id = p.account_id
  JOIN boarding_pass bp ON bp.passenger_id = p.passenger_id
  LEFT OUTER JOIN frequent_flyer ff ON ff.frequent_flyer_id = a.frequent_flyer_id
WHERE f.departure_airport = 'JFK'
  AND f.arrival_airport = 'ORD'
  AND f.scheduled_departure BETWEEN '2020-08-05' AND '2020-08-07';
```

Во-вторых, тип запроса не определяется размером результирующего множества

Длинный запрос, выводящий одну строку

```
SELECT
  avg(flight_length),
  avg(passengers)
FROM
  (
    SELECT
      flight_no,
      scheduled_arrival - scheduled_departure AS flight_length,
      count(passenger_id) passengers
    FROM
      flight f
      JOIN booking_leg bl ON bl.flight_id = f.flight_id
      JOIN passenger p ON p.booking_id = bl.booking_id
    GROUP BY 1, 2
  ) a;
```

6.1. Выбор критериев фильтрации

Когда мы оптимизируем короткий запрос, мы знаем, что в конечном счете мы выбираем относительно небольшое количество записей. Это означает, что цель оптимизации – уменьшить размер результирующего множества как можно раньше. В плане выполнения коротких запросов не должно быть сканирований больших таблиц. Однако для небольших таблиц полное сканирование может применяться.

Нельзя быстро выбрать подмножество записей из таблицы, если нет индекса, поддерживающего соответствующий поиск. Вот почему для ускорения *коротких запросов* требуются *индексы*.

Концепцию селективности запроса можно применить и к индексам: чем меньше количество записей соответствует одному значению индекса, тем ниже значение его селективности.

Нам не нужно создавать индексы с высокой селективностью. Для извлечения данных на основе индекса в этом случае потребуется *больше* времени, чем при *последовательном сканировании*.

Поскольку оптимизатор PostgreSQL заранее определяет стоимость каждого метода доступа, этот индекс *никогда не будет использоваться*, и производительность не пострадает. Однако нежелательно добавлять объект базы данных, который требует места для хранения и дополнительное время для обновления, но не дает никаких преимуществ [2, стр. 69].

В таблице базы данных может быть несколько индексов по разным столбцам, каждый со своей селективностью. Лучшая производительность для короткого запроса достигается, когда используются *индексы с самой низкой селективностью*.

Если самый строгий критерий не проиндексирован, то план выполнения может быть неоптимальным и, вероятно, потребует, дополнительный индекс.

Чем *ниже селективность* индекса, тем *быстрее* выполняется поиск. Таким образом, наиболее эффективный индексы – *уникальные*.

Индекс *уникален*, если каждому *индексированному значению* соответствует ровно *одна строка* в таблице.

Есть несколько разных способов создать уникальный индекс. Во-первых, PostgreSQL автоматически создает *уникальный индекс* для поддержки каждого *первичного ключа* или *ограничения уникальности* в таблице.

Создать первичный ключ, внешний ключ и ограничение уникальности после создания таблицы можно так

```
ALTER TABLE booking
ADD CONSTRAINT booking_pkey PRIMARY KEY (booking_id);

ALTER TABLE booking
ADD CONSTRAINT booking_booking_ref_key UNIQUE (booking_ref);

ALTER TABLE flight
ADD CONSTRAINT arrival_airport_fk FOREIGN KEY (arrival_airport)
REFERENCES airport (airport_code);
```

Замечание

Первичный ключ (PK), внешний ключ (FK) и конечно ограничение уникальности (UNIQUE) – это все ОГРАНИЧЕНИЯ

Первичный ключ не обязан быть числовым и, более того, не обязан определяться только для одного атрибута. Можно определить первичный ключ и как комбинацию нескольких атрибутов.

Первичный ключ должен удовлетворять двум условиям:

1. комбинация должна быть уникальна (UNIQUE),
2. и определена (NOT NULL) для всех участвующих атрибутов.

Отличие *уникальных ограничений* в том, что они допускают неопределенные значения.

Таблица может иметь один первичный ключ (хотя это и не обязательно) и сколько угодно ограничений уникальности. Любое ограничение уникальности, не допускающее неопределенных

значений, может быть выбрано в качестве первичного ключа для таблицы; таким образом, нельзя формально определить лучшего кандидата на роль первичного ключа.

Уникальный индекс можно создать и без формального определения ограничения уникальности. Нужно только добавить ключевое слово `UNIQUE`

```
CREATE UNIQUE INDEX account_freq_flyer ON account(frequent_flyer_id);
```

Внешний ключ – это ограничение ссылочной целостности. Таблица, в которой определен внешний ключ, называется *дочерней*, а таблица, на которую он ссылается, – *родительской*. Ограничение внешнего ключа гарантирует, что для каждого определенного (не `NULL`) значения из дочерней таблицы в родительской таблице есть совпадающие с ним уникальное значение.

Алгоритм соединения вложенным циклом также можно сочетать с доступом к данным на основе индекса, если у таблицы *S* (левая таблица, вложенный цикл) есть *индекс* по атрибутам, используемым в условиях соединения. Для естественных соединений внутренний цикл алгоритма с индексным доступом сокращается до перебора *нескольких* строк таблицы *S* *для каждой* строки из таблицы *R*. Внутренний цикл может даже полностью исчезнуть, если индекс по таблице *S* уникален, например если *атрибут соединения* является *первичным ключом*.

Замечание

Вложенные циклы *всегда эффективны (!)*, когда соединения выполняются по связке «первичный ключ – внешний ключ», но при условии, что внешний ключ проиндексирован [2, стр. 72]

Индекс по столбцу с *ограничением внешнего ключа* имеет смысл создавать только, если *количество различных значений достаточно большое*. Индексы с высокой селективностью вряд ли пригодятся [2, стр. 72].

Например, для таблицы `flight` создавать индекс по атрибуту `aircraft_code` смысла нет, так как у этого столбца всего 12 различных значений, поэтому индекс по нему использоваться не будет.

Замечание

На *маленьких* таблицах *индексы не эффективны!* [2, стр. 73] То есть на маленьких таблицах коэффициент селективности не будет настолько низким, чтобы индексы были эффективны

6.2. Функциональные индексы

В-деревья поддерживают поиск по отношениям `=`, `<`, `>` и `BETWEEN`.

В-деревья по преобразованному атрибуту использовать нельзя. Пусть имеется индекс

```
CREATE INDEX account_last_name ON account (last_name);
```

Следующий запрос не сможет воспользоваться этим индексом!

```
SELECT * FROM account WHERE lower(last_name) = 'daniels';
```

Чтобы решить проблему нужно создать *функциональный* индекс

```
CREATE INDEX account_last_name_lower ON account(lower(last_name));
```

Приведение типа тоже рассматривается как преобразование атрибута. Например, даже если на атрибуте есть индекс, то приведение типа запрещает использование индекса.

Замечание

Индексы не используются для преобразованных атрибутов. Нужно создавать функциональные индексы

Один из частых случаев – поиск изменений, внесенных в таблицу за сегодняшний день. В девяти случаях из десяти это условие формулируется как `update_ts::date = CURRENT_DATE`, фактически лишаясь индекса по столбцу `update_ts`. Чтобы воспользоваться индексом, критерий следует записать как

```
update_ts >= CURRENT_DATE
```

Рассмотрим еще один пример, когда преобразование столбца часто остается незамечным.

Функция `coalesce` в PostgreSQL позволяет нам использовать другое значение, когда первый аргумент неопределен. Таким образом, `coalesce(actual_departure, scheduled_departure)` вернет `actual_departure`, если это значение отлично от `NULL`, и `scheduled_departure` в противном случае

```
SELECT * FROM flight
WHERE coalesce(actual_departure, scheduled_departure)
      BETWEEN '2020-08-17' AND '2020-08-18';
```

Однако, в данном случае индексы для этих атрибутов использоваться не будут, потому что функция `coalesce` изменяет значение атрибута.

Можно создать функциональный индекс, а можно переписать запрос

```
SELECT * FROM flight
WHERE coalesce(actual_departure, scheduled_departure)
      OR (actual_departure IS NULL
          AND scheduled_departure BETWEEN '2020-08-17' AND '2020-08-18');
```

Postgres может использовать результаты поиска по нескольким индексам, создавая в оперативной памяти битовую карту блоков с подходящими записями, а затем применяя к ним операторы «или» либо «и». В результате остаются только блоки, удовлетворяющие всем критериям поиска.

6.3. Составные индексы

Использование операторов «и» и «или» для нескольких битовых карт – очень эффективный механизм применения нескольких фильтров, но не единственный.

Можно создать составной индекс по всем трем столбцам

```
CREATE INDEX flight_departure_arr_sched_dep
ON flight (
    departure_airport,
    arrival_airport,
    scheduled_departure
);
```

Новый составной индекс будет поддерживать поиск по:

- `departure_airport`,
- `departure_airport, arrival_airport`,
- `departure_airport, arrival_airport, scheduled_departure`.

Однако он не поддерживает поиск по:

- arrival_airport,
- arrival_airport, scheduled_departure.

То есть

```
X Y Z
=====
X      +
X Y    +
X Y Z  +
=====
. Y    -
. Y Z  -
. . Z  -
```

Таким образом, при создании составного индекса недостаточно решить, какие столбцы в него включить; необходимо также *учитывать их порядок* [2, стр. 82].

Когда все столбцы из запроса включены в составной индекс, их можно получить без доступа к таблице. Этот метод получения данных называется *сканированием только индекса*.

6.4. Частичные индексы

Частичные индексы строятся на подмножестве таблицы, определяемом предложением **WHERE** команды **CREATE INDEX**.

Частичный индекс может быть эффективен, если применяется к категории с наименьшей мощностью. Например, столбец **status** в таблице **flight** имеет только три возможных значения: **On schedule**, **Delayed** и **Canceled**. Эти значения распределены *неравномерно!*

status	prcnt
Canceled	0.00025030153284073211
Delayed	0.12323617925688256028
On schedule	0.87651351921027670761

Создавать индекс по столбцу **status** было бы непрактично из-за высокой селективности этого столбца. Однако было бы неплохо иметь возможность быстро отфильтровывать отмененные рейсы [2, стр. 89]

```
-- здесь частичный индекс вешается на первичный ключ
CREATE INDEX flight_canceled ON flight (flight_id) -- flight_id это PK
WHERE status = 'Canceled';
```

Можно создавать *частичный индекс* для *первичного ключа* (т.е. для атрибута, для которого уже существует уникальный индекс).

Этот индекс будет использоваться во всех запросах, в которых мы выбираем отмененные рейсы, независимо от любых других условий фильтрации, например

```
SELECT *
FROM flight
WHERE
  scheduled_departure BETWEEN '2020-08-15' AND '2020-08-18'
  AND status = 'Canceled'; -- будет использован частичный индекс flight_canceled
```

6.5. Индексы и порядок соединения

Цель оптимизации коротких запросов – избежать больших промежуточных результатов. Это означает, что самые ограничительные критерии фильтрации должны применяться первыми. И затем после каждой операции соединения результат должен оставаться небольшим.

6.6. Избегаем использования индексов

Индекс не всегда нужен и в некоторых случаях может только мешать. Две основные причины, по которым мы можем захотеть избежать использовать индекс, заключается в следующем [2, стр. 94]:

- небольшая таблица полностью считывается в оперативную память,
- для выполнения запроса нам нужна большая часть строк таблицы.

Преобразования столбцов можно использовать и для повышения производительности, когда нужно предотвратить использование индекса.

Столбец числового типа можно изменить, добавив к его значению ноль. Например, условие `attr1 + 0 = p_value` не даст использовать индекс для столбца `attr1`. Для любого типа данных функция `coalesce` будет блокировать использование индексов, поэтому, предполагая, что `attr2` не допускает неопределенных значений, условие можно изменить и написать что-то вроде

```
coalesce(t1.attr2, '0') = coalesce(t2.attr2, '0')
```

Если PostgreSQL игнорирует использование индекса, то этому скорее всего есть веская причина. Например, если требуется сортировка не по индексированному атрибуту, PostgreSQL потребуется выбрать *все* записи, прежде чем он сможет понять, какие из них идут первыми.

Оптимизатор PostgreSQL может выбрать последовательное сканирование, потому что, например, интервал запроса слишком велик, чтобы получить какие-то преимущества от индексного доступа [2, стр. 96].

6.7. Создавать или не создавать

20 лет назад мы были более осторожны при принятии решения о добавлении еще одного индекса [2, стр. 98]:

- во-первых, индексы занимают дополнительное место в базе данных,
- во-вторых, операции *вставки* и *обновления* замедляются, когда вместе с самой записью приходится обновлять еще и много индексов.

Раньше обычно рекомендовалось удалить все индексы в таблице перед массовой загрузкой, а затем создать их снова. Но времена изменились. При нынешнем аппаратном и программном обеспечении ситуация выглядит иначе и в целом быстрый отклик более ценен, чем экономия места на диске. 20 лет назад настораживало, если размер индексов в таблице превышал размер самой таблицы. В наши дни это норма для OLTP-систем.

В OLTP-системах время отклика обычно имеет решающее значение, и любой короткий запрос должен поддерживаться индексами.

Рекомендуется создавать *частичные* и покрывающие индексы, когда это имеет смысл. Частичные индексы меньше обычных и с большей вероятностью поместятся в оперативной памяти.

Дополнительное время, необходимое для вставки и обновления, обычно менее важно, чем быстрый отклик. Однако за этим временем надо следить.

Хотя обычно нас не волнует место на диске, занимаемое индексами, мы не хотим создавать бесполезные объекты базы данных. Системное представление `pg_stat_all_indexes` показывает статистику использования индекса (общее количество сканирований, количество прочитанных индексных строк и количество выбранных табличных строк) с момента последнего сброса статистики.

Некоторые индексы первичного ключа никогда не используются для извлечения данных, но они жизненно важны для целостности данных и не должны удаляться.

Замечание

Если я правильно понял, то у *коротких запросов* на любом шаге выполнения должно участвовать *небольшое* подмножество строк

Как я понял:

- короткие запросы (OLTP-запросы, транзакционные запросы) предполагают низкую селективность запроса и должны поддерживаться индексами; на всех этапах короткого запроса участвует небольшое подмножество строк; нестрашно, если суммарный размер индексов в несколько раз превышает размер самой таблицы; для OLTP-систем важнее время отклика,
- длинные запросы (OLAP-запросы, аналитические запросы) предполагают высокую селективность запроса по крайней мере для одной из больших таблиц; но тем не менее все равно нужно стараться размер промежуточных наборов данных делать как можно меньше; для длинных запросов индексы не нужны.

7. Длинные запросы и последовательное сканирование

Запрос считается *длинным*, если *селективность* запроса *высока* по крайней мере для одной из больших таблиц: то есть результат, даже если он невелик, определяется почти всеми строками.

Повысить производительность длинного запроса можно с помощью двух стратегий [2, стр. 104]:

- избегать многократных сканирований таблиц,
- уменьшать размер результата на как можно более ранней стадии.

Короткие запросы требуют наличия *индексов* по столбцам (атрибутам), включенным в критерии поиска. Для *длинных запросов* все наоборот: *индексы не нужны* (!), а если таблицы проиндексированы, надо убедиться, что индексы не используются [2, стр. 105].

В большинстве случаев [2, стр. 106]:

- *индексный доступ (Index Scan)* работает хорошо с *алгоритмом вложенного цикла*,
- а *последовательное сканирование (Seq Scan)* хорошо работает с *соединением хешированием (Hash Join)*.

Для *коротких запросов* желаемый порядок соединения – тот, при котором в первую очередь используются *индексы с более низкой селективностью* [2, стр. 106].

Замечание

Мы не ожидаем, что индексы будут использоваться в длинных запросах, но даже когда индексы не используются, порядок соединений имеет значение, потому что важно, чтобы промежуточные наборы данных были как можно меньше [2, стр. 106].

Наиболее ограничительные соединения (то есть соединения, которые сильнее всего уменьшают количество результирующих строк) должны выполняться первыми.

7.1. Полусоединение

Часто наиболее ограничительным соединением в запросе является *полусоединение*. Полусоединение не является дополнительной операцией SQL. Полусоединение удовлетворяет следующим условиям:

- В результирующем множестве появляются *только столбцы* (атрибуты) *первой* таблицы,
- *Строки* из первой таблицы *не дублируются*, если во второй таблице для них есть несколько соответствий.

Пример полусоединения с ключевым словом EXISTS

```
-- только эта версия гарантирует наличия SEMI JOIN
SELECT * FROM flight f
WHERE
  EXISTS (
    SELECT flight_id
    FROM booking_leg
    WHERE flight_id = f.flight_id
  );
```

Пример полусоединения с ключевым словом IN

```
-- соединение вложенными циклами
-- для каждой строки внешнего цикла (flight) нужно найти
-- одну строку (в общем случае все строки) внутреннего цикла booking_leg
SELECT * FROM flight WHERE flight_id IN (
  SELECT flight_id FROM booking_leg
);
```

В плане выполнения должно появиться SEMI JOIN.

Поскольку *полусоединение* может *значительно уменьшить* размер набора результатов и, по определению, *никогда не увеличивает его*, оно часто является *наиболее ограничительным* соединением в запросе, и как указывалось ранее, именно наиболее ограничительное соединение должно выполняться в первую очередь [2, стр. 108].

Если запрос использует полусоединение, а в плане выполнения запроса указывается JOIN, а не SEMI JOIN, то дело вероятнее всего в том, что в ходе выполнения не пришлось удалять дубликаты.

7.2. Антисоединение

Антисоединение – это анти полусоединение.

Пример использования антисоединения с помощью NOT EXISTS

```
-- только эта версия гарантирует наличия ANTI JOIN
SELECT * FROM flight f WHERE NOT EXISTS (
  SELECT flight_id FROM booking_leg WHERE flight_id = f.flight_id
);
```

Использование антисоединения с помощью NOT IN

```
SELECT * FROM flight WHERE flight_id NOT IN (
  SELECT flight_id FROM booking_leg
);
```

Не существует общих указаний относительно того, какой синтаксис для антисоединения лучше. Разработчикам следует опробовать оба способа, чтобы увидеть, какой из них будет работать лучше в их случае.

Антисоединения не может создавать дубликаты, а это значит, что можно использовать `OUTER JOIN` с последующим отбором неопределенных значений

```
SELECT f.flight_id
FROM flight f
LEFT OUTER JOIN booking_leg bl USING (flight_id)
WHERE bl.flight_id IS NULL;
```

Оптимизатор распознает эту конструкцию и переписывает ее на антисоединение.

Длинные запросы более вероятны в OLAP-системах. Другими словами, длинный запрос, скорее всего, представляет собой аналитический отчет, соединяющий внушительное количество таблиц [2, стр. 115].

Когда количество таблиц становится слишком, участвующих в заросе, становится *слишком большим*, оптимизатор *больше не пытается* найти лучший порядок соединения из всех возможных [2, стр. 115].

Параметр `join_collapse_limit` ограничивает количество таблиц в соединении, которые будут обрабатываться стоимостным оптимизатором. Значение этого параметра по умолчанию равно 8. Это означает, что если количество таблиц в соединении ≤ 8 , оптимизатор построит планы-кандидаты, сравнит и выберет лучший. Но если количество таблиц ≥ 9 , он просто выполнит соединение в том порядке, в котором таблицы перечислены в запросе

```
=# show join_collapse_limit; -- 8
=# SET join_collapse_limit = 5;
```

Можно задать любое значение для этого параметра, но чем выше значение параметра, тем больше времени будет потрачено на выбор лучшего плана. Число возможных планов равно факториалу от числа таблиц: `#planes = (#tables)!`.

Если разработчик SQL знает лучший порядок соединений, можно принудительно установить этот порядок, задав для `join_collapse_limit` значение 1. В этом случае оптимизатор сгенерирует план, в котором соединения будут выполняться в том порядке, в каком они указаны в команде `SELECT???` [2, стр. 116].

Еще один способ установить определенный порядок соединений – использовать *общие табличные выражения*.

7.3. Группировка

Неоптимальные решения относительно момента выполнения группировки часто становятся основной причиной медленного выполнения запросов.

Пример перемещения группировки на уровень `GROUP BY`

```
SELECT
  bl.flight_id,
  f.departure_airport,
  (avg(b.price))::numeric (7,2) AS avg_price, -- 7 полей, 2 знака после запятой
  count(DISTINCT passenger_id) AS num_passengers
FROM
  booking b
  JOIN booking_leg bl USING (booking_id)
  JOIN flight f USING (flight_id)
  JOIN passenger p USING (booking_id)
WHERE flight_id = 222183 -- сначала фильтруем
GROUP BY 1, 2; -- потом группируем
```

Замечание

Перед группировкой отфильтруйте строки, не нужные для агрегации [2, стр. 122]:

1. фильтруем,
2. группируем

Длинный запрос к большому объему данных нельзя выполнить за доли секунды, даже если он выполняется оптимально. Главное – использовать так мало строк, как только возможно [2, стр. 123].

В некоторых случаях порядок действий должен быть противоположенным: **GROUP BY** следует выполнить как можно раньше, а затем выполнить другие операции. Такой порядок желателен, когда *группировка уменьшает размер промежуточного набора данных*.

Пример, в котором сначала выполняется группировка

```
SELECT
  city,
  date_trunc('month', scheduled_departure),
  sum(passengers) passengers
FROM
  airport a
  JOIN flight f ON airport_code = departure_airport
  JOIN (
    SELECT
      flight_id,
      count(*) passengers
    FROM
      booking_leg l
      JOIN boarding_pass b USING (booking_leg_id)
    GROUP BY flight_id
  ) cnt USING (flight_id)
GROUP BY 1, 2
ORDER BY 3 DESC;
```

7.4. Использование операций над множествами

Операции над множествами редко используются в SQL запросах, но для *больших* запросов эти операции могут побудить оптимизатор выбрать *более эффективные* алгоритмы.

Часто можно:

- использовать **EXCEPT** вместо **NOT EXISTS** и **NOT IN**,
- использовать **INTERSECT** вместо **EXISTS** и **IN**,
- использовать **UNION** вместо сложных критериев выбора с **OR**.

Иногда может наблюдаться значительный прирост производительности, а иной раз время выполнения изменяется лишь незначительно, но код становится чище и проще в сопровождении.

Например антисоединение

```
SELECT * FROM flight WHERE flight_id NOT IN (
  SELECT flight_id FROM booking_leg
);
```

можно переписать так

```
SELECT flight_id FROM flight
EXCEPT
```



```
SELECT flight_id FROM booking_leg;
```

Или для запроса, выбирающего все рейсы с хотя бы одним бронированием

```
SELECT flight_id FROM flight f
INTERSECTION
SELECT flight_id FROM booking_leg;
```

Нам редко приходится переписывать сложные критерии выбора с помощью OR в UNION ALL, потому что в большинстве случаев оптимизатор PostgreSQL справляется с анализом таких критериев и использованием всех подходящих индексов, однако иногда такое переписывание делает код более удобным для сопровождения, особенно когда запрос содержит большое количество различных критериев выбора, связанных с OR.

Запрос со сложными условиями с OR

```
SELECT CASE
  WHEN actual_departure > scheduled_departure + interval '1 hour'
    THEN 'Late group 1'
    ELSE 'Late group 2'
  END AS grouping,
  flight_id,
  count(*) AS num_passengers
FROM
  boarding_pass bp
  JOIN booking_leg bl USING (booking_leg_id)
  JOIN booking b USING (booking_id)
  JOIN flight f USING (flight_id)
WHERE
  departure_airport = 'FRA'
  AND actual_departure > '2020-07-01'
  AND (
    (
      actual_departure > scheduled_departure + interval '30 minute' AND
      actual_departure <= scheduled_departure + interval '1 hour'
    ) OR (
      actual_departure > scheduled_departure + interval '1 hour' AND
      bp.update_ts > scheduled_departure + interval '30 minute'
    )
  )
GROUP BY 1, 2;
```

Этот запрос можно переписать так

Сложное условие с OR переписано, используя UNION ALL

```
SELECT
  'Late group 1' AS grouping,
  flight_id,
  count(*) AS num_passengers
FROM
  boarding_pass bp
  JOIN booking_leg bl USING (booking_leg_id)
  JOIN booking b USING (booking_id)
  JOIN flight f USING (flight_id)
WHERE
  departure_airport = 'FRA'
  AND actual_departure > scheduled + interval '1 hour'
  AND bp.update_ts > scheduled_deparutre + interval '30 minutes'
  AND actual_departure > '2020-07-01'
```

```

GROUP BY 1, 2
UNION ALL
SELECT
    'Late group 2' AS grouping,
    flight_id,
    count(*) AS num_passengers
FROM
    boarding_pass bp
JOIN booking_leg bl USING (booking_leg_id)
JOIN booking b USING (booking_id)
JOIN flight f USING (flight_id)
WHERE
    departure_airport = 'FRA'
AND actual_departure > scheduled_departure + interval '30 minute'
AND actual_departure <= scheduled_departure + interval '1 hour'
AND actual_departure > '2020-07-01'
GROUP BY 1, 2;

```

Стоит отметить, что с большими запросами всегда нужно учитывать, какой оперативной памятью вы располагаете. Скорость выполнения как соединений хешированием, так и операций со множествами значительно уменьшается, если участвующие наборы данных не помещаются в оперативную память.

7.5. Избегаем многократного сканирования

Еще одна причина медленного выполнения запросов – это многократное сканирование таблиц. Эта распространенная проблема является прямым результатом неудачного проектирования схемы данных.

Пусть система уже запущена и работает, и внезапно нам потребовалось сохранять некую дополнительную информацию для объектов, которые уже присутствуют в базе данных.

В течение последних 30 лет самым простым решением в таких случаях было использование таблицы *сущность-атрибут-значение* (entity-attribute-value, EAV), в которой могут храниться произвольные атрибуты – те, которые нужны сейчас, и те, которые понадобятся когда-нибудь.

Пусть требуется отчет, показывающий имена пассажиров и их паспортные данные. Типичное неудачное решение

```

SELECT
    first_name,
    last_name,
    pn.custom_field_value AS passport_num,
    pe.custom_field_value AS passport_exp_date,
    pc.custom_field_value AS passport_country
FROM passenger p
JOIN custom_field pn ON pn.passenger_id = p.passenger_id AND pn.custom_field_name = '
    passport_name'
JOIN custom_field pe ON pe.passenger_id = p.passenger_id AND pe.custom_field_name = '
    passport_exp_date'
JOIN custom_field pc ON pc.passenger_id = p.passenger_id AND pc.custom_field_name = '
    passport_country'
WHERE p.passenger_id < 5000000;

```

То есть если бы база данных спроектирована нормально, то просто у таблицы `passenger` было бы еще 3 столбца: `passport_num`, `passport_exp_date` и `passport_country`, а теперь приходится 3 раза вычитывать огромную таблицу с 16 млн. строк.

Переписать предыдущий запрос можно так

```

SELECT
  first_name, -- 1
  last_naem, -- 2
  p.passenger_id, -- 3
  coalesce(max(CASE WHEN custom_field_name = 'passport_num'
                    THEN custom_field_value ELSE NULL
                END), '') AS passport_num,
  coalesce(max(CASE WHEN custom_field_name = 'passport_exp_date'
                    THEN cutsom_field_value ELSE NULL
                END, '')) AS passport_exp_date,
  coalesce(max(CASE WHEN custom_field_name = 'passport_country'
                    THEN custom_field_value ELSE NULL
                END, '')) AS passport_country
FROM
  passenger p
  JOIN custom_field cf USING (passenger_id)
WHERE cf.passenger_id < 5000000 AND p.passenger_id < 5000000
GROUP BY 3, 1, 2;

```

Замечание

Выносите значения из таблицы EAV в подзапрос перед соединением с другими таблицами [2, стр. 132]

Пример переноса группировки в подзапрос

```

SELECT
  last_name,
  first_name,
  passport_num,
  passport_exp_date,
  passport_country
FROM
  passenger p
  JOIN (
    SELECT
      cf.passenger_id,
      coalesce(max(CASE WHEN custom_field_name = 'passport_name'
                        THEN custom_field_value ELSE NULL
                    END), '') AS passport_num,
      coalesce(max(CASE WHEN custom_field_name = 'passport_exp_date'
                        THEN custom_field_value ELSE NULL
                    END), '') AS passport_exp_date,
      coalesce(max(CASE WHEN custom_field_name = 'passport_country'
                        THEN custom_field_value ELSE NULL
                    END), '') AS passport_country
    FROM custom_field cf
    WHERE cf.passenger_id < 5000000
    GROUP BY 1 -- группировка перед соединением
  ) info USING (passenger_id)
WHERE p.passenger_id < 5000000;

```

Индексы не обязательно ускоряют выполнение запросов, а фактически могут *замедлить* выполнение *длинных* запросов [2, стр. 133].

Как и короткие запросы, длинные запросы оптимизируются за счет [2, стр. 133]:

- *уменьшения размера* промежуточных результатов,
- и выполнения необходимой работы с *как можно меньшим количеством строк*.

В случае с *короткими* запросами это достигается путем применения *индексов* по наиболее ограничительным критериям (то есть по индексам с более *низкой селективностью*¹).

В случае с *длинными* запросами это достигается за счет [2, стр. 133]:

- выбора правильного порядка соединений,
- использования полу- и антисоединений,
- применения фильтрации перед группировкой,
- группировки перед соединением,
- а также использования операций над множествами.

7.6. Длинные запросы. Дополнительные приемы

Декларативный характер SQL диктует совершенно иной стиль декомпозиции запросов, нежели тот, который использовался бы для кода приложения. Код SQL, как и код на любом другом языке, должен быть простым для понимания и изменения, но не за счет производительности.

7.6.1. Временные таблицы

Иногда попытка разработчиков SQL ускорить выполнение запроса, наоборот, может привести к замедлению. Такое часто случается, когда они решают использовать *временные таблицы*.

Чтобы создать временную таблицу, нужно выполнить обычную команду CREATE TABLE, добавив ключевое слово TEMPORARY или прсото TEMP

```
CREATE TEMP TABLE interim_results;
```

Временные таблицы видны только текущему сеансу и удаляются при его завершении, если не были удалены явно до этого. В остальном они ничем не уступают обычным таблицам: их можно использовать в запросах без всяких ограничений и даже можно инексировать. Временные таблицы часто используются для хранения промежуточных результатов запросов, поэтому команда CREATE нередко выглядит так [2, стр. 134]

```
CREATE TEMP TABLE interim_resulsts AS  
SELECT ...
```

Все это отлично работает, если вы сохраняете результаты своего запроса во временной таблице для *некоего анализа*, а затем *удаляете* ее. Но если использовать временные таблицы для хранения результатов каждого шага, код начинает выглядеть так

```
CREATE TEMP TABLE T1 AS  
SELECT ...;  
  
CREATE TEMP TABLE T2 AS  
SELECT ...  
FROM T1 INNER JOIN ...  
...
```

Цепочка временных таблиц может стать довольно длинной и это вызывает проблемы [2, стр. 136]:

¹Для коротких запросов желаемый порядок соединения – тот, при котором в первую очередь используются индексы с более низкой селективностью. Но даже когда индексы не используются, порядок соединений имеет значение, потому что важно, чтобы промежуточные наборы данных были как можно меньше [2, стр. 106]

- *индексы* – после того как выбранные данные будут сохранены во *временной таблице*, *мы не можем использовать индексы, созданные в исходной таблице (или таблицах)*. Нам либо придется обойти без индексов, либо создать новые во временных таблицах, что требует ресурсов,
- *статистика* – поскольку мы создали новую таблицу, оптимизатор *не может* использовать *статистические данные* о распределении значений из исходной таблицы (или таблиц), поэтому нам придется либо без обойтись без статистики, либо выполнить команду **ANALYZE** для временной таблицы,
- *место на диске* – когда промежуточные результаты не помещаются в доступную оперативную память, временные таблицы сохраняются на диске,
- *чрезмерный ввод-вывод* – временные таблицы – это таблицы, и они могут быть записаны на диск, а для *записи на диск и чтения с диска* требуется дополнительное время.

NB: Но наиболее важным отрицательным следствием чрезмерного использования *временных таблиц* является то, что *такая практика не дает оптимизатору переписывать запросы*.

Сохраняя результаты каждого соединения во временную таблицу, вы *не позволяете оптимизатору выбирать оптимальный порядок соединений*; вы фиксируете тот порядок, в котором создаете временные таблицы [2, стр. 136].

7.6.2. Общие табличные выражения, CTE

Пример запроса с CTE

```
WITH flights_totals AS (  -- CTE
  SELECT
    bl.flight_id,  -- 1
    departure_airport,  -- 2
    (avg(price))::numeric(7, 2) AS avg_price,
    count(DISTINCT passenger_id) AS num_passengers
  FROM
    booking b
    JOIN booking_leg bl USING (booking_id)
    JOIN flight f USING (flight_id)
    JOIN passenger p USING (booking_id)
  GROUP BY 1, 2
)
SELECT
  flight_id,
  avg_price,
  num_passengers
FROM
  flights_totals
WHERE departure_airport = 'ORD';
```

Общие табличные выражения можно рассматривать как временные таблицы, существующие только для одного запроса. В предложении **WITH** могут использоваться команды **SELECT**, **INSERT**, **UPDATE** или **DELETE**, а само предложение **WITH** присоединяется к основному оператору, который также может быть командой **SELECT**, **INSERT**, **UPDATE** или **DELETE**.

Для всех версий ниже 12 общее табличное выражение обрабатывалось точно так же, как временная таблица. Результаты материализовались в основной памяти с возможным сбросом на диск. Это означает, что использование CTE не имело никаких преимуществ перед временной таблицей.

CTE предлагалось использовать в том случае, если какой-то *вложенный подзапрос* используется в запросе *несколько раз*, то его можно определить как общее табличное выражение и ссылаться на него столько раз, сколько потребуется. В этом случае PostgreSQL *вычислит результаты только один раз* и повторно использует их при обращениях [2, стр. 138].

Из-за такого предполагаемого применения оптимизатор *планировал выполнение CTE отдельно* от остальной части запроса и *не перемещал никакие условия соединения внутрь общего табличного выражения*.

PostgreSQL 12 радикально изменил оптимизацию общих табличных выражений [2, стр. 138]:

- о если не рекурсивное CTE используется в запросе *только один раз*, то барьер оптимизации снимается и CTE *встраивается* во внешний запрос,
- о если CTE вызывается *несколько раз*, то сохраняется *старое* поведение.

Описанное поведение используется по умолчанию, но его можно изменить, используя ключевые слова **MATERIALIZED** (старое поведение) и **NOT MATERIALIZED** (встраивание во внешний запрос).

То есть получается, что материализованное общее табличное выражение может мешать оптимизатору, если это общее табличное выражение используется в запросе только один раз, а версия PostgreSQL новее 11, потому что без материализации оптимизатор заметил бы, что CTE используется только один раз и встроил бы его во внешний запрос.

По-прежнему нужно не навязывать неоптимальный план выполнения, но использование цепочки общих табличных выражений намного лучше, чем использование последовательности временных таблиц. В последнем случае оптимизатор беспомощен – он не умеет оптимизировать запросы на базе цепочки временных таблиц.

7.6.3. Преставления

Представления – это объект базы данных, в котором хранится запрос, определяющий виртуальную таблицу.

Представление – это *виртуальная таблица* в том смысле, что синтаксически представления могут использоваться в запросах так же, как и таблицы. Однако они значительно отличаются от таблиц тем, что *не хранят данные*; в базе хранится только *запрос*, определяющий представление.

С одной стороны, представление обычно создают именно для инкапсуляции, чтобы другие могли использовать его, не разбираясь в логике запроса. С другой стороны, эта непрозрачность приводит к низкой производительности.

Когда нет понимания, какой запрос скрывает представление, оно может использоваться для выбора данных, которые намного легче получить из базовых таблиц.

Внутри PostgreSQL любое создание представления включает в себя создание правил, в большинстве случаев неявно. Правила **SELECT** могут ограничивать доступ к базовым таблицам. Правила, триггеры и автоматическое обновление делают представления в PostgreSQL чрезвычайно сложными и обеспечивают функциональность, очень похожую на таблицы.

Однако представления не дают никакого преимущества в плане производительности [2, стр. 146].

7.6.4. Материализованные представления

Материализованное представление – это объект базы данных, который *объединяет* в себе *и определение запроса, и таблицу* для хранения результатов этого запроса, когда он выполнен.

Материализованное представление отличается от обычного представления, потому что *сохраняются результаты запроса, а не только определение представления*. Это означает, что материализованное представление отражает не актуальные данные, а данные на момент последнего обновления.

Материализованное представление отличается от таблицы, потому что *данные* в нем *нельзя изменять напрямую*, их *можно только обновить* с помощью предопределенного запроса.

Когда запросы ссылаются на *материализованные представления*, те (в отличие от обычных представлений) ведут себя точно так же, как *таблицы*. Оптимизатор не будет заменять материализованное представление определяющим запросом, оно будет доступно в виде таблиц.

Для материализованных представлений могут быть созданы *индексы*, однако у них **не может быть первичного и внешнего ключа!**.

Команда REFRESH заполняет материализованное представление результатами базового запроса во время выполнения обновления

```
REFRESH MATERIALIZED VIEW flight_departure_mv;
```

Материализованные представления не могут обновляться инкрементально, а расписание обновления нельзя указать в определении материализованного представления.

Каждый раз при выполнении команды REFRESH *таблица* материализованного представления *опустошается*, и в нее *вставляются результаты* сохраненного запроса. Если во время обновления возникает ошибка, процесс обновления откатывается, и материализованное представление остается неизменным.

Во время обновления материализованное представление блокируется, и его содержимое недоступно для других процессов. Чтобы сделать предыдущую версию материализованного представления доступной во время обновления, к команде добавляется ключевое слово CONCURRENTLY

```
REFRESH MATERIALIZED VIEW CONCURRENTLY flight_departure_mv;
```

Обновление материализованного представления *требует времени*, а *выбор* из материализованного представления происходит *намного быстрее*, чем из обычного представления.

Нужно учесть следующее:

- как часто меняются данные в базовых таблицах?
- насколько важно иметь самые свежие данные?
- как часто нужно выбирать эти данные (а точнее, сколько чтений ожидается на одно обновление)?
- сколько разных запросов будут использовать эти данные?

Хотя запрос материализованного представления выполняется реже, чем используется само представление, нам все равно нужно обращать внимание на время его выполнения [2, стр. 150].

7.6.5. Секционирование

Секция представляет собой другой вид разделения – не кода, а данных. Секционированная таблица состоит из нескольких секций, каждая из которых определяется как отдельная таблица. Каждая табличная строка хранится в одной из секций в соответствии с правилами, указанными при создании секционированной таблицы.

Наиболее распространенный случай – секционирование по диапазонам. Каждая секция содержит строки, в которых значение атрибута находится в диапазоне, назначенном секции. Строку, которая не попадает ни в одну секцию, нельзя вставить.

Пример

```
CREATE TABLE boarding_pass_part (
  boarding_pass_id SERIAL,
  passenger_id BIGINT NOT NULL,
  booking_leg_id BIGINT NOT NULL,
  seat TEXT,
  boarding_time TIMESTAMPTZ,
  precheck BOOLEAN NOT NULL,
  update_ts TIMESTAMPTZ
)
PARTITION BY RANGE (boarding_time);

-- создание секций
--
CREATE TABLE boarding_pass_may
PARTITION OF boarding_pass_part
FOR VALUES FROM ('2020-05-01'::timestamptz) TO ('2020-06-01'::timestamptz);
--
CREATE TABLE boarding_pass_june
PARTITION OF boarding_pass_part
FOR VALUES FROM ('2020-06-01'::timestamptz) TO ('2020-06-01'::timestamptz);
...
INSERT INTO boarding_pass_part SELECT * FROM boarding_pass;
```

Секции можно добавлять к секционированной таблице и можно удалять. Команда DROP выполняется значительно быстрее, чем команда DELETE, удаляющая множество строк, и не требует последующей очистки.

К таблице, разделенной по диапазонам дат (например, по секции на каждый месяц), в конце каждого месяца добавляется новая секция, а самая старая удаляется.

Секционирование можно использовать для распределения больших объемов данных по нескольким серверам баз данных.

С точки зрения производительности, секционирование может сократить время, необходимое для полного сканирования таблицы: если запрос содержит условия для ключа секционирования, сканирование ограничивается только подходящими секциями.

Это делает *секционирование* особенно *полезным* для *длинных* запросов, часто использующих *полное сканирование*.

И сканирования, и соединения хешированием типичны для длинных запросов, для которых ускорение хешированием наиболее значимо.

8. Оптимизация модификации данных

У любой системы баз данных есть два языка: DDL (Data Definition Language – язык описания данных), используемый для создания таблиц и других объектов базы данных, и DML (Data Manipulation Language – язык управления данными), который применяется, чтобы запрашивать и изменять данные к базе.

В PostgreSQL DDL и DML являются частью SQL:

- о DML: INSERT, UPDATE и DELETE,
- о DDL: остальные команды ALTER TABLE, DROP INDEX, CREATE MATERIALIZED VIEW и пр.

В подавляющем большинстве случаев даже системы OLTP выполняют значительно меньше команд DML, чем команд SELECT. Это основная причина того, что люди редко говорят об оптимизации DML. Однако длительно работающие команды DML могут вызывать проблемы не только

потому, что обновленные данные не будут своевременно доступны в системе, но также из-за возможности появления блокирующих замков, которые замедляют выполнение других команд.

В конечном счете любая операция SQL, какой бы она ни была, сводится к паре низкоуровневых операций: чтению и записи отдельных блоков базы данных.

Причина проста [2, стр. 158]:

- *данные, содержащиеся в базе, могут быть обработаны только при извлечении блоков в оперативную память,*
- *и все изменения сначала выполняются в памяти, а уже затем записываются на диск.*

Фундаментальное различие между операциями чтения и записи состоит в том, что *чтение с диска* должно быть завершено до того, как данные можно будет обработать; таким образом, команду **SELECT** нельзя завершить до того, как все необходимые *блоки будут загружены в память*. Напротив, *изменения данных внутри блока* завершаются до начала *записи на диск*; таким образом, операцию SQL можно завершить без задержек. Нет необходимости ждать, пока измененные данные действительно будут записаны на диск.

Входит, любая команда **INSERT**, **UPDATE** или **DELETE** выполняется намного быстрее, чем **SELECT**. Но оптимизация все равно нужна. Во-первых, *запись на диск* все равно *необходима* и, следовательно, потребляет некоторое количество аппаратных ресурсов и может замедлять обработку и даже повлиять на производительность запросов. Во-вторых, модификации могут мешать другим модификациям и даже поиску.

Данные могут быть доступны *одновременно* из разных команд **SELECT**, но модификации не могут выполняться *одновременно*, и в этом случае все решает порядок выполнения операций.

Любые обновления, выполненные зафиксированной транзакцией, должны надежно сохраняться на жестком диске перед фиксацией. Для этого используется *журнал предзаписи*. Все изменения данных регистрируются в журнальных записях на жестком диске, прежде чем транзакцию можно будет зафиксировать.

Одна из отличительных особенностей PostgreSQL заключается в том, что они *никогда не выполняет обновления на месте*. Вместо этого новая версия элемента (например, строки таблицы) вставляется и сохраняется в свободное место в том же или в новом выделенном блоке, в то время как *предыдущая версия не перезаписывается тотчас же*.

Добавление индексов к таблице может потенциально замедлить операции DML, но экспертное мнение таково, что добавление дополнительного индекса в среднем приводит к увеличению времени выполнения команд **INSERT** и **UPDATE** всего на 1% [2, стр. 161].

PostgreSQL вставляет новые версии обновленных строк. Это оказывает определенное отрицательное влияние на производительность: как правило, поскольку новые версии имеют *другое расположение*, необходимо обновить все индексы в таблице.

Чтобы уменьшить этот негативный эффект, PostgreSQL использует технику, которую иногда называют HOT (heap-only tuples). Если при изменении строки в блоке достаточно свободного места, чтобы вставить новую версию в него же, и если обновление не требует внесения каких-либо изменений в индексированные столбцы, то нет необходимости изменять индексы.

Наличие нескольких внешних ключей в таблице потенциально может замедлить DML. Причина, по которой операции манипулирования данными могут замедляться, заключается в том, что для каждой операции **INSERT** или **UPDATE** в таблице с ограничениями целостности движок базы данных должен проверить, присутствуют ли новые значения ограниченных столбцов в соответствующих родительских таблицах, таким образом выполняя дополнительные неявные запросы.

Если размер родительской таблицы сопоставим с размером дочерней таблицы, накладные расходы могут быть более заметными. Как и в большинстве других случаев, фактическое время задержки зависит от параметров системы и характеристик аппаратного обеспечения.

Неформально схема базы данных *нормализована*, если все значения столбцов зависят только от первичного ключа, а данные разбиты на несколько таблиц, чтобы избежать повторений [2, стр. 174].

Повышение производительности не является основной целью нормализации. Нормализация создает чистую логическую структуру и помогает обеспечить целостность данных, особенно когда она поддерживается ограничениями ссылочной целостности. Нормализация необходима по тем же причинам, что реляционная модель: не столько для хранения данных, сколько для обеспечения целостности и возможности использовать язык реляционных запросов.

9. Суррогатные ключи

Суррогатные ключи – это уникальные значения, генерируемые системой для идентификации объектов, хранящихся в базе данных. В PostgreSQL суррогатные значения могут быть выбраны из последовательности. Когда строка вставляется, столбец с псевдотипом **serial** автоматически получает следующее значение из последовательности, связанной с таблицей.

Преимущество суррогатов состоит в том, что значения, присвоенные различным объектам, гарантированно уникальны. Однако значение суррогатного ключа не связано с другими атрибутами и бесполезно при сопоставлении сохраненного объекта с объектом реальным.

Уникальность суррогатных ключей может скрывать определенные ошибки. Реальный объект может быть представлен в базе данных несколько раз с разными суррогатными ключами.

10. Функции

Встроенные функции написаны на языке C и интегрированы с сервером PostgreSQL. Пользовательские функции – это функции, которые создает сам пользователь. PostgreSQL поддерживает три типа пользовательских функций:

- функции на языке запросов, то есть функции, написанные на SQL,
- C-функции, написанные на языке C (или C-подобных языках, например C++),
- функции, написанные на одном из поддерживаемых *процедурных языков*.

Стандартный дистрибутив PostgreSQL поддерживает четыре процедурных языка: PL/pgSQL, PL/Tcl, PL/Perl и PL/Python.

Функция, написанная на PL/pgSQL, может включать в себя любые инструкции SQL (возможно, с некоторыми изменениями), управляющие структуры (IF THEN ELSE, CASE, LOOP) и вызовы других функций.

Пример функции на процедурном языке PL/pgSQL

```
CREATE OR REPLACE FUNCTION text_to_numeric(input_text text)
RETURNS numeric AS
-- тело функции как строковый литерал
$BODY$ -- долларовые кавычки
BEGIN
    RETURN replace(input_text, ',', '');
EXCEPTION
    WHEN OTHERS THEN
```

```

    RETURN NULL::numeric;
END;
$BODY$ LANGUAGE plpgsql;

-- SELECT text_to_numeric('3.14');
-- SELECT text_to_numeric(input_text := '3.14');
```

Тело функции упаковано в *строковый литерал*, однако вместо кавычек он начинается и заканчивается символами `$BODY$`. Такая нотация в PostgreSQL называется *долларовыми кавычками*, и она особенно полезна при создании функций. Действительно, в большом тексте функции, скорее всего, встретятся одинарные кавычки или обратная косая черта, и вам придется экранировать их в каждом случае. С долларовыми кавычками вы определяете строковый литерал, используя два знака доллара, возможно, с некоторым тегом между ними.

Для обработки исключений в функции есть раздел `EXCEPTION`. Ключевое слово `WHEN` может указывать конкретные типы исключений. Форма `WHEN OTHERS` соответствует всем типам исключений, не включенным в предыдущие условия `WHEN`.

Ключевое слово `END` должно быть последней инструкцией тела функции. Любому параметру функции мы можем присвоить значение по умолчанию, которое будет использоваться, если конкретное значение не передается явно.

Параметры обычно имеют имена, но к ним можно обращаться и по номеру позиции (`$1`, `$2` и т.д.). Вместо указания типа возвращаемого значения функции некоторые параметры могут быть определены как `OUT` или `INOUT`.

Функция может и не возвращать значение; в этом случае она определяется как `RETURNS void`.

Как и в других языках программирования, функции в PostgreSQL могут быть *полиморфными*, то есть несколько функций могут использовать одно и то же имя с разной сигнатурой.

Исходный код функции `num_passengers(integer)`

```

CREATE OR REPLACE FUNCTION num_passengers(p_flight_id int)
RETURNS int AS
$$
BEGIN
    RETURN (
        SELECT count(*)
        FROM
            booking_leg bl
            JOIN booking b USING (booking_id)
            JOIN passenger p USING (booking_id)
        WHERE flight_id = p_flight_id
    );
END;
$$ LANGUAGE plpgsql;
-- num_passengers(p_flight_id := 13);
```

Исходный код функции `num_passengers(text, date)`

```

CREATE OR REPLACE FUNCTION num_passengers(
    p_airport_code text,
    p_departure date
) RETURNS int AS
$$
BEGIN
    RETURN (
        SELECT count(*)
        FROM booking_leg bl
```

```

    JOIN booking b USING (booking_id)
    JOIN passenger p USING (booking_id)
    JOIN flight f USING (booking_id)
    WHERE flight_id = p_flight_id
);
END;
$$ LANGUAGE plpgsql;

```

В отличие от других СУБД, функции *интерпретируются*, а не компилируются!

При создании функции в PL/pgSQL [2, стр. 202]:

1. план выполнения не сохраняется,
2. никакие проверки на наличие таблиц, столбцов или других функций не выполняются,
3. вы не знаете, работаете ваша функция или нет, пока не выполните ее (и как правило, не один раз, если есть несколько путей выполнения).

Функции в PostgreSQL могут улучшить читаемость кода, так и ухудшить ее и могут значительно снизить производительность [2, стр. 203].

Если говорить об улучшении производительности отдельной инструкции SQL, то включение ее в функцию не может ускорить выполнение. Однако функции могут быть чрезвычайно полезны, когда оптимизируется процесс.

Пользовательские типы могут быть простыми и составными. Простые пользовательские типы включают в себя домены, перечисления и диапазоны.

Примеры простых типов

```

CREATE DOMAIN timeperiod AS tstzrange;
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TYPE mood_range AS RANGE...
CREATE TYPE <base type>

```

Можно определять *массивы пользовательских типов*

```

DECLARE
    v_moods_set mood[];

```

Составные типы дают нам еще больше возможностей. Составной тип представляет собой *строку* (как в таблице) или *запись*. Определение типа состоит из последовательности имен полей и соответствующих им типов данных.

Например, определим составной тип `boarding_pass_record`

```

-- составной тип
CREATE TYPE boarding_pass_record AS (
    boarding_pass_id int,
    booking_leg_id int,
    flight_no text,
    departure_airport text,
    arrivale_airport text,
    last_name text,
    first_name text,
    seat text,
    boarding_time timestampz
);

```

Теперь, когда определен тип `boarding_pass_record`, мы можем объявлять переменные этого типа так же, как можно объявлять переменные базовых типов:

```
DECLARE
v_new_boarding_pass_record boarding_pass_record;
```

Более того, мы можем создавать функции, возвращающие *множества составных типов*.

Пример функции, которая возвращает все посадочные талоны за раз [2, стр. 207]

```
CREATE OR REPLACE FUNCTION boarding_passes_flight(p_flight_id int)
RETURNS SETOF boarding_pass_record AS -- множество составных типов!
$$
BEGIN
RETURN QUERY
SELECT
    pass_id,
    bp.booking_leg_id,
    flight_no,
    departure_airport::text,
    arrival_airport::text,
    last_name,
    first_name,
    seat,
    boarding_time
FROM
    flight f
    JOIN booking_leg bl USING (flight_id)
    JOIN boarding_pass bp USING (booking_leg_id)
    JOIN passenger USING (passenger_id)
WHERE bl.flight_id = p_flight_id;
END;
$$ LANGUAGE plpgsql;
-- SELECT * FROM boarding_passes_flight(13);
```

Функция, возвращающая набор из одной строки

```
CREATE OR REPLACE FUNCTION boarding_passes_pass(p_pass_id int)
RETURNS SETOF boarding_pass_record AS -- множество составных типов
$$
BEGIN
RETURN QUERY
SELECT
    pass_id,
    bp.booking_leg_id,
    flight_no,
    departure_airport::text,
    arrival_airport::text,
    last_name,
    first_name,
    seat,
    boarding_time
FROM
    flight f
    JOIN booking_leg bl USING (flight_id)
    JOIN boarding_pass bp USING (booking_leg_id)
    JOIN passenger USING (passenger_id)
WHERE pass_id = p_pass_id;
END;
$$ LANGUAGE plpgsql;
```

Применение этих функций могло бы повысить производительность. Дело в том, что приложения редко выполняют инструкцию SQL напрямую. Вместо этого они часто используют инструк-

ции SQL, генерируемые инструментами объектно-реляционного отображения. Скорее всего, в этом случае обращение к таблицам `boarding_passes`, `passengers` и `flights` будет выполняться иначе [2, стр. 208].

Дополнительные составные типы

```
CREATE TYPE flight_record AS (  
    flight_id int,  
    flight_no text,  
    departure_airport_code text,  
    departure_airport_name text,  
    arrival_airport_code text,  
    scheduled_departure timestampz,  
    scheduled_arrival timestampz  
);  
  
CREATE TYPE boarding_pass_record AS (  
    boarding_pass_id integer,  
    booking_leg_id bigint,  
    flight_no text,  
    departure_airport text,  
    arrival_airport text,  
    last_name text,  
    first_name text,  
    seat text,  
    boarding_time timestampz  
);  
  
CREATE TYPE booking_leg_record AS (  
    booking_leg_id int,  
    leg_num int,  
    booking_id int,  
    flight flight_record,  
    boarding_passes boarding_pass_record[] -- массив объектов типа boarding_pass_record  
);
```

Функция, возвращающая сложный объект с вложенной структурой

```
CREATE OR REPLACE FUNCTION booking_leg_select(p_booking_leg_id int)  
RETURNS SETOF booking_leg_record AS -- множество составных типов  
$$  
BEGIN  
    RETURN QUERY  
    SELECT  
        bl.booking_leg_id, -- 1 столбец  
        leg_num, -- 2 столбец  
        bl.booking_id, -- 3 столбец  
        (  
            -- связанный подзапрос  
            SELECT row( -- row собирает элементы строки в один элемент составного типа flight_record  
                flight_id,  
                flight_no,  
                departure_airport,  
                da.airport_name,  
                arrival_airport,  
                aa.airport_name,  
                scheduled_departure,  
                scheduled_arrival  
            )::flight_record  
        FROM  
            flight f  
            JOIN airport da ON da.airport_code = departure_airport
```

```

        JOIN airport aa ON aa.airport_code = arrival_airport
    WHERE flight_id = bl.flight_id  -- ссылается на псевдоним таблицы bl из внешнего запроса
), -- 4 столбец
( -- связанный подзапрос
    SELECT array_agg(row( -- array_agg возвращает МАССИВ записей
        pass_id,
        bp.booking_leg_id,
        flight_no,
        departure_airport,
        arrival_airport,
        last_name,
        first_name,
        seat,
        boarding_time
    )::boarding_pass_record)
    FROM
        flight f1
        JOIN boarding_pass bp ON f1.flight_id = bl.flight_id
        AND bp.booking_leg_id = bl.booking_leg_id
        JOIN passenger p ON p.passenger_id = bp.passenger_id
    ) -- 5 столбец
FROM booking_leg bl
WHERE bl.booking_leg_id = p_booking_leg_id;
END;
$$ LANGUAGE plpgsql;

```

В блоке SELECT подзапросы могут ссылаться на псевдонимы таблиц в главном запросе.

Поскольку инструкции SQL в теле функции не разбираются при создании функции, то нет и зависимостей от таблиц, представлений, материализованных представлений или других функций и хранимых процедур, которые используются в теле функции. По этой причине функции нужно пересоздавать только тогда, когда это действительно нужно, а не просто из-за касадного удаления [2, стр. 213].

Однако функции создают новый тип зависимостей: функции зависят от типов возвращаемых значений, включая пользовательские типы. Чтобы удалить тип, придется удалить и все другие пользовательские типы, которые включают его как элемент, и все функции, которые зависят от того типа.

Функции PL/pgSQL позволяют выполнять любую команду SQL, включая команды DML

```

CREATE OR REPLACE FUNCTION issue_boarding_pass(
    p_booking_leg_id int,
    p_passenger_id int,
    p_seat text,
    p_boarding_time timestampz
)
RETURNS SETOF boarding_pass_record AS
$$
DECLARE
    v_pass_id int;
BEGIN
    INSERT INTO boarding_pass(
        passenger_id,
        booking_leg_id,
        seat,
        boarding_time,
        update_ts
    ) VALUES (
        p_passenger_id,

```

```

    p_booking_leg_id,
    p_seat,
    p_boarding_time,
    now()
)
-- сгенерированный id возвращается и привязывается к переменной pass_id
RETURNING pass_id INTO v_pass_id; -- v_pass_id = pass_id
RETURN QUERY
    SELECT * FROM boarding_passes_pass(v_pass_id);
END;
$$ LANGUAGE plpgsql;

```

Если нужно, чтобы функция отработала, но не возвращала результат (то есть работала как процедура), то можно воспользоваться ключевым словом **PERFORM** (имитирует поведение процедуры)

```
PERFORM issue_boarding_pass(175820, 462972, '22C', '2020-06-16 21:45'::timestampz);
```

Разница между функциями и хранимыми процедурами заключается в том, что *процедуры не возвращают никаких значений*; таким образом, мы не указываем возвращаемый тип [2, стр. 218].

Создать процедуру можно так

```

CREATE PROCEDURE procedure_name(par_name1 pat_type1, ...)
AS
    <procedure body>
LANGUAGE plpgsql;
-- CALL procedure_name;

```

Синтаксис здесь такой же, как и функции, за исключением того, что нет необходимости в типе возвращаемого значения.

Наиболее важное различие между выполнением функций и хранимых процедур заключается в том, что в теле *процедуры* можно *фиксировать* или *откатывать* транзакции

```

-- в процедурах можно управлять транзакциями
CREATE PROCEDURE load_with_transform()
AS $load$
DECLARE
    v_cnt int := 0;
    v_record record;
BEGIN
    FOR v_record IN (SELECT * FROM data_source) LOOP
        PERFORM transform(v_record.id);
        CALL insert_data(v_record.*);
        v_cnt := v_cnt + 1;
        IF v_cnt >= 50000 THEN
            COMMIT;
            v_cnt := 0;
        END IF;
    END LOOP;
    COMMIT;
END;
$load$ LANGUAGE plpgsql;

```

11. Динамический SQL

Динамический SQL – это любая инструкция SQL, которая сначала создается как текстовая строка, а затем выполняется с помощью команды **EXECUTE**.

Почему PostgreSQL работает лучше:

- в PostgreSQL *планы выполнения не кешируются* даже для подготовленных операторов (то есть запросов, которые предварительно разбираются, анализируются и перезаписываются с помощью команды PREPARE). Это означает, что *оптимизация* всегда происходит непосредственно *перед выполнением*,
- этап оптимизации в PostgreSQL происходит позже, чем в других системах. План выполнения создается для определенных значений.

Что касается внедрения SQL-кода, то:

- когда функции получают значения параметров из базы данных напрямую (используя идентификаторы), такие значения не могут содержать никаких вредоносных конструкций,
- значения, полученные в результате ввода данных пользователем, должны быть защищены функциями PostgreSQL (`quote_literal`, `quote_ident` и т.п. или `format`).

PostgreSQL *может* сохранять план выполнения подготовленного оператора. Когда функция вызывается в первый раз в сеансе, каждая выполняемая инструкция SQL разбирается, ее план выполнения оптимизируется и *может* быть закеширован для последующих выполнений.

Чтобы гарантировать, что каждый раз при вызове функции *план выполнения будет оптимизироваться для определенных значений*, мы создаем функции, выполняющие динамический SQL [2, стр. 226].

```
-- оптимизатор не будет использовать кешированный план!!!
CREATE OR REPLACE FUNCTION select_booking_leg_country_dynamic (
    p_country text,
    p_updated timestampz
)
RETURNS SETOF booking_leg_part AS
$body$
BEGIN
    RETURN QUERY EXECUTE $$
        SELECT
            departure_airport,
            booking_id,
            is_returning
        FROM
            booking_leg bl
            JOIN flight f USING (flight_id)
        WHERE departure_airport IN (
            SELECT airport_code
            FROM airport
            WHERE iso_country = $$$| quote_literal(p_country) ||$$$
        )
        AND bl.booking_id IN (
            SELECT booking_id
            FROM booking
            WHERE update_ts > $$$| quote_literal(p_updated) ||$$$
        );
    $$
END; $body$ LANGUAGE plpgsql;
```

Время выполнения функции на базе *динамического SQL* для каждого набора параметров неизменно, а это именно то, что нужно в промышленных системах. Поскольку SQL создается непосредственно перед выполнением, *оптимизатор не использует кешированный план*. Вместо этого он *заново строит план для каждого выполнения*. Может показаться, что на это требуется дополнительное время, но на самом деле происходит обратное [2, стр. 226].

Если я правильно понял, то чтобы гарантировать, что оптимизатор будет оптимизировать план выполнения при каждом вызове функции, для каждого набора параметров, нужно писать функции с использованием приемов динамического SQL.

Проблема $N + 1$ запроса: большинство ORM по умолчанию откладывают загрузку, поэтому для получения дочерних элементов сначала выдается один запрос в базу данных для родительской записи, а затем по одному запросу для *каждой* дочерней записи [2, стр. 259].

12. Более сложная фильтрация и поиск

Полнотекстовый поиск работает и без индексов, но PostgreSQL предоставляет специальные типы индексов, которые могут его ускорить.

Некоторые виды индексов, например *составные индексы*, могут содержать несколько атрибутов, но в этом случае *важен порядок атрибутов*: индекс бесполезен, если не указано значение первого атрибута [2, стр. 261].

Для пространственных данных часто используются другие критерии поиска. Наиболее распространены:

- запросы по диапазону – найти все объекты, расположенные не дальше определенного расстояния от указанной точки в пространстве,
- запросы ближайших соседей – найти k объектов, ближайших к указанной точке.

Эти запросы не поддерживаются одномерными индексами, и несколько одномерных индексов тоже не помогут.

Оператор `CREATE INDEX` позволяет при создании индекса указать его тип. По умолчанию при создании индекса используется тип `btree`. Кроме этого PostgreSQL поддерживает следующие типы: `hash`, `gist`, `spgist`, `gin` и `brin`.

Тип индекса GiST подойдет, если нужно индексировать точки. Условия поиска выражаются прямоугольником: возвращаются все точки, находящиеся внутри этого прямоугольника.

Если говорить точнее, GiST – это семейство индексных структур, каждая из которых поддерживает определенный тип данных. В дистрибутив PostgreSQL включена поддержка точек; некоторые другие типы данных могут быть установлены как расширения. Конечно, атрибуты, включаемые в индекс GiST, не обязательно должны быть координатами. Такие атрибуты могут предоставлять время или другие упорядоченные типы данных.

В целях индексации документ представляется как список термов (или фрагментов) и представляется значением типа данных `tsvector`. Для каждого терма, содержащегося хотя бы в одном документе, инвертированный индекс (GIN) содержит список документов, содержащих этот терм. Таким образом, общая структура симметрична: у документа есть список термов, а у терма есть список документов.

Инвертированные индексы могут эффективно поддерживать текстовый поиск. Например, чтобы найти документы, содержащие все термы, указанные в запросе, PostgreSQL сканирует все списки документов этих термов и оставляет только те документы, которые присутствуют в каждом из списков. Списки упорядочены, поэтому для получения набора результатов достаточно одного прохода по спискам.

Структура GIN не является производной от естественного языка и никак не связана с ним; документы считаются списками. Таким образом, индекс может работать с данными, отличны-

ми от документов – фактически с любым типом атрибута, содержащим несколько элементов, например с массивами.

12.1. Индексирование очень больших таблиц

Любой индекс занимает некоторое место, а индексы для больших таблиц могут быть очень большими. Различают *плотные* и *разреженные* индексы.

Плотные индексы содержат все значения индексируемого столбца. Разреженный индекс содержит *только часть* всех значений, но *уменьшает количество чтений*, необходимых для поиска *любого* из значений индексируемого атрибута. Разреженные индексы отличаются от *частичных индексов*, которые *не ускоряют* поиск значений, *не включенных* в индекс [2, стр. 264].

Иногда разреженные индексы называют *кластерными*, поскольку строки с одинаковыми значениями индексируемого столбца размещены близко друг к другу. Например, строки, регистрирующие определенные виды событий, будут, скорее всего, добавляться к таблице и будут естественным образом упорядочены по времени добавления, что позволяет использовать для них разреженную индексацию.

Обобщение разреженных индексов, реализованное в PostgreSQL, называется BRIN, что расшифровывается как Block Range Index.

Таблица, для которой создается индекс BRIN, рассматривается как *последовательность зон*, где *каждая зона* состоит из фиксированного числа *смежных блоков*. Для каждой зоны запись индекса BRIN содержит *содвку значений*, находящихся в этой зоне. Например, сводка может содержать минимальное и максимальное значение столбца временной отметки в таблице журнала событий.

Чтобы найти любое значение индексируемого атрибута, достаточно найти соответствующую *зону блоков* (используя индекс), а затем *просканировать все блоки в этой зоне*.

Способ вычисления сводки зависит от типа индексируемого столбца. Для интервалов сводка может быть интервалом, включающим все интервалы, содержащиеся в зоне блоков. Для пространственных данных сводка может быть ограничивающим прямоугольником, содержащим все прямоугольники в зоне блоков.

Если значения столбцов не упорядочены или строки в таблице не упорядочены, индекс BRIN вернет несколько зон блоков, каждую из которых придется просканировать.

Вычисление сводки обходится дорого. Поэтому PostgreSQL предоставляет несколько вариантов сопровождения индекса BRIN: индекс может обновляться с помощью триггеров, сводка может обновляться автоматически вместе с очисткой или может быть пересчитана вручную.

Единственная разница между этими типами заключается в том, что JSONB хранит *двоичное представление данных* JSON, а не строку [2, стр. 265].

Для типа JSON можно создать только индексы на основе B-деревьев по определенным тегам, а для этого нужно указать полный путь в JSON, включая индексы массивов, так что проиндексировать «любой» сегмент бронирования не получится.

Если мы хотим иметь высокопроизводительные индексы для столбцов JSON, необходимо использовать тип JSONB. Индекс GIN строится не мгновенно

```
CREATE INDEX idxgin ON booking_jsonb USING GIN (cplx_booking);
```

13. Замечания по использованию различных баз данных для различных проектов

Небольшому стартапу лучше выбрать одну из реляционных баз данных, например, MySQL или PostgreSQL – они подходят для широкого круга задач. Быстрорастущим компаниям есть смысл использовать сразу две-три СУБД. Например, MySQL или PostgreSQL для длительного хранения данных, а Redis – для быстрой обработки запросов.

Проще и дешевле масштабировать облачные базы данных – для проектов с непредсказуемой нагрузкой лучше всего подходят PostgreSQL или MongoDB в конфигурации, предназначенной для быстрого выделения ресурсов. В случае резкого роста трафика компания получает дополнительные мощности облачной СУБД автоматически или по запросу. При этом оплачиваются только фактически потраченные ресурсы.

Если информации для анализа очень много и требуется чтобы машинный алгоритм мог сделать выводы на их основе, необходимо быстро извлекать данные по заданным критериям. С этой задачей не справятся реляционные базы данных MySQL и PostgreSQL – они долго обрабатывают большие массивы информации. Здесь нужна специальная аналитическая база данных, например, ClickHouse. Она быстро выполняет аналитические запросы.

Некоторые сервисы должны быстро обрабатывать запросы пользователей, в том числе в режиме реального времени – для проведения транзакций, аналитики в реальном времени, счетчиков, аутентификации и других операциях, предполагающих быстрый ответ базы данных. Для таких ситуаций подойдет Redis – нереляционная высокопроизводительная СУБД. Redis хранит данные в оперативной памяти, за счет этого отвечает на запросы в десятки раз быстрее, чем MySQL или PostgreSQL. Redis можно использовать и как самостоятельную СУБД, и как дополнительную СУБД в тех случаях, когда число запросов резко растет и нужно быстро их обработать.

Иногда бизнесу требуется решать узко специализированные задачи. Например, шифровать пароли, составлять рейтинги, анализировать маршруты транспорта или следить за передвижением курьеров в реальном времени. Нужна база данных, позволяющая подключать расширения или использовать команды для решения нестандартных задач.

Для PostgreSQL и Postgres Pro разработано большое количество дополнительных расширений. Например, расширение для криптографии шифрует личные данные для безопасной передачи по сети. В случае кражи злоумышленники получают только обезличенную информацию. А расширение PostGIS подходит для картографических сервисов, например, для поиска по географическим данным.

В Redis есть наборы специальных команд и операторов, позволяющих использовать базу данных для решения узких задач в режиме реального времени. Например, составления рейтингов игроков в играх, аналитики контента, управления геоданными, в том числе отслеживания местоположения транспорта.

Итак:

- У стартапа может смениться бизнес-модель. Поэтому лучше выбрать СУБД, подходящую для решения широкого круга задач, или заложить сразу две-три базы данных,
- В бизнесе, где нагрузка на инфраструктуру зависит от сезона, важно быстро масштабировать базу данных – это удобно делать в облаке,
- Хранить информацию в неизменном виде позволяют реляционные базы данных, например, MySQL или PostgreSQL,

- Для работы с большими данными подойдет ClickHouse – база данных, позволяющая быстро выполнять аналитические запросы,
- Быстро отвечать на запросы пользователей позволяет Redis – нереляционная высокопроизводительная СУБД, она хранит данные в оперативной памяти.

14. Ответвления от PostgreSQL

От PostgreSQL существует более 20 ответвлений. На протяжении многих лет различные группы создавали ответвления и затем включали результаты своей работы в PostgreSQL:

- HadoopDB – гибрид PostgreSQL с технологиями MapReduce, ориентированный на аналитику,
- Greenplum – реляционная СУБД, имеющая массово-параллельную архитектуру без разделения ресурсов. Или еще можно сказать, что Greenplum представляет собой кластер над экземплярами PostgreSQL без разделения ресурсов с массово параллельной обработкой (MPR). Применяется для создания хранилищ данных и аналитики,
- Vertica – столбцовая СУБД ориентированная на аналитику данных. Терминальный клиент для Python можно изучить на странице проекта <https://github.com/vertica/vertica-python>,
- Amazon Redshift – популярное хранилище данных на базе PostgreSQL 8.0.2. Предназначено в основном для OLAP-приложений.

С различными СУБД (реляционными, нереляционными, аналитическими, графовыми, документарными и пр.) удобно работать с помощью кроссплатформенного менеджера баз данных DBeaver <https://dbeaver.io/download/>.

14.1. Базовые сведения о Greenplum

Greenplum – реляционная СУБД с массово-параллельной архитектурой без разделения ресурсов, ориентированная на OLAP-приложения. Другими словами, основная задача Greenplum работа с аналитической нагрузкой. PostgreSQL это все-таки классическая реляционная СУБД ориентированная главным образом на OLTP-приложения, хотя с помощью этой СУБД можно строить и относительно простые OLAP-приложения. Индексы, которые активно используются в PostgreSQL, полезны для OLTP-задач, но бесполезны в OLAP-задачах. Индексы эффективны, когда подмножество строк таблицы составляет небольшую долю от общего числа. В аналитических же задачах приходится работать с широкими длинными таблицами, т.е. с таблицами, у которых несколько сотен столбцов и очень много строк, порядка нескольких сотен тысяч или даже нескольких миллионов строк.

Профиль аналитической нагрузки:

- «Широкие» таблицы (сотни столбцов, много строк),
- Работа большими пакетами данных (чтение, вставка),
- При чтении обрабатывается много строк, но мало столбцов (1 - 10%),
- Обновления и удаления редки,
- Мало параллельных запросов в системе,
- Допустимы задержки при получении результата.

Цель: максимизация пропускной способности запроса.

Кластеры бывают гомогенные (все ноды кластера одинаковые) и с ролями (часть нод выполняют одну роль, часть другую, часть третью). Ноды могут представлять собой либо просто экземпляры СУБД, либо отказоустойчивые группы.

Так вот **Greenplum** это кластер на ролях с отказоустойчивыми группами. Здесь отказоустойчивая группа это пара «Мастер-Резерв» (Dispatch). Она управляет другими группами, которые хранят данные (Execute). В эти группы входят сегмент и зеркало.

Greenplum остается транзакционной системой в отличие от прочих аналитических решений.

Замечание

Greenplum не нужно использовать в OLTP-приложениях, т.е. в приложениях, в которых речь идет о транзакционной нагрузке

Greenplum лучше всего справляется с:

- Сложные запросы, обрабатывающие большие объемы (в том числе сложные аналитические функции и т.д.),
- ETL/ELT,
- Работа с индексами,
- Data Science,
- Аналитические функции на PL,
- Ad-hoc аналитика (когда мы не знаем какой запрос напишет пользователь).

15. Хранилища данных

Хранилище данных (Data Warehouse) – предметно-ориентированная информационная база данных, специально разработанная и предназначенная для подготовки отчетов и бизнес-анализа с целью поддержки принятия решений в организации. Строится на базе систем управления базами данных и систем поддержки принятия решений. Данные, поступающие в хранилища, как правило, доступны только для чтения.

Существует два варианта *обновления* данных в хранилище:

- *полное* обновление данных в хранилище. Сначала старые данные удаляются, потом происходит загрузка новых данных. Процесс выполняется с определенной периодичностью, при этом актуальность данных может несколько отставать от OLTP-системы,
- *инкрементальное* обновление – обновляются только те данные, которые изменились в OLTP-системе.

Принципы организации хранилища:

- *Проблемно-предметная ориентация*: данные объединяются в категории и хранятся в соответствии с областями, которые они описывают, а не с приложениями, которые они используют,
- *Интегрированность*: данные объединены так, чтобы они удовлетворяли всем требованиям предприятия в целом, а не единственной функции бизнеса,
- *Некорректируемость*: данные в хранилище не создаются; то есть данные поступают из внешних источников, не корректируются и не удаляются,
- *Зависимость от времени*: данные в хранилище точны и корректны только в том случае, когда они привязаны к некоторому промежутку или моменту времени.

Существует два архитектурных направления:

- нормализованные хранилища данных,
- хранилища данных с измерениями.

В нормализованных хранилищах, данные находятся в предметно-ориентированных таблицах третьей нормальной формы. Нормализованные хранилища характеризуются как простые в создании и управлении, недостатки нормализованных таблиц – большое количество таблиц как атрибут нормализации, в следствие чего для получения какой-либо информации нужно делать выборку из многих таблиц одновременно, что приводит к снижению производительности системы. Для решения этой проблемы используются денормализованные таблицы – витрины данных, на основе которых уже выводятся отчетные формы. При очень больших объемах данных могут использоваться несколько уровней витрин/хранилищ.

Хранилища с измерениями, как правило, используют схему «звезда» или «снежинка». При этом в центре «звезды» находятся данные (таблица фактов), а измерения образуют лучи звезды. Различные таблицы фактов совместно используют таблицы измерений, что значительно облегчает операции объединения данных из нескольких предметных таблиц фактов. Основным недостатком является более сложные процедуры подготовки и загрузки данных, а также управление и изменение измерений данных.

16. Основные ошибки при работе с PostgreSQL

Отличный доклад Алексея Лесовского <https://www.youtube.com/watch?v=HjLnY0aPQZo&list=PL2SBCjiIRNXBIARbUE-BprXkH-HhZ2xr4&index=7>.

Чаще всего при работе с PostgreSQL не хватает дискового пространства или дисковой производительности.

Есть OLTP-нагрузка, которая характеризуется быстрыми, легкими, короткими запросами, и OLAP-нагрузка, характеризующаяся медленными, долгими, тяжелыми аналитическими запросами, которые вычитывают большие объемы данных.

Практика показывает, что нужно разносить OLTP и OLAP части, хотя бы потому, что OLAP-запросы блокируют OLTP-запросы.

PostgreSQL поддерживает *потокковую* и *логическую* репликацию. В отличие от потоковой репликации, при которой двоичные данные копируются побитово, механизм логической репликации преобразует WAL-файлы в набор логических изменений. Логическая репликация позволяет скопировать часть данных, тогда как в случае потоковой репликации ведомый узел является точной копией ведущего.

Организовать независимую работу OLAP и OLTP запросов можно с помощью *потокковой репликации* (поддерживаются синхронная, асинхронная и каскадная репликации). В каскадном режиме ведомый узел является ведущим для другой реплики. Это позволяет горизонтально масштабировать PostgreSQL на операциях чтения.

Можно использовать механизм *логической репликации*². Когда есть несколько независимых PostgreSQL-серверов со своими базами. Эти базы можно подключать в соседние базы. Все изменения источников будут реплицироваться на базу назначения и там будут видны.

Другой подход, *декларативное партиционирование* и механизм внешних таблиц. Пусть есть несколько PostgreSQL-серверов, которые хранят данные за какие-то диапазоны. Сводную базу данных, имеющую доступ к партициям хранят на мастер-сервере.

²Еще называют механизмом логических публикаций и подписок

И, наконец, эти схемы можно комбинировать.

Начать имеет смысл с разнесения нагрузки. В мастер идет запись, а с реплик выполняется чтение. Для выполнения сложных аналитических запросов нужна отдельная реплика.

Еще важно помнить, что базы данных это все-таки не хранилище, они больше ориентированны на конкурентный доступ. Не нужно строить индексы на все случаи жизни. Индексы снижают время обновления и в случае аварии время восстановления. Если строится индекс, то должна быть железная уверенность в том, что он там нужен.

Основные рекомендации:

- Базу данных следует размещать на SSD. Это надежное, устойчивое решение,
- Нельзя писать в базу данных все подряд,
- Важно проводить партиционирование (секционирование) данных,
- Очень важно организовать мониторинг³; нельзя использовать конфигурационные файлы с настройками по умолчанию, важно уметь конфигурировать PostgreSQL,
- Надо стараться разнести потоки чтения и записи, т.е. лучше не читать и не писать в одно и то же место; PostgreSQL очень хорошо масштабируется,
- Очень важно контролировать ничего не делающие транзакции, они снижают производительность и убивают базу,

17. Файл подкачки

Подкачка страниц представляет собой механизм виртуальной памяти, перемещающий неиспользуемые (неактивные) фрагменты памяти в другое хранилище (жесткий диск или какое-либо другое место), тем самым освобождая место в оперативной памяти для загрузки активных данных. В качестве хранилища могут выступать раздел подкачки или файл подкачки (swap-файл). Раздел подкачки обычно создается в момент установки операционной системы, а файл подкачки можно создать в любой момент, главное чтобы было место.

Пространство подкачки главным образом используется для расширения виртуальной памяти за пределы установленной оперативной памяти (RAM).

Ресурс твердотельных накопителей (SSD) напрямую зависит от числа циклов записи/перезаписи, поэтому не рекомендуется использовать swap на SSD.

18. Первичные и внешние ключи

Ограничение внешнего ключа указывает, что значения столбца (или группы столбцов) должны соответствовать значениям в некоторой строке другой таблицы. Или другими словами ограничение внешнего ключа обеспечивает *ссылочную целостность* связанных таблиц.

Ограничение первичного ключа означает, что образующий его столбец или группа столбцов может быть уникальным идентификатором строк в таблице. Или другими словами обеспечивает *сущностную целостность*.

```
CREATE TABLE products (  
  product_no integer PRIMARY KEY, -- первичный ключ уровня столбца  
  name text,  
  price numeric
```

³Можно следить за статусом клиентов, ошибками (появились ошибки, есть повод заглянуть в log), качественными/количественными показателями запросов (нет ли медленных запросов, например)


```
);

CREATE TABLE orders (
  order_id integer PRIMARY KEY, -- первичный ключ уровня столбца
  shipping_address text,
  ...
);

CREATE TABLE order_items (
  product_no integer REFERENCES products ON DELETE RESTRICT, -- внешний ключ уровня столбца (ограничивающее удаление)
  order_id integer REFERENCES orders ON DELETE CASCADE, -- внешний ключ уровня столбца (каскадное удаление)
  quantity integer,
  PRIMARY KEY (product_no, order_id) -- составной первичный ключ уровня таблицы
);
```

Ограничивающие и каскадные удаления — два наиболее распространённых варианта. **RESTRICT** предотвращает удаление связанной строки. **NO ACTION** означает, что если зависимые строки продолжают существовать при проверке ограничения, возникает ошибка (это поведение по умолчанию). Главным отличием этих двух вариантов является то, что **NO ACTION** позволяет отложить проверку в процессе транзакции, а **RESTRICT** — нет.

CASCADE указывает, что при удалении связанных строк зависимые от них будут так же автоматически удалены. Есть ещё два варианта: **SET NULL** и **SET DEFAULT**. При удалении связанных строк они назначают зависимым столбцам в подчинённой таблице значения **NULL** или значения по умолчанию, соответственно. Если в качестве действия задано **SET DEFAULT**, но значение по умолчанию не удовлетворяет ограничению внешнего ключа, операция закончится ошибкой.

19. Буферный кеш в PostgreSQL

Размер кеша устанавливается параметром **shared_buffers**. Значение по умолчанию смехотворное — 128 Мб. Это один из параметров, которые имеет смысл увеличить сразу же после установки PostgreSQL.

```
SELECT setting, unit
FROM pg_settings
WHERE name = 'shared_buffers';
```

```
+-----+-----+
| setting | unit |
+-----+-----+
| 16384   | 8kB  |
+-----+-----+
```

Надо иметь в виду, что изменение параметра требует перезапуска сервера, поскольку вся необходимая под кеш память выделяется при старте сервера.

Оптимальное значение буферного кеша зависит данных, приложения, нагрузки и т.п. Стандартная рекомендация — взять в качестве первого приближения $1/4$ оперативной памяти, а дальше по ситуации.

Исключение из общего правила представляют временные таблицы. Поскольку временные данные видны только одному процессу, им нечего делать в общем буферном кеше. Более того, вре-

менные данные существуют только в рамках одного сеанса, так что их не нужно защищать от сбоя.

Для временных данных используется кеш в локальной памяти того процесса, который владеет таблицей. Поскольку такие данные доступны только одному процессу, их не требуется защищать блокировками. В локальном кеше используется обычный алгоритм вытеснения.

В отличие от общего буферного кеша, память под локальный кеш выделяется по мере необходимости, ведь временные таблицы используются далеко не во всех сеансах. Максимальный объем памяти для временных таблиц одного сеанса ограничен параметром `temp_buffers`.

Для справки перезапустить сервер на Windows, используя командную оболочку `bash`, можно с помощью утилиты `pg_ctl`⁴ так

```
pg_ctl restart -D '/c/Program Files/PostgreSQL/11/data'
```

или так, если использовать `cmd.exe`

```
pg_ctl restart -D "C:\Program Files\PostgreSQL\11\data"
```

А посмотреть статус можно так

```
pg_ctl status -D "C:\Program Files\PostgreSQL\11\data"
```

Для справки все изменения (например, изменение номера порта), вносимые в файл `postgresql.conf` требуют перезапуска сервера, т.е. `pg_ctl restart -D "C:\Prog...\data"`.

Помимо `set_config`, PostgreSQL предоставляет для изменения конфигурационных параметров команду `ALTER SYSTEM` со следующим синтаксисом

```
ALTER SYSTEM SET configuration_parameter { TO | = } { value | 'value' | DEFAULT }
ALTER SYSTEM RESET configuration_parameter
ALTER SYSTEM RESET ALL
```

Иногда для вступления значения в силу требуется перезагрузка системы или перезапуск

```
$ SHOW work_mem; -- 4MB
$ ALTER SYSTEM SET work_mem TO '8MB';
$ SELECT pg_reload_conf();
$ SHOW work_mem; -- 8MB
```

Получить список параметров, значения которым присвоены в файле `postgresql.conf`, а не по умолчанию можно так

```
SELECT name, current_setting(name), source
FROM pg_settings
WHERE source IN ('configuration file');
```

Получить размер базы данных можно с помощью специальных функций

```
SELECT datname, pg_size_pretty(pg_database_size(datname)) AS size
FROM pg_database;
```

Выведет

```
+-----+-----+
| datname | size |
+-----+-----+
| postgres | 8393 kB |
```

⁴Еще перезагрузить кластер после внесения изменений в файл `pg_hba.conf` или `postgresql.conf` можно с помощью функции `pg_reload_conf()`

```
| demo          | 304 MB |
| template1    | 7769 kB |
+-----+-----+
```

Функция `pg_total_relation_size()` возвращает размер таблицы со всеми индексами и TOAST-таблицами. Если интересует только размер самой таблицы, то следует воспользоваться функцией `pg_relation_size()`. Эта информация поможет следить за ростом таблицы и табличных пространств

```
SELECT pg_size_pretty(pg_total_relation_size('table'));
```

Наконец, размер индекса также вернет функция `pg_relation_size()`

```
SELECT indexrelid::regclass, pg_size_pretty(pg_relation_size(indexrelid::regclass))
FROM pg_index
LIMIT 2;
```

20. План выполнения запроса и команда EXPLAIN

У команды `EXPLAIN` есть несколько опций; при наличии опции `ANALYZE` команда выполняется и возвращает фактическое время и количество строк.

Для поддержания статистики базы данных в актуальном состоянии важно, чтобы работал процесс автоочистки.

Разумеется, если статистика неправильная, то ожидать хороших планов выполнения не приходится. После *массовой загрузки данных* в базу рекомендуется выполнить команду `ANALYZE` для обновления статистики.

Рассмотрим пример плана выполнения запроса

```
-----
Seq Scan on <table_name> (cost=0.00..1834.00 rows=100000 width=37)
```

Стоимость запроса (`cost`) – это оценка затрат на его выполнение. В этом примере 0.00 – стоимость выборки *первой строки*, 1834.00 – стоимость выборки *всех* строк, вычисляемая по формуле $(relpages * seq_page_cost) + (reltuples * cpu_tuple_cost)$. Количество страниц отношения (`relpages`) и строк отношения (`reltuples`) можно найти в таблице `pg_class`. Величины `seq_page_cost` и `cpu_tuple_cost` – это конфигурационные параметры планировщика. Таким образом мы получаем число 1834.

В этом простом примере последовательного просмотра вычислить стоимость не составляет труда. Но если запрос включает вычисление предикатов, группировку, сортировку и соединение, то получить оценку стоимости гораздо сложнее. Наконец, величина 37 в плане выполнения – это средняя ширина кортежа в байтах, которую можно найти в таблице `pg_stats`.

Чтобы выполнить запрос и одновременно получить его стоимость, имеется команда `EXPLAIN ANALYZE`. План запроса следует читать снизу вверх, начиная строки с наибольшим отступом.

Выполнить запрос и вывести его план можно и так

```
EXPLAIN (ANALYZE, FORMAT YAML, BUFFERS) SELECT * FROM <table_name>;
```

выведет что-то похожее на

```
+-----+
|          QUERY PLAN          |
+-----+
```

```

| - Plan:                                     +|
|   Node Type: "Seq Scan"                   +|
|   Parallel Aware: false                   +|
|   Relation Name: "test_json"              +|
|   Alias: "test_json"                     +|
|   Startup Cost: 0.00                      +|
|   Total Cost: 1.03                       +|
|   Plan Rows: 3                           +|
|   Plan Width: 105                        +|
|   Actual Startup Time: 0.019+            +|
|   Actual Total Time: 0.021               +|
|   Actual Rows: 3                         +|
|   Actual Loops: 1                        +|
|   Shared Hit Blocks: 1                    +| <-- NB!
|   Shared Read Blocks: 0                  +|
|   Shared Dirtied Blocks: 0               +|
|   Shared Written Blocks: 0              +|
|   Local Hit Blocks: 0                    +|
|   Local Read Blocks: 0                   +|
|   Local Dirtied Blocks: 0                +|
|   Local Written Blocks: 0                +|
|   Temp Read Blocks: 0                    +|
|   Temp Written Blocks: 0                 +|
|   Planning Time: 0.105                   +|
|   Triggers:                              +|
|   Execution Time: 0.048                  |
+-----+

```

Строка «Shared Hit Blocks: 1» (разделяемые блоки в кеше) означает, что данные читаются из памяти, а не с диска.

20.1. Обнаружение проблем в планах выполнения запросов

Команда EXPLAIN может показать, почему запрос работает медленно, особенно если выполнять ее с опциями BUFFER и ANALYZE. Вот на что следует обращать внимание, пытаясь понять, хорош план выполнения или нет:

- *Оценка числа строк и истинное число строк.* Это важно, потому что планировщик ориентируется на эту величину, выбирая метод выполнения запроса. Есть два случая: оценка числа строк завышена или занижена. Неправильная оценка влияет на все алгоритмы: выборки данных с диска, сортировки, соединения и т.д. Вообще говоря, завышенная оценка – это нехорошо, но *заниженная* – гораздо хуже. С одной стороны, при выполнении соединения очень больших таблиц методом вложенных циклов время выполнения растет экспоненциально, если для простоты предположить, что стоимость этого алгоритма равно $O(n^2)$. С другой стороны, соединение с небольшой таблицей методом хеширования замедляет выполнение запроса, но эффект оказывается не таким чудовищным, как в первом случае.
- *Сортировка в памяти или на диске.* Сортировка выполняется при наличии таких фраз, как DISTINCT, LIMIT, ORDER, GROUP BY, и некоторых других. Если памяти достаточно, то сортировка производится в памяти, иначе на диске.
- *Буферный кеш.* Важно обращать внимание на то, сколько данных находится в буферах и сколько из них грязных. Чтение данных из буфера значительно повышает производительность.

Замечание

Если план выполнения длинный и сложный и трудно понять в чем «затык», то можно попробовать запретить алгоритм с экспоненциальным временем работы, например *соединение методом вложенных циклов*

Еще один очень любопытный момент связан с *межстолбцовой корреляцией*. Из-за межстолбцовой корреляции возможны ошибки в оценке количества строк, поскольку PostgreSQL предполагает, что все столбцы независимы. Но на самом деле это далеко не всегда так. Пример – страна и язык в предпочтениях клиентов.

Чтобы избежать проблем, связанных с межстолбцовой корреляцией, следует объединять такие столбцы в один с типом `jsonb`.

В PostgreSQL 10 добавлена перекрестная статистика столбцов, так что больше менять физическую структуру таблицы не нужно. Однако необходимо уведомить планировщик о наличии таких зависимостей. Чтобы расширить статистику, выполним такую команду

```
CREATE STATISTICS stats(dependencies) ON country, language FROM client;
```

21. Сброс пароля для psql и pgAdmin4

Для того чтобы доступ к базам данных через терминальный клиент `psql` или через веб-интерфейс `pgAdmin4` можно было выполнять без ввода пароля, нужно сделать следующее:

- найти файл `pg_hba.conf`; на ОС Windows он располагается по адресу `C:\Program Files\PostgreSQL\11\data`,
- заменить в этом файле метод `md5` (в нижней части файла) на `trust`.

После исправлений файл `pg_hba.conf` должен выглядеть приблизительно так

pg_hba.conf

```
...
# TYPE DATABASE USER ADDRESS METHOD
# IPv4 local connections:
host all all 127.0.0.1/32 trust
host all all 0.0.0.0/0 trust
# IPv6 local connections:
host all all ::1/128 trust
host all all ::0/0 trust
# Allow replication connections from localhost, by a user with the
# replication privilege.
host replication all 127.0.0.1/32 trust
host replication all ::1/128 trust
host appdb app all trust
```

22. Создание резервной копии базы данных

Задачу резервного копирования можно запускать с выделенного backup-сервера или с сервера базы данных. В комплекте PostgreSQL есть 2 утилиты, которые позволяют делать резервные копии: `pg_dump`, `pg_dumpall` (сохраняют резервную копию базы данных в текстовом файле или другом виде) и `pg_basebackup` (делает базовую резервную копию *работающего* сервера PostgreSQL).

Кроме того есть возможность использовать утилиты файлового копирования, такие `rsync`, `tar`, `cp` и т.п.

Утилита `pg_dump` подходит для случаев, когда нужно сделать резервную копию таблицы, базы или схемы, а `pg_basebackup` – подходит для случаев, когда нужно сделать резервную копию целиком всего кластера базы данных или настроить реплику. Утилиты `rsync`, `tar`, `cp` также используются для случаев копирования всего кластера.

Примеры создания резервной копии базы данных

```
pg_dump -U postgres -d demo --format=tar -v -f $(date +%Y-%m-%d).demo.tar
pg_dump -U postgres -d demo -f $(date +%Y-%m-%d).demo.sql
```

Создать резервную копию с backup-сервера в каталог `/backup` (каталог должен существовать)

```
pg_basebackup -x -h db01.example.com -U backup -D /backup
```

Копирование со сжатием в `bzip2`, для случаев, когда нужно использовать нестандартный алгоритм сжатия

```
pg_basebackup -x --format=tar -h db01.example.com -U backup -D - | bzip2 -9 > /backup/db01/
backup-$(date +%Y-%m-%d).tar.bz2
```

Копирование со сжатием в несколько потоков (задействуем 8 ядер)

```
pg_basebackup -x --format=tar -h db01.example.com -U backup -D - | lzip -n 8 -9 > /backup/db01/
backup-$(date +%Y-%m-%d).tar.bz2
```

Копирование запускается на сервере базы данных. Формируемая резервная копия отправляется на удаленный сервер по `ssh`

```
pg_basebackup -x --format=tar -h 127.0.0.1 -U backup -D - | \
ssh backup@backup.example.com "tar -xf - -C /backup/"
```

Восстановить данные из созданных backup-файлов можно с помощью утилиты `pg_restore`.

23. Логический порядок обработки инструкции SELECT

Порядок обработки инструкции `SELECT` определяет, когда объекты, определенные в одном шаге, становятся доступными для предложений в последующих шагах. Например, если обработчик запросов можно привязать к таблицам или представлениям, определенным в предложении `FROM`, эти объекты и их столбцы становятся доступными для всех последующих шагов.

Общая процедура выполнения `SELECT` следующая (подробности см. в документации [SELECT](#)):

1. `WITH`: выполняются все запросы в списке `WITH`; по сути они формируют временные таблицы, к которым затем можно обращаться в списке `FROM`; запрос в `WITH` выполняется только один раз, даже если он фигурирует в списке `FROM` неоднократно,
2. `FROM`: вычисляются все элементы в списке `FROM` (каждый элемент в списке `FROM` представляет собой реальную или виртуальную таблицу); другими словами конструируются таблицы из списка `FROM`,
3. `ON`: выбираются строки, удовлетворяющие заданному условию,
4. `JOIN`: выполняется объединение таблиц,
5. `WHERE`: исключаются строки, не удовлетворяющие заданному условию,

6. **GROUP BY**: вывод разделяется по группам строк, соответствующим одному или нескольким значениям, а затем вычисляются результаты агрегатных функций,
7. **HAVING**: исключаются группы, не удовлетворяющие заданному условию,
8. **SELECT**,
9. **DISTINCT**: исключаются *повторяющиеся* строки; **SELECT DISTINCT ON** исключает строки, совпадающие по всем указанным выражениям; **SELECT ALL** (по умолчанию) возвращает все строки результата, включая дубликаты,
10. **UNION**, **INTERSECT** и **EXCEPT**: объединяется вывод нескольких команд **SELECT** в один результирующий набор.
11. **ORDER BY**: строки сортируются в указанном порядке; в отсутствие **ORDER BY** строки возвращаются в том порядке, в каком системе будет проще их выдавать,
12. **LIMIT** (или **FETCH FIRST**), либо **OFFSET**: возвращается только подмножество строк результата.
13. Если указано **FOR UPDATE**, **FOR NO KEY UPDATE**, **FOR SHARE** или **FOR KEY SHARE**, оператор **SELECT** блокирует выбранные строки, защищая их от одновременных изменений.

24. Смена схемы базы данных

Вывести список доступных схем

```
SHOW search_path;
```

Задать схему

```
SET search_path TO new_schema;
```

или, если требуется доступ к нескольким схемам

```
SET search_path TO new_schema1, new_schema2, public;
```

25. Секционирование

26. Транзакции и управление параллельным доступом

В реляционной модели описана логическая единица обработки данных – *транзакция*. Можно сказать, что транзакция – это множество последовательно выполняемых операций. Реляционная СУБД предоставляет механизм блокировки, гарантирующий целостность транзакций.

Транзакция – это множество операций, в состав которого могут входить операции обновления, удаления, вставки и выборки данных. Часто эти операции погружаются в язык более высокого уровня или явно обертываются в блок транзакций, заключенный между командами **BEGIN** и **END**.

Транзакция считается успешно выполненной, если успешно выполнены все составляющие ее операции. Если какая-то операция транзакции завершается неудачно, то частично выполненные действия можно откатить.

Для явного управления транзакциями можно поставить команду **BEGIN** в ее начале и команду **END** или **COMMIT** в конце. В следующем примере показано, как выполнить команду SQL в транзакции

```
BEGIN;  
CREATE TABLE emloyee(  
    id serial primary key,
```

```
name text,  
salary numeric  
);  
COMMIT; -- зафиксировать изменения
```

Помимо обеспечения целостности данных транзакции позволяют легко отменить изменения, внесенные в базу в процессе разработки и отладки. В интерактивном режиме блок транзакции можно сочетать с командой **SAVEPOINT**, чтобы поставить отметку в точке сохранения состояния.

Точка сохранения – это способ откатить не всю транзакцию целиком, а только ее часть. В примере ниже демонстрируется использование **SAVEPOINT**

```
BEGIN;  
UPDATE employee SET salary = salary*1.1;  
SAVEPOINT increase_salary; -- точка сохранения  
UPDATE employee SET salary = salary + 500 WHERE name = 'join';  
ROLLBACK TO increase_salary; -- возврат к точке сохранения  
COMMIT;
```

Этим приемом очень удобно пользоваться в транзакциях, изменяющих данные в таблице. Сначала ставим точку сохранения с помощью **SAVEPOINT**, затем проводим операцию, изменяющую данные, а затем либо фиксируем результат с помощью **COMMIT**, либо откатываемся с помощью **ROLLBACK**.

Рекомендации:

- Никогда не следует отключать процесс очистки (**VACUUM**⁵), иначе база данных рано или поздно остановится из-за циклического оборачивания идентификатора,
- Команды массовой вставки или обновления следует заключать в явные блоки транзакций,
- Следует иметь в виду, что активно обновляемые таблицы могут «разбухать» из-за большого числа мертвых строк.

27. Уровня изоляции транзакций

Разработчик может задать уровень изоляции транзакции, выполнив такую команду

```
BEGIN TRANSACTION ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ  
UNCOMMITTED };
```

Что такое уровень изоляции, проще объяснить, описав побочные эффекты, возникающие на каждом уровне:

- *грязное чтение* – это происходит, когда транзакция читает данные из кортежа, который был модифицирован другой транзакцией и еще не зафиксирован. В PostgreSQL грязное чтение невозможно, так как PostgreSQL не поддерживает уровень изоляции транзакций **READ UNCOMMITTED**, который разрешает транзакции читать незафиксированные данные,
- *неповторяемое чтение* – это происходит, если в одной транзакции некоторая строка читается дважды и при этом получаются разные результаты. Такое бывает, если уровень изоляции равен **READ COMMITTED**, и зачастую является следствием многократного обновления строки другими транзакциями,
- *фантомное чтение* – это происходит, если на протяжении транзакции новая строка (или строки) сначала появляется, а потом исчезает. Часто это результат зафиксированной вставки, за которой следует зафиксированное удаление,

⁵Для того чтобы почистить разбухшие таблицы и индексы нужно просто выполнить команду **VACUUM**

- *аномалия сериализации* – результат выполнения группы транзакций зависит от порядка их выполнения. Это может случиться только на уровне REPEATABLE READ.

28. Создание таблицы

28.1. Базовые синтаксис создания таблицы

Для создания таблиц в языке SQL служит команда CREATE TABLE. Упрощенный синтаксис таков

```
CREATE TABLE table_name(  
    field_name data_type [constraint],  
    field_name data_type [constraint],  
    ...  
    [constraint],  
    [primary key],  
    [foreign key]  
);
```

Пример

```
CREATE TABLE aircrafts(  
    aircraft_code CHAR(3) NOT NULL,  
    model TEXT NOT NULL,  
    range INTEGER NOT NULL,  
    CHECK (range > 0),           -- ограничение  
    PRIMARY KEY (aircraft_code) -- первичный ключ  
);
```

Создать таблицу с *автоматическим* вычислением идентификационного номера строки

```
CREATE TABLE test(  
    id SERIAL PRIMARY KEY,  
    name TEXT  
);
```

Теперь при вставке строки без явного указания идентификационного номера последний будет вычисляться автоматически⁶

```
INSERT INTO test(name) VALUES ('Ansys') RETURNING id;
```

ВЫВЕДЕТ

```
+----+  
| id |  
+----+  
|  1 |  
+----+
```

Еще возвращенное значение можно было бы прочитать с помощью *анонимной функции*

```
DO $$ -- анонимная функция  
    DECLARE -- объявление переменной  
        auto_generated_id INT;  
    BEGIN  
        -- значение атрибута id, которое возвращает RETURNING  
        -- можно связать с переменной auto_generated_id с помощью ключевого слова INTO
```

⁶Если при вставке строки имена некоторых атрибутов явно не указываются, то считается, что значения этих атрибутов вычисляются автоматически, в противном случае они имеют значение <NULL>

```
INSERT INTO test(name) VALUES ('Nastran') RETURNING id INTO auto_generated_id;
RAISE NOTICE 'The primary key is: %', auto_generated_id;
END
$$;
```

Выведет

```
NOTICE: The primary key is: 2
```

28.2. Создать таблицу по образу другой таблицы

Создать таблицу со структурой данных, аналогичной другой таблице (но без ограничений базовой таблицы) можно так

```
CREATE TABLE tbl_name AS
SELECT * FROM base_tbl LIMIT 0;
```

или более короткий вариант

```
CREATE TABLE tbl_name(LIKE base_tbl);
```

29. Подзапросы

По связанности подзапросы делятся на:

- *связанные* (или коррелированные) подзапросы, т.е. такие подзапросы, которые ссылаются на элементы внешнего подзапроса или элементы главного запроса,
- *несвязанные* (или некоррелированные) подзапросы.

Фундаментальная концепция состоит в том, что связанные подзапросы выполняются для *каждой* записи (строки) из внешнего подзапроса (или главного запроса), а несвязанные выполняются только один раз.

29.1. Латеральные подзапросы

Очень удобно использовать подзапросы в списке выборке. Например, в командах **SELECT** с их помощью можно создавать вычисляемые атрибуты при опросе таблицы. Пример с использованием скалярных подзапросов

```
SELECT
  car_id,
  manufacture_year,
  CASE WHEN manufacture_year <= (
    SELECT avg(manufacture_year) FROM car_portal_app.car
    WHERE car_model_id = c.car_model_id
  ) THEN 'old' ELSE 'new'
  END AS age,
  (SELECT count(*) FROM car_portal_app.car
   WHERE car_model_id = c.car_model_id) AS same_model_count
FROM car_portal_app.car AS c;
```

Эти подзапросы интересны тем, что могут ссылаться на главную таблицу в своей фразе **WHERE**. С помощью подобных подзапросов легко добавить в запрос дополнительные столбцы. Но есть проблема – производительность. Таблица **car** просматривается сервером один раз в главном запросе, а затем еще два раза для каждой выбранной строки, т.е. для столбцов **age** и **same_model_count**.

Другими словами, выбирается значение атрибута `car_model_id` таблицы `car` подзапроса для каждой строки таблицы `car` главного запроса. Для каждой строки главного запроса вычисляется скалярный подзапрос, связанный с псевдонимом `age`, и возвращающий для каждой строки год производства автомобиля, усредненный по группе марок автомобиля. А затем аналогичная процедура повторяется для второго подзапроса, связанного с псевдонимом `same_model_count`.

Конечно, можно независимо вычислить эти агрегаты по одному разу для каждой модели, а затем соединить результаты с таблицей `car`

```
SELECT
    car_id,
    manufacture_year,
    CASE
        WHEN manufacture_year <= avg_year THEN 'old'
        ELSE 'new'
    END AS age,
    same_model_count
FROM car_portal_app.car
    INNER JOIN (
        SELECT
            car_model_id,
            avg(manufacture_year) AS avg_year,
            count(*) AS same_model_count
        FROM car_portal_app.car
        GROUP BY car_model_id
    ) AS subq
    USING (car_model_id);
```

Результат тот же самый, а запрос выполняется значительно быстрее. Однако этот запрос хорош только тогда, когда нужно выбрать из базы данных много строк. Если нужно получить информацию только об одном автомобиле, то первый запрос будет быстрее.

Существует еще один способ использования подзапросов. Он сочетает в себе преимущества подзапросов в списке выборки, способных обращаться к главной таблице из фразы `WHERE`, с подзапросами во фразе `FROM`, которые могут возвращать несколько столбцов. Если поместить ключевое слово `LATERAL` перед подзапросом во фразе `FROM`, то он сможет ссылаться на любой элемент, предшествующий ему во фразе `FROM`.

Запрос выглядит следующим образом

```
SELECT
    car_id,
    manufacture_year,
    CASE
        WHEN manufacture_year <= avg_year THEN 'old'
        ELSE 'new'
    END AS age,
    same_model_count
FROM
    car_portal_app.car AS c,
    LATERAL ( -- латеральный связанный подзапрос; выполняется для каждой строки
               -- таблицы 'car' внешнего запроса
        SELECT
            avg(manufacture_year) AS avg_year,
            count(*) AS same_model_count
        FROM car_portal_app.car
        WHERE car_model_id = c.car_model_id
    ) AS subq; -- у подзапроса во фразе FROM обязательно должен быть псевдоним
```

Итак, ключевое слово **LATERAL** позволяет ссылаться в подзапросах на столбцы предшествующих элементов списка **FROM**. Без **LATERAL** каждый подзапрос выполняется независимо и поэтому не может обращаться к другим элементам **FROM**.

Замечание

В контексте фразы **FROM** по умолчанию подзапросы не могут ссылаться на другие элементы фразы **FROM**

Когда элемент **FROM** содержит ссылки **LATERAL**, запрос выполняется следующим образом: сначала для строки элемента **FROM** с целевыми столбцами, или набора строк из нескольких элементов **FROM**, содержащих целевые столбцы, вычисляется элемент **LATERAL** со значением этих столбцов. Затем результирующие строки обычным образом соединяются со строками, из которых они были вычислены. Эта процедура повторяется для всех строк исходных таблиц.

Применять **LATERAL** имеет смысл в основном, когда для вычисления соединяемых строк необходимо обратиться к столбцам других таблиц.

Особенно полезно бывает использовать **LEFT JOIN** с подзапросом **LATERAL**, чтобы исходные строки оказались в результате, даже если подзапрос **LATERAL** не возвращает строк. Например, если функция `get_product_names()` выдает названия продуктов, выпущенных определенным производителем, но о продукции некоторых производителей информации нет, мы можем найти, каких именно, примерно так:

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

Рассмотренный запрос работает приблизительно в два раза быстрее первого, он оптимален, когда нужно выбрать из таблицы **car** всего одну строку.

Синтаксис **JOIN** также допускается в латеральных подзапросах, хотя в большинстве случаев условие соединения каким-то образом включается во фразу **WHERE** подзапросов.

Употреблять **LATERAL** с функциями, возвращающими множества, нет необходимости. Все функции, упомянутые во фразе **FROM**, и так могут использовать результаты любых предшествующих функций или подзапросов.

29.2. Дополнительные средства группировки

Базы данных часто служат источником данных для различных отчетов. А в отчетах принято в одной и той же таблице показывать подытоги, итоги и общие итоги, подразумевающие группировку и агрегирование. Рассмотрим отчет о количестве объявлений в разрезе марок и кварталов, в котором требуется также показать итоги по каждому кварталу (суммарно по всем маркам) и общий итог. Вот как отбираются данные для такого отчета

```
SELECT
    to_char(advertisement_date, 'YYYY-Q') AS quarter,
    make,
    count(*)
FROM advertisement a
    INNER JOIN car c ON a.car_id = c.car_id
    INNER JOIN car_model m ON m.car_model_id = c.car_model_id
GROUP BY quarter, make;
```

выведет

quarter	make	count
2014-4	Peugeot	12
2014-2	Daewoo	8
2014-4	Skoda	5
...		

Чтобы вычислить итоги, понадобятся дополнительные запросы

```
SELECT to_char(advertisement_date, 'YYYY-Q') AS quarter, count(*)
FROM advertisement a
      INNER JOIN car c ON a.car_id = c.car_id
      INNER JOIN car_model m ON m.car_model_id = c.car_model_id
GROUP BY quarter;

SELECT count(*)
FROM advertisement a
      INNER JOIN car c ON a.car_id = c.car_id
      INNER JOIN car_model m ON m.car_model_id = c.car_model_id;
```

PostgreSQL позволяет объединить все три запроса в один с помощью специальной конструкции GROUP BY GROUPING SETS

```
SELECT
  to_char(advertisement_date, 'YYYY-Q') AS quarter,
  make,
  count(*)
FROM advertisement a
      INNER JOIN car c ON a.car_id = c.car_id
      INNER JOIN car_model m ON m.car_model_id = c.car_model_id
GROUP BY GROUPING SETS ((quarter, make), (quarter), ()) -- << NB
ORDER BY quarter NULLS LAST, make NULLS LAST;
```

Благодаря конструкции GROUPING SETS ((quarter, make), (quarter), ()) запрос работает как UNION ALL одинаковых запросов с разными фразами GROUP BY:

- GROUP BY quarter, make,
- GROUP BY – здесь поле make будет иметь значение NULL,
- все строки помещаются в одну группу, оба поля – quarter и make – будут иметь значение NULL.

Вообще говоря, фраза GROUP BY принимает не только выражения, но и группирующие элементы, которые могут быть выражениями или конструкциями типа GROUPING SETS.

Два других группирующих элемента – ROLLUP и CUBE

- ROLLUP (a, b, c) эквивалентно GROUPING SETS ((a, b, c), (a, b), (c), ()).
- CUBE (a, b, c) эквивалентно GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ()) – все возможные комбинации аргументов.

Разрешено также комбинировать различные группирующие элементы в одной фразе GROUP BY, например: GROUP BY a, CUBE (b, c) – по сути дела, комбинация группирующих элементов.

29.3. Дополнительные виды агрегирования

Есть несколько агрегатных функций, которые выполняются специальным образом. Первая группа таких функций называется *агрегатами по упорядоченному множеству*. Они учитывают не только значения выражений, переданных в качестве аргументов, но и их порядок.

PostgreSQL позволяет вычислить непрерывный или дискретный процентиль с помощью функций `percentile_cont` и `percentile_disc` соответственно.

Дискретный процентиль совпадает с одним из значений группы, а непрерывный – результат интерполяции двух существующих значений. Есть возможность вычислить один процентиль для заданной процентной доли или сразу несколько percentилей для заданного массива долей.

Вот, например, запрос о распределении количества объявлений по автомобилям

```
SELECT percentile_disc('{0.25, 0.5, 0.75}'::real[]) WITHIN GROUP (ORDER BY range)
FROM aircrafts;
```

Выведет

```
+-----+
| percentile_disc |
+-----+
| {3000,5600,6700} |
+-----+
```

Результат означает, что для 25 процентов наблюдений *не превосходят* 3000, 50 процентов – 5600, а 75 процентов – 6700.

Синтаксис агрегатных функций по упорядоченному множеству отличается от обычных агрегатных функций, в нем используется специальная конструкция `WITHIN GROUP (ORDER BY expression)`. Здесь выражение `expression` – фактически аргумент функции. На результат функции влияет не только порядок строк, но и значения этих выражений. В противоположность фразе `GROUP BY` в команде `SELECT` допускается только одно выражение, и ссылок на номера результирующих столбцов быть не должно.

Еще один достойный упоминания аспект агрегатных функций – фраза `FILTER`. Она по заданному условию отфильтровывает строки, передаваемые агрегатной функции. Например, пусть требуется подсчитать количество автомобилей для каждой модели и каждого значения количества дверей

```
SELECT
  car_model_id,
  count(*) FILTER (WHERE number_of_doors = 2) doors2,
  count(*) FILTER (WHERE number_of_doors = 3) doors3,
  count(*) FILTER (WHERE number_of_doors = 4) doors4,
  count(*) FILTER (WHERE number_of_doors = 5) doors5
FROM car_portal_app.car
GROUP BY car_model_id;
```

Выведет

```
car_model_id | doors2 | doors3 | doors4 | doors5
-----+-----+-----+-----+-----
          43 |      0 |      0 |      0 |      2
           8 |      0 |      0 |      1 |      0
          11 |      0 |      2 |      1 |      0
          80 |      0 |      1 |      0 |      0
          ...
```

Того же результата можно достичь и таким образом

```
count(CASE WHEN number_of_doors = 2 THEN 1 END) doors2
```

но вариант с фразой `FILTER` короче и проще.

Еще один пример на использование фразы `FILTER`

```
SELECT
    sum(amount) FILTER (WHERE fare_conditions = 'Business') AS business,
    sum(amount) FILTER (WHERE fare_conditions = 'Comfort') AS comfort,
    sum(amount) FILTER (WHERE fare_conditions = 'Economy') AS economy,
FROM ticket_flights;
```

30. Копирование данных между файлом и таблицей

Скопировать данные из внешнего файла в таблицу (по сути загрузить данные) можно с помощью команды COPY. С помощью этой же команды можно записать данные из таблицы в файл.

Общий синтаксис команды выглядит так

```
-- копирует содержимое файла в таблицу
COPY имя_таблицы [ ( имя_столбца [, ...] ) ]
    FROM { 'имя_файла' | PROGRAM 'команда' | STDIN }
    [ [ WITH ] ( параметр [, ...] ) ]

-- копирует содержимое таблицы в файл
COPY { имя_таблицы [ ( имя_столбца [, ...] ) ] | ( запрос ) }
    TO { 'имя_файла' | PROGRAM 'команда' | STDOUT }
    [ [ WITH ] ( параметр [, ...] ) ]
```

Здесь допускается параметр:

```
FORMAT имя_формата
OIDS [ boolean ]
FREEZE [ boolean ]
DELIMITER 'символ_разделитель'
NULL 'маркер_NULL'
HEADER [ boolean ]
QUOTE 'символ_кавычек'
ESCAPE 'символ_экранирования'
FORCE_QUOTE { ( имя_столбца [, ...] ) | * }
FORCE_NOT_NULL ( имя_столбца [, ...] )
FORCE_NULL ( имя_столбца [, ...] )
ENCODING 'имя_кодировки'
```

Примеры

```
-- сохранить данные из таблицы 'family' в файл 'family.csv'
postgres=# COPY family TO 'E:/[WorkDirectory]/GARBAGE/family.csv' DELIMITER ',';

-- загрузить в таблицу 'family' данные из файла 'family.csv'
postgres=# CREATE TABLE family (
    person TEXT PRIMARY KEY,
    parent TEXT REFERENCES family -- создать внешний ключ на person
);
postgres=# COPY family FROM 'E:/[WorkDirectory]/GARBAGE/family.csv' DELIMITER ',';
```

31. Обновление записей. Команда UPDATE

Изменить слово Drama на Dramatic в столбце kind таблицы films

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

Изменить значение температуры и сбросить уровень осадков к значению по умолчанию в одной строке таблицы `weather`

```
UPDATE
  weather
SET
  temp_lo = temp_lo + 1,
  temp_hi = temp_lo + 15,
  prcp = DEFAULT
WHERE
  city = 'San Francisco' AND
  dates = '2003-07-03';
```

32. Общие табличные выражения

В конструкциях общих табличных выражений с `WITH` имена временных таблиц указываются без перечисления имен столбцов, а в конструкциях с `WITH RECURSIVE` – с перечислением, например, `WITH RECURSIVE tab(col1, col2, ...) AS (...)`.

Ссылки на общие табличные выражения в главном запросе можно трактовать как имена таблиц. PostgreSQL выполняет общее табличное выражение *только один раз*, кеширует результаты, а затем повторно его использует, вместо того чтобы выполнять подзапросы всякий раз, как они встречаются в главном запросе [1, стр. 169].

Порядок выполнения CTE не определен. Цель PostgreSQL – выполнить *главный запрос*. Если в нем есть ссылки на CTE, то PostgreSQL выполняет сначала их. Если на CTE типа `SELECT` нет ни прямых, ни косвенных ссылок из главного запроса, то оно не выполняется вовсе. Подкоманды, *изменяющие данные*, выполняются *всегда*.

В PostgreSQL 12 по умолчанию CTE *не материализуются*! В нематериализованных CTE используются индексы.

Теперь есть возможность выбрать поведение CTE в зависимости от контекста

```
WITH temp_tbl AS MATERIALIZED (
  ...
)
-- или
WITH temp_tbl AS NOT MATERIALIZED (
  ...
)
```

Однако, если CTE используется несколько раз, то оно будет материализовано в любом случае. Чтобы этого избежать, следует явно указывать `NOT MATERIALIZED`.

32.1. Конструкция WITH

Основное предназначение `SELECT` в предложении `WITH` (Common Table Expression, Общие Табличные Выражения) заключается в разбиении сложных запросов на простые части. Например, пусть задана некоторая таблица `orders`⁷

```
WITH --part 1, common table expression
  regional_sales AS ( --def temp_table1
    SELECT region, sum(amount) AS total_sales
    FROM orders --base table
```

⁷См. документацию PostgreSQL <https://postgrespro.ru/docs/postgrespro/9.5/queries-with>


```

        GROUP BY region
    ),
    top_regions AS ( --def temp_table2
        SELECT region
        FROM regional_sales --temp_table1
        WHERE total_sales > (
            SELECT SUM(total_sales)/10
            FROM regional_sales --temp_table2
        )
    )
SELECT --part 2
    region,
    product,
    SUM(quantity) AS product_units,
    SUM(amount) AS product_sales
FROM orders
WHERE region IN (
    SELECT region
    FROM top_regions --temp_table2
)
GROUP BY region, product;

```

Здесь в инструкции WITH объявляются две *временные таблицы* `regional_sales` и `top_regions`. Вторая временная таблица `top_regions` ссылается на временную таблицу `regional_sales`, сформированную в первых строках настоящего запроса. Во второй части запроса также используется временная таблица `top_regions`.

Еще один пример. Пусть задана таблица

```
# SELECT * FROM test_tab;
```

id	cae_name	solver	num_cores
1	ANSYS	Direct	32
3	Comsole	Direct	16
4	LMS Virtual Lab	Direct	32
2	Nastran	Iterativ	16

(4 строки)

Требуется выяснить сколько CAE-пакетов имеют прямой, а сколько итерационный решатель. Эту задачу можно решить следующим образом

```

WITH sub_tab AS ( --make temp table
    SELECT solver, 1 AS count
    FROM test_tab
)
SELECT solver, sum(count)
FROM sub_tab --link to temp table
GROUP BY solver;

```

Часть с WITH возвращает

```
# SELECT solver, 1 AS count FROM test_tab;
```

solver	count
Direct	1
Direct	1
Direct	1
Iterativ	1

(4 строки)

Полезный пример с использованием конструкции CASE...END и WHEN...THEN

```

WITH cte_film AS ( --part 1
    SELECT
        film_id,
        title,
        (CASE --start block
            WHEN length < 30 THEN 'Short'
            WHEN length < 90 THEN 'Medium'
            ELSE 'Long'
        END) length
    FROM
        film
)
SELECT --part 2
    film_id,
    title,
    length
FROM
    cte_film
WHERE
    length = 'Long'
ORDER BY
    title;

```

Пример с использованием логических операторов

```

WITH cte_films AS (
    SELECT
        film_id,
        title,
        (CASE
            WHEN length < 30 THEN 'Short'
            WHEN length >= 30 AND length < 90 THEN 'Medium'
            WHEN length > 90 THEN 'Long'
        END) length
    FROM
        film
)

```

32.2. Конструкция WITH RECURSIVE

Если к WITH добавить RECURSIVE, то можно будет получить доступ к промежуточному результату. Например,

```

WITH RECURSIVE tbl(n) AS ( --part 1
    SELECT 1 --or VALUES(1). This is nonrecursive part
    UNION ALL
    SELECT n+1 FROM tbl WHERE n < 10 --and this is recursive part
)
SELECT sum(n) from tbl; --part 2

```

На первой итерации в таблице `tbl` в атрибуте `n` находится значение 1. На этом вычисления некурсивной части заканчиваются. Далее переходим к вычислениям в рекурсивной части. Таблица `tbl` ссылается на последнее вычисленное значение, поэтому на второй итерации удастся выполнить `n+1`, после чего новым значением таблицы `tbl` станет 2 (`tbl -> 2`). Проверяем условие `n < 10`, а затем переходим к следующей итерации и т.д.

Удобно представлять, что вычисленные значения хранятся в некоторой промежуточной области в порядке вычисления, а таблица `tbl` всегда ссылается на последнее вычисленное значение.

На последнем этапе 1 объединяется с 2, 3 и т.д., т.е. в итоге получается последовательность от 1 до 10. Во второй части запроса остается лишь просуммировать элементы этой последовательности и вывести на экран.

Рассмотрим еще такой пример

```
WITH RECURSIVE
  included_parts(sub_part, part, quantity) AS (
    SELECT --nonrecursive part
      sub_part,
      part,
      quantity
    FROM parts --base table
    WHERE part = "our_product"
    UNION ALL
    SELECT --recursive part
      p.sub_part,
      p.part,
      p.quantity
    FROM
      included_parts pr,
      parts p
    WHERE p.part = pr.sub_part
  )
SELECT sub_part, SUM(quantity) AS total_quantity
FROM included_parts
GROUP BY sub_part
```

На первой итерации временная таблица `included_parts`, вычисленная в некурсивной части, представляет собой результат выборки строк и столбцов из таблицы `parts`. В рекурсивной части можно получить доступ к этой таблице. В завершении выполняем выборку из таблицы `included_parts` по столбцу `sub_part`, группируем по нему и выводим сумму по `quantity`.

Еще один полезный пример. Пусть дана таблица сотрудников

employees				
id	name	salary	job	manager_id
1	John	10000	CEO	null
2	Ben	1400	Junior Developer	5
3	Barry	500	Intern	5
4	George	1800	Developer	5
5	James	3000	Manager	7
6	Steven	2400	DevOps Engineer	7
7	Alice	4200	VP	1
8	Jerry	3500	Manager	1
9	Adam	2000	Data Analyst	8
10	Grace	2500	Developer	8
11	Leor	5000	Data Scientist	6

Выведем иерархию подчинения сотрудников в компании

```
WITH RECURSIVE managers(id, name, manager_id, job, level) AS (
  SELECT id, name, manager_id, job, 1
  FROM employees --base table
  WHERE id = 7
  UNION ALL
  SELECT e.id, e.name, e.manager_id, e.job, m.level+1
  FROM employees e JOIN managers m ON e.manager_id = m.id
)
```

```
SELECT * FROM managers;
```

Сначала в *некурсивной* части WITH RECURSIVE объявляется временная таблица `managers(id, name, ...)`. Она строится на базе таблицы `employees`, к которой слева добавляется столбец, состоящий из одних единиц. Затем выбираются строки, удовлетворяющие условию WHERE; в данном случае это одна строка `e.manager_id=m.id`.

И, таким образом, на данном этапе во *временную* таблицу `managers` попадет только одна строка

managers, вычисленная в некурсивной части

id	name	manager_id	job	level
7	Alice	1	VP	1

Переходим в рекурсивную часть CTE. Из базовой таблицы `employees` выбираем те строки, которые в столбце `manager_id` имеют те же значения, что и в столбце `id` временной таблицы `managers` (на данном этапе таблица состоит из одной строки). Другими словами, выбрать нужно те строки, у которых в столбце `manager_id` таблицы `employees` стоит цифра 7.

В результате временная таблица `managers` на текущем этапе будет иметь вид

managers, вычисленная в рекурсивной части

id	name	manager_id	job	level
5	James	7	Manager	2
6	Steven	7	DevOps Engineer	2

Временные таблицы *рекурсивных общих табличных выражений* всегда ссылаются на результат последних вычислений, т.е. на данном этапе временная таблица `managers` ссылается на таблицу, состоящую из двух строк.

Теперь мы снова выбираем из базовой таблицы `employees` и временной таблицы те строки, у которых в столбцах `e.manager_id` и `m.id` стоят одинаковые числа (в данном случае 5 и 6).

Таким образом

managers, вычисленная в рекурсивной части на 2-ой итерации

id	name	manager_id	job	level
2	Ben	5	Junior Developer	3
3	Barry	5	Intern	3
4	George	5	Developer	3
11	Leor	6	Data Scientist	3

Наконец все временные подтаблицы «склеиваются» и конструкция `SELECT * FROM managers` возвращает таблицу `managers`

id	name	manager_id	job	level
7	Alice	1	VP	1
5	James	7	Manager	2
6	Steven	7	DevOps Engineer	2
2	Ben	5	Junior Developer	3
3	Barry	5	Intern	3
4	George	5	Developer	3

32.3. Изменение данных в WITH

В предложении WITH можно также использовать операторы, изменяющие данные (INSERT, UPDATE или DELETE). Это позволяет выполнять в одном запросе сразу несколько разных операций. Например

```
WITH moved_rows AS (
  DELETE FROM products
  WHERE
    dates >= '2010-10-01' AND
    dates < '2010-11-01'
  RETURNING *
)
INSERT INTO products_log (SELECT * FROM moved_rows);
```

Этот запрос фактически перемещает строки из таблицы `products` в таблицу `products_log` (таблица должна уже существовать на момент выполнения запроса). Оператор `DELETE` удаляет указанные строки из `products` и возвращает их содержимое в предложении `RETURNING`, а затем главный запрос читает это содержимое и вставляет в таблицу `products_log`.

33. Оконные функции

Помимо группировки и агрегирования, PostgreSQL предлагает еще один способ вычислений, затрагивающих несколько записей, — *оконные функции*. В случае группировки и агрегирования выводится одна запись для каждой группы входных записей. Оконные функции делают нечто похожее, но выполняются для каждой записи, а количество записей на входе и на выходе одинаковое.

При работе с оконными функциями группировка не обязательна, хотя и возможна. Оконные функции вычисляются *после группировки и агрегирования*. Поэтому в запросе `SELECT` они могут находиться только в списке выборки и во фразе `ORDER BY`.

33.1. Определение окна

Синтаксис оконных функций выглядит так

```
<function_name> (<function_arguments>)
OVER (
  [PARTITION BY <expression_list>]
  [ORDER BY <order_by_list>]
  [{ROWS | RANGE} <frame_start> |
   {ROWS | RANGE} BETWEEN <frame_start> AND <frame_end>]
)
```

Конструкция в скобках после слова `OVER` называется *определением окна*. Последняя его часть, начинающаяся словом `ROWS`, называется *определением фрейма*. Ключевое слово `OVER` делает агрегатную функцию оконной.

Набор записей, обрабатываемых оконной функцией, строится следующим образом. Вначале обрабатывается фраза `PARTITION BY`. Отбираются все записи, для которых выражения, перечисленные в списке `expression_list`, имеют такие же значения, как и в текущей записи. Это

множество строк называется *разделом* (partition). Текущая строка также включается в раздел. В общем-то, фраза **PARTITION BY** логически и синтаксически идентична фразе **GROUP BY**, только в ней запрещается ссылаться на выходные столбцы по именам или по номерам.

Иными словами, при обработке каждой записи оконная функция просматривает все остальные записи, чтобы понять, какие из них попадают в один раздел с текущей. Если фраза **PARTITION BY** отсутствует, то на данном шаге создается один раздел, содержащий все записи.

Далее раздел сортируется в соответствии с фразой **ORDER BY**, логически и синтаксически идентичной той же фразе в команде **SELECT**. И снова ссылки на выходные столбцы по имени или по номерам не допускаются. Если фраза **ORDER BY** опущена, то считается, что позиция каждой записи множества одинакова.

Наконец, обрабатывается определение фрейма. Это означает, что мы берем подмножество раздела, которое следует передать оконной функции. Оно и называется оконным фреймом. У фрейма есть начало и конец.

Начало фрейма может принимать следующие значения:

- **UNBOUNDED PRECEDING**: самая первая запись раздела,
- **<N> PRECEDING**: запись, предшествующая текущей в упорядоченном разделе и отстоящая от нее на *N* записей. Здесь **<N>** – целочисленное выражение, которое не может быть отрицательным и не может содержать агрегатные или другие оконные функции. **0 PRECEDING** указывает на текущую запись,
- **CURRENT ROW**: текущая строка,
- **<N> FOLLOWING**: запись, следующая за текущей в упорядоченном разделе и отстоящая от нее на *N* записей.

Конец фрейма может принимать следующие значения:

- **<N> PRECEDING**,
- **CURRENT ROW**,
- **<N> FOLLOWING**,
- **UNBOUNDED FOLLOWING**: последняя запись раздела.

Оконный фрейм можно определить в режиме **ROWS** или **RANGE**. Режим влияет на семантику **CURRENT ROW**. В режиме **ROWS** конструкция **CURRENT ROW** указывает на саму текущую запись, а в режиме **RANGE** – на первую или последнюю запись с такой же позицией, что у текущей, в смысле сортировки, заданной фразой **ORDER BY**.

В режиме **RANGE** в определении фрейма можно использовать варианты **BOUNDED . . .** или **CURRENT ROW**, но не **<N> PRECEDING**.

Если часть **frame_end** опущена, предполагается **CURRENT ROW**. Если опущено все определение фрейма, то предполагается определение **RANGE UNBOUNDED PRECEDING**.

Например, конструкция

OVER (PARTITION BY a ORDER BY b ROWS BETWEEN UNBOUNDED PRECEDING AND 5 FOLLOWING)

означает, что для каждой строки раздел образуют все записи с одинаковым значением в поле **a**. Затем раздел сортируется в порядке возрастания поля **b**, а фрейм содержит все записи от первой до пятой после текущей строки.

33.2. Фраза WINDOWS

Определения окон могут быть довольно длинными, и часто использовать их в списке выборки неудобно. PostgreSQL предлагает способ определять и именовать окна, которые затем используются во фразе `OVER` оконной функции. Это делается с помощью фразы `WINDOW` команды `SELECT`, которая может находиться после фразы `HAVING`, например

```
SELECT count() OVER w, sum(b) OVER w, avg(b) OVER (w ORDER BY c ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
FROM table1
WINDOW w AS (PARTITION BY a) -- определение окна
```

Определенное таким образом окно можно использовать как есть. В примере выше функции `count` и `sum` так и делают. Но его можно и уточнить, как сделано в функции `avg`.

Синтаксически разница состоит в следующем: чтобы повторно использовать определение окна, имя этого окна следует указать после ключевого слова `OVER` без скобок. Если же мы хотим расширить определение окна, то имя окна следует указать внутри скобок.

33.3. Использование оконных функций

Все агрегатные функции, в т.ч. определенные пользователем, можно использовать как оконные, за исключением агрегатов по упорядоченным наборам и по гипотетическим наборам. На то, что функция выступает в роли оконной, указывает наличие фразы `OVER`.

Если агрегатная функция используется в качестве оконной, то она агрегирует строки, принадлежащие *оконному фрейму текущей строки*.

Пример

```
-- общее табличное выражение выполняется перед главным запросом
WITH monthly_data AS ( -- CTE
    SELECT date_trunc('month', advertisement_date) AS month,
           count(*) AS cnt -- число строк для каждой группы
    FROM car_portal_app.advertisement
    GROUP BY date_trunc('month', advertisement_date)
    -- чтобы не дублировать записи во фразах SELECT и GROUP BY, во фразе GROUP BY
    -- можно сослаться на первый элемент из списка выборки по номеру
    -- т.е. можно было бы написать GROUP BY 1
)
SELECT
    to_char(month, 'YYYY-MM') AS month,
    cnt,
    sum(cnt) OVER ( -- накопительная сумма
        w ORDER BY month
    ) AS cnt_year,
    round(
        avg(cnt) OVER ( -- скользящее среднее
            ORDER BY month
            ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING
        ),
        1) AS mov_avg,
    -- для того чтобы деление было вещественным нужно, чтобы хотя бы один операнд
    -- был приведен к вещественному типу
    -- round работает только с типом numeric
    round((cnt::numeric / sum(cnt) OVER w) * 100, 2) AS ratio_year -- здесь sum(cnt) вычисляется
    для всего раздела без учета порядка следования записей в таблице
FROM monthly_data -- обращение к результату работы CTE
WINDOW w AS (PARTITION BY date_trunc('year', month)); -- определение окна
```

Выведет

month	cnt	cnt_year	ratio_year
2014-01	42	42	5.78
2014-02	49	91	6.74
2014-03	30	121	4.13
2014-04	57	178	7.84
2014-05	106	284	14.58
2014-06	103	387	14.17
...			

Здесь общее табличное выражение вычисляется перед главным запросом, так как имя временной таблицы `monthly_data`, связанной с CTE, присутствует в главном запросе. Результат кешируется и используется повторно.

Сначала, как обычно вычисляются объекты, указанные в предложении `FROM`, т.е. в данном случае таблица `advertisement` (дополнительно указана схема базы данных `car_portal_app`). Группировка выполняется по временной метке «усеченной» до месяца (т.е. в формате `'2020-07-01 00:00:00+03'`).

В предложении `SELECT` конструкция, указанная во фразе `GROUP BY`, включается полностью, без изменений и связывается с псевдонимом `month`. Как известно, в *список выборки* фразы `SELECT` в случае группировки могут включаться только атрибуты, указанные в `GROUP BY` и агрегатные функции.

Результат работы общего табличного выражения будет доступен через имя `monthly_data`.

Переходим к главному запросу. Здесь читается временная таблица `monthly_data` и из нее выбираются атрибуты, перечисленные в списке выборки. Функция `to_char` преобразует временную метку к формату `'YYYY-MM'` и связывается с псевдонимом `month`. Атрибут `cnt` выбирается как есть, без изменений.

Далее, первая оконная функция `sum` использует окно `w`, описанное с помощью ключевого слова `WINDOW` и расширенное `ORDER BY`. Определение фрейма в данном случае опущено, поэтому предполагается фрейм `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. Следовательно, функция вычисляет сумму значений по всем записям, начиная с начала года и до текущего месяца включительно, т.е. накопительный итог за год.

Вторая функция `avg` для каждой записи вычисляет скользящее среднее по пяти записям – две предшествующие, текущая и две последующие. Ранее определенное окно не используется, потому что при вычислении скользящего среднего год не принимается во внимание, важен только порядок записей.

Третья оконная функция `sum` пользуется тем же определением окна `w`. Вычисленная ею сумма значений за год является знаменателем в выражении, которое дает вклад текущего месяца в сумму. Очень важный момент заключается в том, что в данном случае фраза `ORDER BY` опущена и это значит, что накопительная сумма вычисляется независимо от порядка следования записей в таблице, и потому возвращается только одно число, равное накопительной сумме по разделу, а не как в первом случае вычисления накопительной суммы (там использовалась фраза `ORDER BY`).

Замечание

В агрегатной функции `count` можно использовать ключевое слово `DISTINCT`, чтобы вывести только уникальные строки; например,

```
SELECT count(*), count(DISTINCT name_id) FROM table;
```

Существует ряд *оконных функций*, не являющихся *агрегатными*. Они служат для получения значений других записей в разделе, для вычисления ранга текущей записи относительно остальных и для генерации номеров строк.

Изменим предыдущий отчет, будет считать, что требуется вычислить две разности: между количеством объявлений в текущем месяце и предыдущем месяце того же года и в том же месяце прошлого года, а также ранг текущего месяца. Вот как выглядит подзапрос

```
WITH monthly_data AS ( -- CTE, выполняется прежде главного запроса
    SELECT date_trunc('month', advertisement_date) AS month,
           count(*) AS cnt
    FROM car_portal_app.advertisement
    GROUP BY date_trunc('month', advertisement_date)
)
SELECT to_char(month, 'YYYY-MM') AS month,
       cnt,
       cnt - lag(cnt) OVER (ORDER BY month) AS prev_m,
       cnt - lag(cnt,12) OVER (ORDER BY month) AS prev_y,
       rank() OVER (w ORDER BY cnt DESC) AS rank
FROM monthly_data
WINDOW w AS (PARTITION BY date_trunc('year', month))
ORDER BY month DESC;
```

Выведет

```
+-----+-----+-----+-----+-----+
| month | cnt | prev_m | prev_y | rank |
+-----+-----+-----+-----+-----+
| 2015-02 | 9 | -39 | -40 | 2 |
| 2015-01 | 48 | 11 | 6 | 1 |
| 2014-12 | 37 | 1 | <NULL> | 10 |
| 2014-11 | 36 | -10 | <NULL> | 11 |
| 2014-10 | 46 | -36 | <NULL> | 8 |
| 2014-09 | 82 | 12 | <NULL> | 3 |
...
```

Функция `lag` возвращает значение указанного выражения для записи, отстоящей на заданное число записей (по умолчанию 1) назад от текущей записи. Другими словами, функция `lag` просто смещает значения в столбце на заданное число записей *вниз*, замещая отсутствующие значения в верхней части с помощью `<NULL>`.

Пример

```
-- здесь создается один раздел, упорядоченные по столбцу 'n'
SELECT n,
       lag(n) OVER (ORDER BY n) AS shift
FROM generate_series(10,15) AS t(n);
```

Выведет

```
+---+-----+
| n | shift |
+---+-----+
| 10 | <NULL> |
```

```
| 11 | 10 |  
| 12 | 11 |  
...
```

Как видно, в феврале 2015 года опубликовано 9 объявлений – на 39 меньше, чем в январе 2015.

Функция `rank` возвращает ранг текущей строки внутри раздела. Имеется в виду ранг с промежутками, т.е. в случае, когда две записи занимают одинаковую позицию в порядке, заданном фразой `ORDER BY`, обе получают одинаковый ранг, а следующая получает ранг на две единицы больше. То есть у нас будет две первые записи и одна третья.

Перечислим другие оконные функции:

- `lead`: аналогична `lag`, но возвращает значения выражения, вычисленное для записи, отстоящей от текущей на указанное количество записей вперед, т.е. смещает элементы столбца вверх, замещая отсутствующие значения в нижней части с помощью `<NULL>`,
- `first_value`, `last_value`, `nth_value`: возвращает значения выражения, вычисленные соответственно для первой, последней и n -ой записи,
- `row_number`: возвращает номер текущей строки в разделе,
- `dense_rank`: возвращает ранг текущей строки без промежутков (плотный ранг),
- `percent_rank` и `cume_dist`: возвращает относительный ранг текущей строки. Разница между функциями состоит в том, что в первой числителем дроби является ранг, а во второй – номер строки,
- `ntile`: делит раздел на заданное количество равных частей и возвращает номер части, в которую попала текущая строка.

34. Продвинутые методы работы с SQL

34.1. Выборка первых записей

Часто бывает необходимо найти первые записи относительно какого-то критерия. Допустим, к примеру, что в базе данных `car_portal` нужно найти первое объявление для каждого идентификатора `car_id` в таблице `advertisement`.

```
SELECT  
    advertisement_id,  
    advertisement_date,  
    adv.car_id,  
    seller_account_id  
FROM  
    car_portal_app.advertisement AS adv  
INNER JOIN  
    (SELECT  
        car_id,  
        min(advertisement_date) AS min_date  
    FROM  
        car_portal_app.dvertisement  
    GROUP BY car_id) AS first  
ON adv.car_id = first.car_id  
AND adv.advertisement_date = first.min_date;
```

Но если логика упорядочения настолько сложна, что для ее реализации функции `min` недостаточно, то такой подход работать не будет. Проблему могут решить оконные функции, но они не всегда удобны

```
SELECT DISTINCT
    first_value(advertisement_id) OVER w AS advertisement_id,
    min(advertisement_date) OVER w AS advertisement_date,
    car_id,
    first_value(seller_account_id) OVER w AS seller_account_id
FROM
    car_portal_app.advertisement
WINDOW w AS (PARTITION BY car_id ORDER BY advertisement_date);
```

PostgreSQL предлагает явный способ выбрать первую запись из каждой группы – ключевое слово `DISTINCT ON`. Для каждой уникальной комбинации значений из списка выражений команды `SELECT` возвращает только первую запись. Для определения того, что такое «первая» запись, служит фраза `ORDER BY`

```
SELECT DISTINCT ON (car_id) advertisement_id, advertisement_date, car_id, seller_account_id
FROM car_portal_app.advertisement
ORDER BY car_id, advertisement_date;
```

Другой пример. Пусть дана следующая таблица

month	cnt
2014-01-10 00:00:00	42
2014-02-10 00:00:00	49
2014-03-10 00:00:00	30
2014-04-10 00:00:00	57
2014-05-10 00:00:00	106
2014-06-10 00:00:00	103
2014-07-10 00:00:00	69
2014-08-10 00:00:00	70
2014-09-10 00:00:00	82
2014-10-10 00:00:00	46
2014-11-10 00:00:00	36
2014-12-10 00:00:00	37
2015-01-10 00:00:00	48
2015-02-10 00:00:00	9

Требуется из каждой группы, сформированной по году, выбрать первую запись. Записи упорядочиваются по атрибуту `month`

```
SELECT DISTINCT ON (date_trunc('year', month)) cnt, to_char(month, 'YYYY-MM') AS month
FROM monthly_data;
```

Выведет

cnt	month
42	2014-01
48	2015-01

То есть конструкция `DISTINCT ON` выполняет группировку по указанному атрибуту. И фраза `ORDER BY` не обязательна.

35. Пользовательские типы данных

В PostgreSQL есть два способа определить пользовательский тип данных:

- о команда `CREATE DOMAIN`: позволяет создавать пользовательские типы данных с ограничениями с целью сделать исходный код более модульным,
- о команда `CREATE TYPE`: часто используется для создания составного типа, что полезно в процедурных языках, где такие типы служат для возврата значений. Эта же команда может создавать тип `ENUM`, позволяющий уменьшить количество соединений со справочными таблицами.

Объекты доменов, как и другие объекты базы данных, должны иметь уникальные имена в пределах схемы. Основное применение доменов – в качестве образцов. Рассмотрим, к примеру, текстовый тип, который не может принимать значение `NULL` и содержать пробелы. Это образец.

```
first_name TEXT NOT NULL,  
last_name TEXT NOT NULL,  
CHECK(first_name !~ 's' AND last_name !~ 's')
```

Вместо ограничений можно создать домен

```
CREATE DOMAIN text_without_space_and_null AS TEXT NOT NULL CHECK (value !~ 's');
```

Теперь этот домен можно использовать при создании таблиц

```
CREATE TABLE test_domain(  
  test_att text_without_space_and_null  
);
```

Домен можно изменить командой `ALTER DOMAIN`. Если в домен добавляется новое ограничение, то все атрибуты, определенные с помощью этого домена, будут проверены на соблюдение этого ограничения. При желании проверку старых значений можно подавить, а затем почистить таблицы вручную. Допустим, что мы хотим добавить в домен `text_without_space_and_not_null` ограничение на длину текста

```
ALTER DOMAIN text_without_space_and_null ADD CONSTRAINT text_without_space_and_null_length_chk  
CHECK (length(value) <= 15);
```

Эта команда завершиться неудачно, если какой-нибудь атрибут, принадлежащий этому домену, длиннее 15 символов. Если мы хотим все же применить ограничение к новым данным, а старые пока оставить как есть, то это можно сделать

```
-- указание NOT VALID допускается только для ограничений CHECK  
ALTER DOMAIN text_without_space_and_null  
ADD CONSTRAINT  
  -- указывается имя ограничения  
  text_without_space_and_null_length_chk CHECK (length(value) <= 15) NOT VALID;
```

После вычистки старых данных можно будет проверить для них ограничение, выполнив команду `ALTER DOMAIN ... VALIDATE CONSTRAINT`.

```
ALTER DOMAIN  
  text_without_space_and_null -- имя существующего домена  
VALIDATE CONSTRAINT  
  text_without_space_and_null_length_chk; -- имя ограничения
```

Наконец, метакоманда `psql \dD+` выводит описание домена.

Составные типы данных очень полезны для создания функций, особенно когда возвращаемый тип представляет собой строку из нескольких значений. Например, нам нужна функция, возвращающая `seller_id`, `seller_name`, количество объявлений и полный ранг продавца. Создадим новый тип

```
CREATE TYPE seller_information AS (seller_id INT, seller_name TEXT, number_of_advertisements BIGINT, total_rank FLOAT);
```

Воспользуемся новым типом данных для возврата значения из функции

```
CREATE OR REPLACE FUNCTION seller_information(account_id INT)
RETURNS seller_information AS
$$
SELECT seller_account.seller_account_id, first_name || last_name AS seller_name, count(*), sum(
    rank)::float/count(*)
FROM account
INNER JOIN
    seller_account ON account.account_id = seller_account.account_id
LEFT JOIN
    advertisement ON advertisement.seller_account_id = seller_account.seller_account_id
LEFT JOIN
    advertisement_rating ON advertisement.advertisement_id = advertisement_rating.advertisement_id
WHERE account.account_id = $1
GROUP BY seller_account.seller_account_id, first_name, last_name
$$ LANGUAGE sql;
```

Команду `CREATE TYPE` можно использовать также для определения перечисления `ENUM`

```
-- создать составной тип
CREATE TYPE rank AS ENUM ('poor', 'fair', 'good', 'very good', 'excellent');
-- вывести описание типа перечисления
SELECT enum_range(null::rank);
```

Теперь в столбец с типом `rank` можно будет вставить только значение из списка, указанного в описании типа `rank`.

36. Работа со строками и регулярные выражения

Больше информации про строковые функции и операторы можно найти на страницах официальной документации PostgreSQL по ссылке <https://postgrespro.ru/docs/postgrespro/9.6/functions-string>.

Пусть дана таблица `employees` вида

id	name	salary	job	manager_id
1	John	10000	CEO	
2	Ben	1400	Junior Developer	5
3	Barry	500	Intern	5
4	George	1800	Developer	5
5	James	3000	Manager	7
6	Steven	2400	DevOps Engineer	7
7	Alice	4200	VP	1
8	Jerry	3500	Manager	1
9	Adam	2000	Data Analyst	8
10	Grace	2500	Developer	8
11	Leor	50000	Data Scientist	6

Выбрать те строки из столбца `job`, в которых содержатся строковые значения, удовлетворяющие шаблону `'_ata %'`, означающий, что первый символ строки может быть любым, а после пробелов может не быть ни одного символа или быть сколько угодно символов. То есть символ «`_`» совпадает с любым символом, а символ «`%`» совпадает с произвольным количеством символов. Здесь используется предложение `LIKE`, которое чувствительно к регистру. В качестве альтернативного варианта можно использовать предложение `ILIKE`⁸, которое не учитывает регистр.

```
SELECT * FROM employees WHERE job LIKE '_ata %';
```

Выведет

id	name	salary	job	manager_id
9	Adam	2000	Data Analyst	8
11	Leor	50000	Data Scientist	6

Подобные задачи можно решать и с помощью предложения `SIMILAR TO`, которое похоже на `LIKE`, но в отличие от последнего при интерпретации шаблонов использует определение регулярного выражения стандарта `SQL`. Регулярные выражения `SQL` – это смесь нотации предложения `LIKE` и нотации регулярных выражений.

Шаблоны и `LIKE`, и `SIMILAR TO` должны соответствовать *всей* строке целиком, что, вообще говоря, не согласуется с концепцией обычных регулярных выражений, когда шаблон может соответствовать любой части строки.

`SIMILAR TO`, как и `LIKE` использует символ «`_`» и символ «`%`», что соответствует `.` и `*` в регулярных выражениях `POSIX`. Дополнительно поддерживаются символы `|` (указывает альтернативные варианты), `*` (указывает, что стоящий слева элемент повторяется ноль или более раз), `+` (указывает, что стоящий слева элемент повторяется один или более раз), `(...)` (могут использоваться для указания групп), а `[...]` (определяют символьный класс как в `POSIX`). Однако `?` и `{...}` не поддерживаются и кроме того «`.`» не является метасимволом.

Символ «`\`» экранирует метасимволы, т.е. «снимает» их специальное значение. Другой символ для экранирования можно задать с помощью предложения `ESCAPE`.

Рассмотренную выше задачу можно решить с помощью `SIMILAR TO` следующим образом

```
SELECT * FROM employees WHERE job SIMILAR TO '_ata (A/S)%';
```

Здесь символ «`%`», означающий произвольную последовательность символов, обязателен, так как шаблоны `SIMILAR TO` должны совпадать со *всей строкой*.

Очень полезна бывает функция `substring()`. Как и `SIMILAR TO` шаблон должен совпадать со всей строкой, например

```
SELECT name, job FROM employees
WHERE substring(job from '%#"Dev#"%' for '#')='Dev';
```

Последовательность символов `#"...#` задают левую и правую скобки группы (можно указать и какой-то другой символ в качестве скобки, например, `:"...:"`, но его нужно указать в `for ':'`). Последовательность, попавшая между этих символов будет возвращена. Как и раньше символы «`%`» здесь нужны для того чтобы шаблон совпадал со всей строкой.

Обрезать строку можно так

```
SELECT name, job, substring(job,1,4) FROM employees;
```

⁸Это расширение PostgreSQL, которое не имеет отношения к стандарту SQL

Здесь первое число – позиция элемента строки (нумерация начинается с единицы), а второе число – число элементов подстроки, которое нужно оставить в выводе.

Эквивалентная конструкция

```
SELECT name, job, substring(job from 1 for 4) FROM employees;
```

Очень гибкое решение дают *регулярные выражения* POSIX (Portable Operating System Interface – переносимый интерфейс операционных систем). Например для того чтобы выбрать, как и в рассмотренном выше примере, всех, кто имеет отношение к работе с данными, можно использовать такую конструкцию

```
SELECT * FROM employees WHERE job ~* 'data .*';
```

Здесь ~* – оператор соответствию шаблону *без* учета регистра. Если нужно учесть регистр, то используется оператор ~.

Аналогично !~ – оператор несоответствия шаблону с учетом регистра, оператор !~* – оператор несоответствия шаблону *без* учета регистра.

Регулярные выражения могут совпадать с строкой в любом месте (не обязательно со всей строкой).

Еще пример. Нужно выбрать строки из столбца job, в которых последовательность символов dev стоит в начале строки («^» – якорь начала строки)

```
select * from employees where job ~* '^dev.*';
```

Пример с положительной проверкой вперед

```
select * from employees where job ~* '(?=engineer)';
```

Выведет

id	name	salary	job	manager_id
6	Steven	2400	DevOps Engineer	7
20	Alex	25050	Data Engineer	10

У регулярных выражений есть один тонкий нюанс, связанный с «жадностью» квантификатора. Рассмотрим пример

```
SELECT substring('xy1234z', 'y*(\d{1,3})'); -- вернет '123'
```

Здесь используется «жадный» квантификатор *. В общем случае этот квантификатор соответствует элементу, который не повторяется ни разу или повторяется произвольное число раз, однако в данном случае он соответствует строго единственному символу 'y' (больше в строке символов 'y' нет). Другими словами подшаблон y* жадно забирает символ 'y' и на этом успокаивается, так как больше ничего от этой последовательности взять не сможет. А затем подшаблон \d{1,3} жадно забирает 3 цифры, так как ему никто не мешает это сделать.

Но если использовать «нежадный» вариант квантификатора *?, то картина изменится

```
SELECT substring('xy1234z', 'y*(\d{1,3})?'); -- вернет '1'
```

Вариант, когда подстрока не имеет ни одного элемента устраивает подшаблон y*? (потому что используется нежадный квантификатор), но не устраивает подшаблон \d{1,3}, согласно которому в последовательности должна быть хотя бы одна цифра. Приходится как бы «тянуть» подшаблон y*? вправо, потому что нежадный квантификатор стремится схлопнуться в пустое начало строки. Таким образом, на 2-ой итерации курсор сдвигается вправо на один символ и теперь

указывате на 'х'. Нежадный подшаблон `у*?` удовлетворен (ему достаточно и пустой подстроки), но не удовлетворен второй подшаблон. Курсор передвигается еще на один элемент вправо. И теперь указывает на 'у' (подстрока: 'ху'). Нежадный подшаблон `у*?` снова удовлетворен, но подшаблон `\d{1,3}` все еще не содержит ни одной цифры, поэтому курсор снова перемещается вправо еще на один элемент. В этот раз подстрока выглядит как 'ху1', что удовлетворяет и первый, и второй подшаблоны, но подшаблон `у*?` больше бы устроил вариант, когда подстрока пустая. С другой стороны подшаблон `\d{1,3}` требует, чтобы в подстроке была хотя бы одна цифра, поэтому подстрока вида 'ху1' (в группу попадает только 1) – это компромисс.

Замечание

Удобно представлять, что *нежадные квантификаторы* типа `*?` представляют собой пружинки сжатия, которые тянут влево и в самом простом случае довольствуются пустой последовательностью, а *жадные квантификаторы* (`*`) можно представлять как пружинки растяжения, которые стремятся занять всю последовательность и с неохотой уступают элементы

37. Программирование на PL/pgSQL

37.1. Примеры работы с функциями

Требуется построить текстовое представление ранга объявления

```
CREATE OR REPLACE FUNCTION cast_rank_to_text (rank INT) RETURNS TEXT AS
$$
DECLARE -- объявление переменных
    rank_result TEXT;
BEGIN -- область видимости
    IF rank = 5 THEN rank_result = 'Excellent';
    ELSIF rank = 4 THEN rank_result = 'Very Good';
    ELSIF rank = 3 THEN rank_result = 'Fair';
    ELSIF rank = 2 THEN rank_result = 'Poor';
    ELSE rank_result = 'No such rank';
    END IF;
    RETURN rank_result;
END;
$$ LANGUAGE plpgsql;

SELECT n, cast_rank_to_text(n) FROM generate_series(1,6) AS t(n);
```

К слову, если бы этот код находился в отдельном файле, например, `sql_query.sql`, то его можно было запустить и посмотреть результат в терминале так

```
psql -U postgres -d postgres -c '\! chcp 1251' -f sql_query.sql
```

ВЫВЕДЕТ

```
+---+-----+
| n | cast_rank_to_text |
+---+-----+
| 1 | No such rank      |
| 2 | Poor              |
| 3 | Fair              |
| 4 | Very Good         |
| 5 | Excellent         |
| 6 | No such rank      |
+---+-----+
```


Рассмотренную выше задачу можно решить и с помощью конструкции CASE

```
CREATE OR REPLACE FUNCTION cast_rank_to_text (rank INT) RETURNS TEXT AS
$$
DECLARE
    rank_result TEXT;
BEGIN
    CASE rank
        WHEN 5 THEN rank_result = 'Excellent';
        WHEN 4 THEN rank_result = 'Very Good';
        WHEN 3 THEN rank_result = 'Good';
        WHEN 2 THEN rank_result = 'Fair';
        WHEN 1 THEN rank_result = 'Poor';
        ELSE rank_result = 'No such rank';
    END CASE;
    RETURN rank_result;
END; $$ LANGUAGE plpgsql;
```

Важно отметить, что в этой форме CASE запрещено сопоставление со значением NULL, потому что результатом сравнения с NULL является NULL. Чтобы обойти это ограничение, можно воспользоваться второй формой и задать условие сопоставления явно

```
...
WHEN rank IS NULL THEN RAISE EXCEPTION 'Rank should be not NULL';
...
```

37.2. Итерирование

Итерирование позволяет повторять блок команд. При этом часто требуется определить начальную точку и условие окончания. В PostgreSQL есть несколько команд для обхода результатов запроса и организации циклов: LOOP, CONTINUE, EXIT, FOR, WHILE и FOR EACH.

Самая простая команда выглядит так

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

Для демонстрации перепишем функцию factorial_psql

```
DROP FUNCTION IF EXISTS factorial_psql (fact INT);
CREATE OR REPLACE FUNCTION factorial_psql (fact INT) RETURNS BIGINT AS
$$
DECLARE -- область объявления переменных
    result BIGINT = 1;
BEGIN -- область видимости функции
    IF fact = 1 THEN RETURN 1;
    ELIF fact IS NULL OR fact < 1 THEN RAISE EXCEPTION 'Provide a positive integer';
    ELSE
        LOOP
            result = result*fact;
            fact = fact - 1;
            EXIT WHEN fact = 1;
        END LOOP;
    END if;
    RETURN result;
END; $$ LANGUAGE plpgsql;

SELECT n, 'fact('||n||')' || ' -> ' || factorial_psql(n) AS result
```

```
FROM (VALUES (3),(5),(8)) AS t(n); -- источником данных выступает виртуальная таблица
```

Выведет

```
+---+-----+
| n |      result      |
+---+-----+
| 3 | fact(3) -> 6      |
| 5 | fact(5) -> 120     |
| 8 | fact(8) -> 40320   |
+---+-----+
```

Здесь условная команда EXIT предотвращает заикливание. После выполнения EXIT управление передается команде, следующей за LOOP. Для управления выполнением команд внутри цикла LOOP PL/pgSQL предоставляет также команду CONTINUE, которая похожа на EXIT, только вместо выхода из цикла передает управление на его начало, пропуская код, оставшийся до конца цикла.

Замечание

Рекомендуется избегать команд CONTINUE и EXIT, особенно в середине блока, потому что они нарушают порядок выполнения, что затрудняет чтение кода

Команда WHILE продолжает выполнять блок команд, пока истинно заданное условие. Вот ее синтаксис

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

Ниже мы печатаем дни текущего месяца в цикле WHILE

```
DO $$
DECLARE -- область объявления переменных
    first_day_in_month date := date_trunc('month', current_date)::date;
    last_day_in_month date := (
        date_trunc('month', current_date) + '1 month - 1 day'::interval
    )::date;
    counter date := first_day_in_month;
BEGIN -- область видимости
    WHILE (counter <= last_day_in_month) LOOP
        RAISE NOTICE '%', counter;
        counter := counter + '1 day'::interval;
    END LOOP;
END; $$ LANGUAGE plpgsql;
```

В PL/pgSQL есть две формы команды FOR, предназначенные для:

- обход строк, возвращенных SQL-запросом;
- обхода диапазона целых чисел.

Синтаксис команды FOR представлен ниже:

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression1 .. expression2 [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

Здесь **name** – имя локальной переменной типа **integer**. Областью видимости этой переменной является цикл FOR. Команды внутри цикла могут ее читать, но не могут изменять. Но это

поведение можно изменить, определив переменную в секции объявлений объемлющего блока. Выражения `expression1` и `expression2` должны иметь целые значения. Если `expression1` равно `expression2`, то цикл `FOR` выполняется только один раз.

Ключевое слово `REVERSE` необязательно; оно определяет порядок, в котором генерируется диапазон (возрастающий или убывающий). Если слово `REVERSE` опущено, то `expression1` должно быть меньше `expression2`, иначе цикл не выполнится ни разу. Наконец, ключевое слово `BY` задает шаг между двумя последовательными числами; следующее за ним выражение `expression` должно быть положительным целым числом. В показанном ниже цикле `FOR` производится обход диапазона отрицательных чисел в обратном порядке и печатаются значения -1, -3, ..., -9

```
DO $$ -- анонимная функция
BEGIN
    FOR j REVERSE -1 .. -10 BY 2 LOOP
        RAISE NOTICE '%', j;
    END LOOP;
END; $$ LANGUAGE plpgsql;
```

Синтаксис цикла для обхода результатов запроса несколько иной

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

Локальная переменная `target` объявлена в объемлющем блоке. Она необязательно должна быть примитивного типа, скажем, `INTEGER` или `TEXT`, допускается также составной тип и тип `RECORD`. В PL/pgSQL обходить можно результат выполнения курсора или команды `SELECT`. Курсор – это специальный объект, инкапсулирующий запрос `SELECT` и позволяющий читать результаты по несколько строк за раз. В следующем примере печатаются имена всех баз данных

```
DO $$
DECLARE
    database RECORD;
BEGIN
    FOR database IN SELECT * FROM pg_database LOOP
        RAISE NOTICE '%', database.datname;
    END LOOP;
END; $$ LANGUAGE plpgsql;
```

Здесь конструкция `SELECT * FROM pg_database` возвращает набор строк, содержащих информацию о базах данных PostgreSQL. Каждая строка, возвращенная запросом, связывается с переменной цикла `database`, а затем выбирается значение атрибута `datname`.

37.3. Возврат из функции

Команда возврата завершает функцию и передает управление вызывающей стороне. Существует несколько разновидностей команды возврата: `RETURN`, `RETURN NEXT`, `RETURN QUERY`, `RETURN QUERY EXECUTE` и т.д. Команда `RETURN` может возвращать *одно значение* или *множество*.

```
DO $$
BEGIN
    RETURN; -- по достижении этого места функция завершает работу
    RAISE NOTICE 'This statement will not be executed';
END; $$ LANGUAGE plpgsql;
```

Тип `VOID` используется, когда смысл функции заключается в побочных эффектах, например записи в журнал. Встроенная функция `pg_sleep` задерживает выполнение серверного процесса на заданное число секунд.

Тип возвращаемого значения может быть базовым, составным, доменным или псевдотипом. Показанная ниже функция возвращает представление учетной записи в формате JSON

```
-- sql
CREATE OR REPLACE FUNCTION car_portal_app.get_account_in_json (account_id INT)
RETURNS JSON AS $$
    SELECT row_to_json(account)
    FROM car_portal_app.account
    WHERE account_id = $1;
$$ LANGUAGE sql;

-- plpgsql
CREATE OR REPLACE FUNCTION car_portal_app.get_account_in_json (acc_id INT)
RETURNS JSON AS
$$
BEGIN
    -- если возвращается одна строка, то просто RETURN!!!
    RETURN (SELECT row_to_json(account)
            FROM car_portal_app.account
            WHERE account_id = acc_id);
END; $$ LANGUAGE plpgsql;
```

Для возврата нескольких строк используются функции, возвращающие множества. Строка может иметь базовый тип (например, `integer`), составной, табличный, доменный или псевдотип. Функция, *возвращающая множество*, помечается ключевым словом `SETOF`

```
-- sql
CREATE OR REPLACE FUNCTION car_portal_app.car_model(model_name TEXT)
RETURNS SETOF car_portal_app.car_model AS
$$
SELECT car_model_id, make, model
FROM car_portal_app.car_model
WHERE model = model_name;
$$ LANGUAGE sql;

-- plpgsql
CREATE OR REPLACE FUNCTION car_portal_app.car_model(model_name TEXT)
RETURNS SETOF car_portal_app.car_model AS $$
BEGIN
    RETURN QUERY SELECT car_model_id, make, model FROM car_portal_app.car_model
    WHERE model = model_name;
END; $$ LANGUAGE plpgsql;
```

Если для возвращаемого значения еще нет типа, то можно [1, стр. 214]:

- определить и использовать новый тип данных,
- использовать тип `TABLE`,
- использовать выходные параметры и тип данных `RECORD`.

Пусть требуется вернуть только поля `car_model_id` и `make`, и соответствующий тип данных не определен. Тогда предыдущую функцию можно переписать в виде

```
CREATE OR REPLACE car_portal_app.car_model(model_name TEXT)
RETURNS TABLE (car_model_id INT, make TEXT) AS $$
BEGIN
    -- если возвращается множество строк, то RETURN QUERY!!!
```

```

RETURN QUERY SELECT car_model_id, make FROM car_portal_app.car_model
WHERE model = model_name;
END; $$ LANGUAGE plpgsql;

```

37.4. Обработка исключений

Для перехвата и возбуждения исключений в PostgreSQL предназначены команды `EXCEPTION` и `RAISE`. Ошибки возникают при нарушении ограничений целостности данных или при выполнении недопустимых операций, например: присваивание текста целому числу, деление на нуль, присваивание переменной значения вне допустимого диапазона и т.д. По умолчанию любая ошибка в PL/pgSQL-функции приводит к прерыванию выполнения и откату изменений. Чтобы восстановиться, можно перехватить ошибку с помощью фразы `EXCEPTION`, синтаксически очень похожей на блоки в PL/pgSQL. Кроме того, ошибки можно возбуждать в команде `RAISE`. Пример

```

CREATE OR REPLACE FUNCTION check_not_null (value anyelement) RETURNS VOID AS
$$
BEGIN
    IF (value IS NULL) THEN
        RAISE EXCEPTION USING ERRCODE = 'check_violation';
    END IF;
END; $$ LANGUAGE plpgsql;

```

Полиморфная функция `check_not_null` просто возбуждает ошибку, в которой состояние `SQLSTATE` содержит строку `check_violation`. Если вызвать эту функцию, передав ей `NULL` в качестве аргумента, то возникнет ошибка.

Чтобы можно было точно узнать, когда и почему возникло исключение, в PostgreSQL определено несколько категорий кодов ошибок. Например, если пользователь возбудил исключение, не указав `ERRCODE`, то `SQLSTATE` будет содержать строку `P001`, а если нарушено ограничение уникальности, то строку `23505`.

Для определения причины ошибки в команде `EXCEPTION` можно сравнивать `SQLSTATE` или название условия

```

WHEN unique_violation THEN ...
WHEN SQLSTATE '23505' THEN ...

```

Наконец, при возбуждении ошибки можно задать сообщение об ошибке и `SQLSTATE`, памятуя о том, что код ошибки `ERRCODE` должен содержать 5 цифр и / или букв в кодировке ASCII, но не должен быть равен 00000, например

```

DO $$
BEGIN
    RAISE EXCEPTION USING ERRCODE = '1234X', MESSAGE = 'test customized SQLSTATE: ';
    EXCEPTION WHEN SQLSTATE '1234X' THEN RAISE NOTICE '% %', SQLERRM, SQLSTATE;
END; $$ LANGUAGE plpgsql;

```

Для демонстрации перехвата исключений перепишем функцию `factorial` так, чтобы она возвращала `NULL`, если передан аргумент `NULL`

```

DROP FUNCTION IF EXISTS factorial (INT);
CREATE OR REPLACE FUNCTION factorial(fact INT) RETURNS BIGINT AS
$$
BEGIN
    PERFORM check_not_null(fact);
    IF fact > 1 THEN RETURN factorial(fact - 1)*fact;

```

```

ELSIF fact IN (0,1) THEN RETURN 1;
ELSE RETURN NULL;
END IF;

EXCEPTION
    WHEN check_violation THEN RETURN NULL;
    WHEN OTHERS THEN RAISE NOTICE '% %', SQLERRM, SQLSTATE;
END; LANGUAGE plpgsql;

```

Если теперь вызвать

```
SELECT * FROM factorial(NULL::INT); -- вернет NULL
```

Функция `factorial` не возбудила ошибку, поскольку она была перехвачена в команде `EXCEPTION` и вместо нее возвращено `NULL`. Обратите внимание, что сравнивается не `SQLSTATE`, а имя условия. Специальное имя условия `OTHERS` соответствует любой ошибке, оно часто используется, чтобы перехватить неожиданные ошибки.

37.5. Динамический SQL

Динамический SQL служит для построения и выполнения запросов «на лету». В отличие от статической команды SQL, полный текст динамической команды заранее неизвестен и может меняться от выполнения к выполнению. Допускаются команды подязыков DDL, DCL и DML. Динамический SQL позволяет сократить объем однотипных задач.

Еще одно очень важное применение динамического SQL – устранение побочных эффектов кеширования в PL/pgSQL, поскольку запросы, выполняемые с помощью команды `EXECUTE`, *не кешируются*.

Для реализации динамического SQL предназначена команда `EXECUTE`. Она принимает и интерпретирует строку. Синтаксис этой команды показан ниже

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ...] ];
```

Динамическое выполнение команд DDL Иногда бывает необходимо выполнить операции на уровне объектов базы данных: таблиц, индексов, столбцов, ролей и т.д. Например, после развертывания таблицы с целью обновления статистики часто производят ее очистку и анализ. Так для анализа таблиц из схемы `car_portal_app` можно было бы написать такой скрипт

```

-- VACUUM не может использоваться в функциях
DO $$ -- анонимная функция
DECLARE -- область определения переменных
    table_name text;
    schema_name text := 'bookings';
BEGIN -- область видимости
    FOR table_name IN (SELECT tablename FROM pg_tables
                       WHERE schemaname = schema_name) LOOP
        RAISE NOTICE 'Analyzing %', table_name;
        EXECUTE 'ANALYZE ' || schema_name || '.' || table_name;
        -- или так: EXECUTE format('ANALYZE %I.%I', schema_name, table_name);
    END LOOP;
END; $$ LANGUAGE plpgsql;

```

Динамическое выполнение команд DML Некоторые приложения работают с данными интерактивно. Например, ежемесячно могут генерировать данные для выставления счетов. Бывает и так, что приложение фильтрует данные на основе критерия, заданного пользователем. Для таких задач очень удобен динамический SQL. Пример

```
CREATE OR REPLACE FUNCTION get_lines(predicate TEXT)
  RETURNS TABLE(name VARCHAR, job VARCHAR) AS
$$
DECLARE
  table_name TEXT := 'employees';
BEGIN
  RETURN QUERY EXECUTE 'SELECT name, job FROM table_name WHERE ' || predicate;
END; $$ LANGUAGE plpgsql;
```

Протестируем

```
SELECT * FROM get_lines('id IN (1, 10, 15)');
```

Динамический SQL и кеширование Как уже было сказано, PL/pgSQL *кеширует планы выполнения*. Это хорошо, если сгенерированный план статический. Например, для следующей команды ожидается, что всегда будет использоваться просмотр индекса благодаря его избирательности. В таком случае кеширование плана экономит время и повышает производительность.

Но бывает и по-другому. Предположим, что построен индекс по столбцу `advertisement_date`, и мы хотим получить количество объявлений, начиная с некоторой даты. Для чтения записей из таблицы `advertisement_date` можно применить как просмотр индекса, так и последовательный просмотр. Все зависит от избирательности индекса. Кеширование плана выполнения такого запроса может создать серьезные проблемы.

Для решения проблемы можно переписать функцию, указав в качестве языка SQL, или оставить PL/pgSQL, но воспользоваться командой `EXECUTE` (тогда план выполнения запроса не будет кешироваться)

```
-- с помощью $1, $2 и пр. можно передавать ТОЛЬКО значения!!!
-- для динамической подстановки имен таблиц, столбцов и пр. служит функция format()
CREATE OR REPLACE FUNCTION car_portal_app.get_advertisement_count(some_date timestampz)
  RETURNS BIGINT AS
$$
DECLARE
  count BIGINT;
BEGIN
  -- запросы EXECUTE не кешируются!!!
  EXECUTE 'SELECT count(*) FROM car_portal_app.advertisement
    WHERE advertisement_date >= $1'
  USING some_date -- символ $1 ссылается на значения в предложении USING
  INTO count; -- число строк, возвращенное запросом, связывается с переменной count
  RETURN count;
END; $$ LANGUAGE plpgsql;
```

Пример работы с json-объектом с помощью `RETURN QUERY EXECUTE`, функцией `format` и языком путей SQL/JSON

Листинг 1. Функция на PL/pgSQL, возвращающая подвыборку по условию

```
CREATE OR REPLACE FUNCTION pivot(lang_expr TEXT)
  RETURNS TABLE(package TEXT, language TEXT) AS
$$
```

```

BEGIN
RETURN QUERY EXECUTE format(
    E'SELECT obj ->> \'package name\' AS "package name",
        obj ->> \'language\' AS language
    FROM test_json
    WHERE obj @? \'$.language[*] ? (@ like_regex "%I" flag "i")\'', $1); -- $1 указывает на
    lang_expr
END; $$ LANGUAGE plpgsql;
-- использование
SELECT * FROM pivot('python'::text);

```

ВЫВЕДЕТ

```

+-----+-----+
| package | language |
+-----+-----+
| Nastran  | ["C++", "Python"] |
| LMS Virtual Lab | ["Scala", "Python"] |
+-----+-----+

```

Нужно указывать `SELECT * FROM pivot('python'::text)`. Если указать `SELECT pivot('python'::text)`, то выведет

```

+-----+
| pivot |
+-----+
| (Nastran,"["C++","Python"])" |
| ("LMS Virtual Lab","["Scala","Python"])" |
+-----+
(2 строки)

```

Важные моменты: если в строке необходимо использовать несколько одинарных кавычек, то должна использоваться следующая конструкция `E'... \'...\''`; для подстановки в строку лучше использовать функцию `format`.

38. Массивы

Массив состоит из набора элементов (значений или переменных), порядок массива важен, каждый элемент идентифицируется своим индексом. Индексы элементов начинаются с 1. Все элементы массива должны иметь один и тот же тип, например `INT`, `TEXT` и т.п.

Поддерживаются также многомерные массивы. В массивах PostgreSQL допускаются дубликаты и значения `NULL`.

Инициализируем одномерный массив и получим первый элемент

```

SELECT ('{red,blue,green}'::text[])[1]; -- red

```

По умолчанию длина массива не задается, но ее можно указать, когда массивы используются для определения отношения. По умолчанию индексирование массива начинается с единицы, но и это поведение можно изменить, определив размерность на этапе инициализации

```

SELECT ('[0:2]={red,blue,green}'::text[])[1] AS color; -- blue

```

Пример

```

-- сначала создаем массив, а затем можно обращаться к его элементам
WITH test AS ( -- общее табличное выражение
    SELECT '[0:1]={10,20}'::int[] AS arr

```



```
)
SELECT arr, arr[0] FROM test;
```

Выведет

```
+-----+-----+
|      arr      | arr |
+-----+-----+
| [0:1]={10,20} | 10  |
+-----+-----+
```

Для инициализации массива используется конструкция {}. Но есть и другой способ

```
SELECT array['red', 'green', 'blue'] AS primary_colors;
```

PostgreSQL предоставляет много функций для манипулирования массивами, например `array_remove` для удаления элемента. Ниже продемонстрированы некоторые функции массивов:

```
SELECT
  array_ndims(two_dim_array) AS "Number of dimensions",
  array_dims(two_dim_array) AS "Dimensions index range",
  array_length(two_dim_array, 1) AS "The array length of 1st dimension",
  cardinality(two_dim_array) AS "Number of elements",
  two_dim_array[1][1] AS "The first element"
FROM
  (VALUES ('{ {red,green,blue}, {red,green,blue} }'::text[] [])) AS foo(two_dim_array);
```

выведет (предварительно нужно задать в psql \x)

```
+-[ RECORD 1 ]-----+-----+
| Number of dimensions      | 2      |
| Dimensions index range    | [1:2][1:3] |
| The array length of 1st dimension | 2      |
| Number of elements        | 6      |
| The first element         | red     |
+-----+-----+
```

Распространенное применение массивов – моделирование многозначных атрибутов. Также они используются для моделирования хранилищ ключей и значений. Для этого заводят два массива: один содержит ключи, другой – значения, а для связывания ключа со значением используется индекс массива.

Так, в таблице `pg_stats` этот подход применим для хранения информации о гистограмме типичных значений. В столбцах `most_common_vals` и `most_common_freqs` хранятся, соответственно, типичные значения в некотором столбце и частоты этих значений, как показано в примере ниже

```
SELECT tablename, attname, most_common_vals, most_common_freqs
FROM pg_stats
WHERE array_length(most_common_vals, 1) < 10 AND schemaname NOT IN
      ('pg_catalog', 'information_schema') LIMIT 1;
```

Для того чтобы передать функции произвольное число аргументов, следует использовать ключевое слово `VARIADIC`. Например

```
CREATE OR REPLACE null_count (VARIADIC arr int[]) RETURNS INT AS
$$
  SELECT count(CASE WHEN m IS NOT NULL THEN 1 ELSE NULL END)::int
  FROM unnest($1) AS m(n)
$$ LANGUAGE sql;
```

Пример использования

```
SELECT null_count(NULL, 5, NULL, 10); -- 2
SELECT * FROM null_count(VARIADIC '{NULL,1}':int[]); -- 1
```

Еще один пример работы с массивами

```
SELECT substring('python',1,i) FROM generate_series(1,length('python')) AS g(i);
```

ВЫВЕДЕТ

```
+-----+
| substring |
+-----+
| p         |
| py        |
| pyt       |
| pyth      |
| pytho     |
| python    |
+-----+
```

Теперь можно обернуть этот вывод массивом

```
SELECT
array(SELECT substring('python',1,i)
      FROM generate_series(1,length('python')) AS g(i)
)::text[];
```

ВЫВЕДЕТ

```
+-----+
|          array          |
+-----+
| {p,py,pyt,pyth,pytho,python} |
+-----+
```

38.1. Функции и операторы массивов

Для массивов, как и для других типов данных, определены операторы. Например, оператор = сравнивает на равенство, а оператор || производит конкатенацию. Операторы @> и <@ возвращают истину, если левый массив содержит в правом или содержится соответственно

```
SELECT '{2,5,10}':int[] @> '{2}':int[]; --t
SELECT '{42}':int[] <@ '{2,42,100}':int[]; --t
```

Элементы, попавшие в одну группу, можно представить в виде массива с помощью функции array_agg.

Функция ANY аналогична конструкции SQL IN() и служит для проверки принадлежности

```
SELECT 1 IN (1,2,3), 1 = ANY ('{1,2,3}':int[]);
```

38.2. Доступ к элементам массива и их модификация

К элементу массива обращаются по индексу; если элемент с указанным индексом не существует, то возвращается значение NULL:

```
CREATE TABLE colors(
    color text[]
);

INSERT INTO colors(color) VALUES ('{red,green}'::text[]);
INSERT INTO colors(color) VALUES ('{red}'::text[]);
```

Что теперь храниться в таблице?

```
-- в данном случае нумерация начинается с 1
SELECT color[2] FROM colors;
```

Выведет

```
+-----+
| color |
+-----+
| green |
| <NULL> |
+-----+
```

Чтобы получить булеву маску для столбца color выполним такой запрос

```
SELECT
    color[2] IS NOT DISTINCT FROM ('{green}'::text[])[1]
    -- или просто
    -- color[2] IS NOT DISTINCT FROM 'green'
FROM colors;
```

вернет

```
+-----+
| ?column? |
+-----+
| t        |
| f        |
+-----+
```

Можно получить срез массива, указав верхнюю и нижнюю границы

```
-- если индексов нет в массиве, то вернется пустой массив {}
SELECT color[1:2] FROM colors;
```

Обновить можно массив целиком, срез массива или отдельный элемент. Можно также дописать элементы в конец массива, воспользовавшись оператором конкатенации ||

```
SELECT array['red', 'green'] || '{blue}'::text[] AS append;
```

Для удаления всех элементов с указанным значением предназначена функция array_remove

```
SELECT array_remove('{Ansys,Nastran,Abaqus}'::text[], 'Ansys');
```

Выведет

```
+-----+
| array_remove |
+-----+
| {Nastran,Abaqus} |
+-----+
```

Чтобы удалить элемент по индексу, можно воспользоваться фразой WITH ORDINALITY

```
SELECT array(
  SELECT unnest
  FROM unnest( '{Ansys,Nastran,Abaqus}'::text[] )
  WITH ordinality
  WHERE ordinality <> 1
);
```

Здесь функция `unnest` преобразует массив в последовательность строк. Конструкция `WITH ORDINALITY` создает еще один столбец с именем `ordinality`, который по сути просто задает номера строк. В предложении `WHERE` выбираются те строки, у которых номера не равны 1. В список выборки включается только столбец `unnest`, который создается функцией `unnest`.

38.3. Индексирование массивов

Для индексирования массивов используется GIN-индекс. В стандартном дистрибутиве PostgreSQL имеется класс операторов GIN для одномерных массивов. GIN-индекс поддерживают следующие операторы:

- оператор содержит `@>`,
- оператор содержится в `<@`,
- пересекается `&&`,
- оператор сравнения на равенство `=`.

GIN-индекс по столбцу `color` создается следующим образом

```
CREATE INDEX ON colors USING GIN(color);
```

Чтобы заставить просматривать индекс, следует

```
postgres=> SET enable_seqscan TO off;
postgres=> EXPLAIN SELECT * FROM colors WHERE '{red}'::text[] && color;
```

39. Хранилище ключей и значений

Хранилище ключей и значений, или ассоциативный массив, – очень популярная структура данных в современных языках программирования. Имеются также базы данных, специально заточенные под эту структуру данных, например Redis.

PostgreSQL поддерживает хранилище ключей и значений – `hstore`. Расширение `hstore` дает в руки разработчику лучшее из обоих миров, оно предлагает дополнительную гибкость, не жертвуя функциональностью PostgreSQL.

Кроме того, `hstore` позволяет моделировать *слабоструктурированные данные и разреженные массивы*, оставаясь в рамках реляционной модели.

Для создания расширения `hstore` нужно выполнить следующую команду от имени супер-пользователя

```
CREATE EXTENSION hstore;
```

Текстовое представление `hstore` содержит нуль или более пар `key=>value`, разделенных запятой, например

```
postgres=# SELECT 'tires=>"winter tires", seat=>leather'::hstore;
```

выведет

```
+-----+
|               hstore               |
+-----+
| "seat"=>"leather", "tires"=>"winter tires" |
+-----+
```

Можно также сгенерировать одиночное значение в `hstore` с помощью функции `hstore(key, value)`

```
SELECT hstore('CAE-package', 'Ansys');
```

ВЫВЕДЕТ

```
+-----+
|               hstore               |
+-----+
| "CAE-package"=>"Ansys" |
+-----+
```

Все ключи в хранилище `hstore` должны быть уникальны. Однако у `hstore` есть ограничение:

- о это не настоящее хранилище документов, поэтому представить в нем вложенные объекты трудно,
- о управление множеством ключей, поскольку ключи в `hstore` чувствительны к регистру.

Чтобы получить значение ключа, используйте оператор `->`. Для добавления в `hstore` служит оператор конкатенации `||`, а для удаления пары ключ-значение – оператор `-`. Для обновления хранилища `hstore` следует конкатенировать его с другим экземпляром `hstore`, который содержит новое значение.

```
CREATE TABLE features(
    feature hstore
);
```

```
-- вставить в виртуальную таблицу пару <<ключ : значение>>
-- и вернуть эту пару с помощью RETURNING
postgres=# INSERT INTO features(feature) VALUES ('Engine=>Diesel')::hstore RETURNING *;
```

```
-- добавить новую пару <<ключ : значение>>
UPDATE features SET feature = feature || 'Seat=>Lether')::hstore RETURNING *;
```

```
-- обновление значения для ключа, существующего в хеш-таблице, аналогично добавлению пары
-- если ключ есть в хеш-таблице, то его значение обновляется
UPDATE features SET feature = feature || 'Engine=>Petrol')::hstore RETURNING *;
```

```
-- удаление ключа
UPDATE features SET feature = feature - 'Seat')::text RETURNING *;
```

Тип данных `hstore` преобразуется во множество с помощью функции `each(hstore)`, и это открывает перед разработчиком возможность применять к `hstore` все операции реляционной алгебры, например, `DISTINCT`, `GROUP BY` и `ORDER BY`.

```
-- получить уникальные ключи
SELECT DISTINCT (each(feature)).key FROM features;
```

```
-- получить пары
SELECT DISTINCT (each(feature)).* FROM features;
```

ВЫВЕДЕТ

```
+-----+-----+
|   key   |   value   |
+-----+-----+
| CAE-package | Ansys      |
| Language   | IronPython |
| Solver type | Direct     |
+-----+-----+
```

39.1. Индексирование hstore

Для индексирования данных типа **hstore** можно воспользоваться индексами GIN и GIST, а какой именно выбрать зависит от ряда факторов: количества строк в таблице, наличного места, требуемой производительности поиска и обновления по индексу, характера запросов и т.д.

Прежде чем остановится на том или ином индексе, нужно провести тщательное эталонное тестирование.

Сначала создадим GIN-индекс для выборки записи по ключу

```
CREATE INDEX ON features USING GIN(feature);
```

Проверить содержит ли хранилище заданный ключ можно с помощью оператора **?**, вывести значение по ключу с помощью оператора **->** (поддерживаются только одинарные кавычки)

```
-- выводит значения по заданному ключу для тех строк, в которых хранится хеш-таблица, в которой
    содержится ключ 'Language'
SELECT feature->'Language'
FROM features
WHERE feature ? 'Language';
```

```
-- выводит значения по заданному ключу для тех строк, в которых есть хеш-таблица с ключом '
    Solver' и значением, которое начинается на 'dir'
SELECT feature->'Solver'
FROM features
WHERE feature->'Solver' ~* 'dir.*';
```

Еще пример на использование оператора **->**

```
CREATE TABLE hstore_doc(
    id serial primary key,
    doc hstore
);
INSERT INTO hstore_doc(doc) VALUES ('"solver type"=>direct,"package name"=>Ansys,language=>
    IronPython');
INSERT INTO hstore_doc(doc) VALUES ('"solver type"=>iterative,"package name"=>Nastran,language=>
    C++'::hstore);
INSERT INTO hstore_doc(doc) VALUES ('language=>Java,"package name"=>Abaqus')
SELECT -- при обращении по ключу поддерживаются только одинарные кавычки
    doc->'solver type' AS "solver type",
    doc->'package name' AS "package name"
FROM hstore_doc;
```

ВЫВЕДЕТ

```
+-----+-----+
| solver type | package name |
+-----+-----+
| direct      | Ansys        |
+-----+-----+
```

iterative	Nastran	
<NULL>	Abaqus	
+-----+	+-----+	

Пример на использование оператора ?

```
SELECT
doc->'solver type' AS "solver type",
doc->'package name' AS "package name"
FROM hstore_doc
WHERE doc ? 'solver type'; -- те строки, в которых есть хранилище с ключом 'solver type'
```

Разумеется, если некоторый оператор не поддерживает GIN-индексом, как, например, оператор ->, можно использовать и индекс типа B-tree

```
CREATE INDEX ON features ((feature->'Engine'))
```

```
SELECT feature->'Engine'
FROM features
WHERE feature->'Engine' = 'Diesel'; -- значение, возвращенное по заданному ключу,
-- сравнивается с 'Diesel'
```

40. Структура данных JSON

JSON – универсальная структура данных, понятная как человеку, так и машине. Она поддерживается почти всеми современными языками программирования и повсеместно применяется в качестве формата обмена данными в REST-совместимых веб-службах.

40.1. JSON и XML

И XML, и JSON используются для определения структуры данных передаваемых документов. Грамматически JSON проще, чем XML, вследствие чего JSON-документы компактнее и проще для чтения и написания.

40.2. Типы данных JSON в PostgreSQL

PostgreSQL поддерживает два типа JSON: JSON и JSONB, тот и другой реализуют спецификацию RFC 7159. Оба типа применимы для контроля правил JSON. Они почти идентичны, но JSONB эффективнее, потому что является двоичным форматом и поддерживает индексы.

При работе с типом JSON рекомендуется задавать для базы данных кодировку UTF8, чтобы обеспечить его совместимость со стандартом RFC 7159. Документ, сохраняемый как объект типа JSON, хранится в текстовом формате. Если же объект храниться как JSONB, то примитивные типы JSON – string, boolean, number – отображаются, соответственно, на типы text, Boolean и numeric.

40.3. Доступ к объектам типа JSON и их модификация

Если текст приводится к типу json, то он сохраняется и отображаются без какой-либо обработки, поэтому сохраняются все пробелы, форматирование чисел и другие детали. В формате jsonb эти детали не сохраняются.

JSON-объекты могут содержать вложенные JSON-объекты, массивы, вложенные массивы, массивы JSON-объектов и т.д. Глубина вложенности произвольна, так что можно конструировать весьма сложные JSON-документы. В одном массиве JSON-документа могут находиться элементы разных типов

```
SELECT '{
  "name" : "John",
  "Address" : {"Street" : "Some street",
               "city" : "Some city"},
  "rank" : [5,3,4,5,2,3,4,5]
}'::jsonb;
```

Можно получить поле JSON-объекта в виде JSON-объекта или текста. К полям можно также обращаться по индексу или по имени поля:

- -> (для JSON), ->> (для текста): возвращает поле JSON-объекта по индексу или по имени поля,
- #> (для JSON), #>> (для текста): возвращает поле JSON-объекта по указанному пути.

Пример

```
CREATE TABLE json_doc( doc jsonb );
INSERT INTO json_doc SELECT
  '{"name" : "John",
   "Address" : {
     "Street" : "Some street",
     "city" : "Some city"
   },
   "rank" : [5,3,4,5,2,3,4,5]
}'::jsonb;
```

Если требуется вернуть города

```
SELECT doc->'Address'->>'city', doc#>>'{Address,city}' FROM json_doc;
```

Вставить в JSON-объект пару ключ-значение

```
UPDATE json_doc SET doc = jsonb_insert(doc, '{hobby}', '["swim", "read"]', true) RETURNING *;
```

А вот так изменяется существующая пара ключ-значение

```
UPDATE json_doc SET doc = jsonb_set(doc, '{hobby}', '["read"]', true) RETURNING *;
```

И наконец, удалим пару ключ-значение

```
UPDATE json_doc SET doc = doc - 'hobby' RETURNING *;
```

40.4. Индексирование JSON-документов

Для индексирования JSONB-документов используется GIN-индекс, который поддерживает следующие операторы:

- @>: содержит ли JSON-документ в левой части значение в правой части,
- ? : существует ли указанный ключ в JSON-документе,
- ?& : существует ли в JSON-документе хотя бы один из элементов текстового массива,
- ?| : существуют ли в JSON-документе все элементы текстового массива (ключи).

Содержит ли JSON-объект указанную пару «ключ-значение»


```
SELECT '{ "package name" : "Ansys", "solver type" : "direct" }'::jsonb @> '{ "package name" : "Ansys" }'::jsonb; -- t
```

JSON-документ можно преобразовать во множество с помощью функции `json_to_record()`. Это полезно, потому что позволяет сортировать данные, как в обычных хранилищах. Кроме того, можно агрегировать данные из нескольких строк и сконструировать массив JSON-объектов с помощью функции `json_agg()`.

41. PostgreSQL и Python

41.1. Простой пример работы с PostgreSQL через библиотеки SQLAlchemy и psycopg2

Модуль `psycopg2` умеет также выполнять команду `COPY`, чтобы быстро прочитать из базы или записать в базу большой объем данных. Методы `copy_to()` и `copy_from()` класса `cursor` соответственно копируют данные из таблицы в файлоподобный объект или из файлоподобного объекта в таблицу. Имеется также метод `cursor.copy_expert()`, представляющий значительно более развитую функциональность, – столь же гибкий, как сама команда `COPY`.

Метод `cursor.copy_expert()` принимает следующие параметры:

- `file`: файлоподобный объект, принимающий данные. Это может быть файл, или объект `StringIO`, или любой другой объект, поддерживающий метод `write()`,
- `table`: имя копируемой таблицы,
- `sep`: разделитель полей. По умолчанию знак табуляции,
- `null`: текстовое представление значений `NULL`,
- `columns`: список столбцов, подлежащих копированию.

Записи в объекте `file` будут разделены знаками новой строки. Ниже демонстрируется применение метода `copy_to()`:

```
import io
with io.StringIO() as s: # из таблицы в файлоподобный объект
    cur.copy_to(table='car_portal_app.car_model', file=s)
    print(s.getvalue())
```

В результате на консоль (т.е. в стандартный поток вывода) будет выведено содержимое таблицы `car_portal_app.car_model`, как это сделала бы команда `COPY car_portal_app.car_model TO STDOUT;`, выполненная в `psql`.

Метод `cursor.copy_from()` принимает такие же параметры и дополнительно параметр `size`, задающий размер буфера для чтения из объекта `file`. Объект `file` должен поддерживать методы `read()` и `readline()`. Вот пример того, как вставить записи в таблицу `car_portal_app.car_model` с помощью команды `COPY`

```
import io
with io.StringIO('Tesla\tModel-X\n') as s: # из файлоподобного объекта в таблицу
    cur.copy_from(table='car_portal_app.car_model',
                  file=s, columns=['make', 'model'])
```

Или так, в случае реального файла

```
import psycopg2
...
```

```

try:
    cur.copy_from( # из файла в таблицу
        file=open('ram_price.csv', 'r'), # файловый объект
        table='public.ram_test', # таблица
        sep=',',
        columns = ['id', 'date', 'price']
    )
except psycopg2.Error as err:
    print(err)

```

Пример работы с базой данных при помощи библиотеки `psycopg2`

```

import psycopg2

conn_info = {
    'user' : 'postgres',
    'dbname' : 'demo',
    'port' : 5432
}

# создаем соединение с базой данных
conn = psycopg2.connect(**conn_info)

# создаем объект-курсор
cur = conn.cursor()

# выполняем запрос
cur.execute('''
    SELECT count(*), count(DISTINCT fare_conditions)
    FROM ticket_flights
''')

# выводим результат работы SQL-запроса
cur.fetchall() # [(1045726, 3)]

# если сейчас попытаться извлечь лишь первую строку результата, то вывод будет пустым
cur.fetchone()

# чтобы все-таки вывести первую строку, требуется повторить выборку
cur.execute('SELECT count(*), count(DISTINCT fare_conditions) FROM ticket_flights')
cur.fetchone() # (1045726, 3)

```

Для разработки асинхронных приложений используется библиотека `asyncpg`. Она предоставляет асинхронный API, предназначенный для работы совместно с `asyncio` – библиотекой, используемой для написания конкурентного кода на Python.

Выше были описаны библиотеки низкого уровня. Чтобы их использовать, разработчик должен понимать, как работают базы данных, и знать язык SQL. С другой стороны, если база данных – всего лишь один из компонентов программного решения, обеспечивающий хранение данных, а вся логика сосредоточена в приложениях верхнего уровня, то разработчик должен заниматься бизнес-логикой, а не реализацией взаимодействия с базой данных на уровне отдельных запросов.

В приложениях верхнего уровня бизнес-объекты представлены классами и их экземплярами. Методы классов соответствуют операциям предметной области. Задачи сохранения состояния объекта в базе данных и загрузка оттуда к таким операциям не относятся.

В разработке ПО существует концепция *объектно-реляционного отображения* (object relational mapping – ORM). Так называемый программный слой, который представляет хранящиеся в ба-

зе записи в виде экземпляров классов. При создании такого экземпляра запись вставляется в таблицу, а в результате его модификации запись обновляется.

Для Python существует библиотека SQLAlchemy, которая реализует ORM и умеет работать с разными базами данных, в т.ч. с PostgreSQL.

Чтобы подключиться к базе данных, нужно прежде всего создать объект `engine`, для чего предназначена функция `create_engine()`

```
import sqlalchemy

engine = sqlalchemy.create_engine(
    "postgresql+pg8000://car_portal_app@localhost/car_portal", echo=True
)
```

Здесь мы воспользовались строкой соединения, имеющей формат `dialect[+driver]://user:password@host/dbname`.

Параметр `echo`, равный `True`, означает, что SQLAlchemy должна протоколировать все выполненные команды SQL для отладки. В примере выше для подключения к базе использован драйвер `pg8000`. Заметим, что в этот момент приложение еще не подключилось к базе. Оно создало пул соединений, и, когда библиотеке SQLAlchemy понадобится выполнить команду, она запросит соединение из этого пула. Если соединение еще не существует, оно будет установлено.

Пример работы с базой данных при помощи библиотеки `sqlalchemy`

```
import sqlalchemy
import pandas as pd

engine = sqlalchemy.create_engine('postgresql://postgres@localhost:5432/demo')

# создаем соединение
conn = engine.connect(); conn # <sqlalchemy.engine.base.Connection at 0xd732088>

# выполняем запрос
conn.execute('''
    SELECT count(*), count(DISTINCT fare_conditions)
    FROM ticket_flights
''').fetchall() # [(1045726, 3)]

# и тут же, без повторного выполнения запроса можно извлечь только первую строку результата
conn.execute('''
    SELECT count(*), count(DISTINCT fare_conditions)
    FROM ticket_flights
''').fetchone() # (1045726, 3)

conn.execute('''
    SELECT fare_conditions, max(amount)
    FROM ticket_flights
    GROUP BY 1
''').fetchall()
# [('Business', Decimal('203300.00')),
#  ('Comfort', Decimal('47600.00')),
#  ('Economy', Decimal('74500.00'))]

conn.execute('''
    SELECT fare_conditions, max(amount)
    FROM ticket_flights
    GROUP BY 1
''').fetchone()
# ('Business', Decimal('203300.00'))
```

```
pd.read_sql('''
    select fare_conditions, max(amount)
    from ticket_flights
    group by 1;''',
engine) # обязательно нужно передать ранее созданный движок или соединение
# fare_conditions      max
# 0      Business  203300.0
# 1      Comfort   47600.0
# 2      Economy   74500.0
```

К слову, еще с помощью библиотеки `sqlalchemy` можно работать, например, с *распределенным аналитическим хранилищем больших данных* [Apache Kylin](#)

```
import sqlalchemy
import pandas as pd
import matplotlib.pyplot as plt

kylin_engine = sqlalchemy.create_engine('kylin://<username>:<password>@1<IP>:<PORT>/<
    project_name>', connect_args={'timeout' : 60})
sql = '''
    SELECT movieid, count(DISTINCT userid) AS COUNT_USERS
    FROM userratings
    GROUP BY movieid
    ORDER BY count(DISTINCT userid) DESC
    LIMIT 15
'''

moviecount = pd.read_sql(sql, kylin_engine)

df = moviecount.sort_values(by='COUNT_USERS', ascending=False, na_position='first')
...
```

Пример на динамическое конструирование SQL-запросов

```
import sqlalchemy

engine = sqlalchemy.create_engine('postgresql://postgres@localhost:5432/demo')
conn = engine.connect()
# динамическое построение запроса
conn.execute("SELECT * FROM {} WHERE prog_lang ~* '{}'".format('caepackage_test', '~py.*'))

# но вставка значений в таблицу возможно только через параметры метода 'execute'
conn.execute('INSERT INTO test_cae(id, name) VALUES %(id)s, %(name)s',
              {'id' : 1, 'name' : 'Ansys'}) # параметры, заданные с помощью словаря
)
conn.execute('INSERT INTO test_cae(id, name) VALUES (%s, %s)',
              (2, 'Abaqus')) # параметры, заданные с помощью кортежа
```

В предыдущих примерах метод `cursor.execute()` блокировал программу на все время выполнения запроса. Если запрос сложный, то программа будет заблокирована надолго, что не всегда желательно. Модуль `psycopg2` предоставляет способ асинхронного выполнения запросов. Это означает, что пока запрос выполняется, программа может заниматься другими делами.


Модуль `psycopg2` предоставляет вспомогательную функцию, которая блокирует выполнение и ждет завершения асинхронной операции. Это функция `psycopg2.extras.wait_select(conn)`.

42. Приемы работы в pgAdmin 4

43. Приемы работы в psql

psql – это терминальный клиент для работы с PostgreSQL. Утилита psql предоставляет ряд метакоманд и различные возможности, подобные тем, что имеются у командных оболочек, для облегчения написания скриптов и автоматизации широкого спектра задач.

43.1. Конфигурационный файл

Поведением интерактивного терминала psql можно управлять с помощью конфигурационного файла ~/.psqlrc. На ОС Windows этот файл располагается по адресу  C: \Users \ADM \AppData \Roaming \postgresql и называется psqlrc.conf.

Конфигурационный файл утилиты psql может включать следующие настройки

psqlrc.conf

```
\set QUIET 1
\timing on
\pset border 2
\pset null <NULL>
\set ON_ERROR_STOP on
\setenv PSQL_EDITOR "C:\Program Files\Git\usr\bin\vim.exe"
\set VERBOSITY verbose
\set QUIET 0
```

43.2. Метакоманды psql

\с или \connect: устанавливает новое подключение к серверу PostgreSQL. Параметры подключения можно указывать как позиционно, так и передавая аргумент.

Например

```
=> \с "host=localhost port=5432 dbname=mydb connect_timeout=10 sslmode=disable"
=> \с postgresql://tom@localhost/mydb?application_name=myapp
```

\cd: сменяет текущий рабочий каталог на заданный. Без аргументов устанавливает домашний каталог пользователя.

\conninfo: выводит информацию о текущем подключении к базе данных.

\copy: производит копирование данных с участием клиента. При этом выполняются SQL-команда COPY, но вместо чтения или записи в файл на сервере, psql читает или записывает файл и пересылает данные между сервером и локальной файловой системой. Это означает, что для доступа к файлам используются привилегии локального пользователя, а не сервера, и не требуются привилегии суперпользователя SQL. Синтаксис команды похож на синтаксис SQL-команды COPY. Все параметры, кроме источника и получателя данных, соответствуют параметрам COPY. Альтернативный способ получить тот же результат, что и с \copy ... to – использовать SQL-команду COPY ... TO STDOUT и завершить ее командой \g name.

\d[S+]: для каждого отношения (таблицы, представления, материализованного представления, индекса, последовательности, внешней таблицы) или составного типа, соответствующих шаблону, показывает все столбцы, их типы, табличное пространство и любые специальные атрибуты, такие как NOT NULL или значения по умолчанию. Также показываются связанные индексы, ограничения, правила и триггеры.

`\da[S]`: выводит список агрегатных функций вместе с типом возвращаемого значения и типами данных, которыми они оперируют.

`\dD[S+]`: выводит список доменов.

`\dL[S+]`: выводит список процедурных языков.

`\dn[S+]`: выводит список схем (пространств имен).

`\encoding`: устанавливает кодировку набора символов на клиенте. Без аргументов команда показывает текущую кодировку.

`\!`: выполняет команду операционной системы.

ЗАМЕЧАНИЕ: если после перехода на новую версию PostgreSQL и изменения номера порта (скажем, на 5433), команда

```
psql -U postgres
```

возвращает

```
psql: error: could not connect to server: No such file or directory
Is the server running locally and accepting
connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

то можно попробовать вариант с явным указанием порта

```
psql -U postgres -h localhost -p 5433
```

К слову, информацию о текущих процессах, связанных с Postgres можно посмотреть так

```
ps aux | grep -inE postgres
```

Эта команда должна вернуть что-то вроде

```
14:leor.finkelberg  3426  0,1  0,0  4481224  3616  ??  Ss   3:39    0:00.11 /Applications/
Postgres.app/Contents/Versions/12/bin/postgres -D /Users/leor.finkelberg/Library/Application
Support/Postgres/var-12 -p 5433
```

Здесь и информация о data-директории (-D), и информация о порту (-p).

43.3. Примеры использования

Вывести список таблиц, присутствующих в базе данных `postgres`, предварительно задав кодовую страницу, соответствующую Windows-кодировке

```
$ psql -U postgtes -d postgres -c '\! chcp 1251' -c '\d'
```

Вывести первые 5 строк таблицы `aircrafts` из базы данных `demo`, организовав вывод строк в вертикальном формате (`\x`; эквивалентно ключу `psql -x`)

```
$ psql -U postgtes -d demo -c '\x' -c 'TABLE aircrafts LIMIT 5;'
```

Выполнить вычисления факториала для чисел 5, 8 и 12, результат представить в виде таблицы из пар «число - факториал_от_числа», а затем результат выполнения запроса записать в csv-файл

```
psql -U postgres -d postgres
-c '\! chcp 1251'\ -- настройка кодовой страницы
-c 'DROP TABLE IF EXISTS fact_t;'\
-c 'CREATE TABLE fact_t AS
  SELECT n, factorial(n) AS fact
  FROM (VALUES (5),(8),(12)) AS t(n);'\
-c "COPY fact_t TO 'E:/[WorkDirectory]/GARBAGE/fact_out.csv' WITH (FORMAT csv);"
```

Загрузить файл `gemini_ethusd_d.csv` в подготовленную таблицу `gemini`

```
psql -U postgres -d postgres
-- создать таблицу в текущей БД
-c 'CREATE TABLE gemini (
    date timestamp,
    symbol text not null,
    open real not null,
    high real not null,
    low real not null,
    close real not null,
    volume_eth real not null,
    volume_usd real not null
    PRIMARY KEY (date));'
-- записать в подготовленную таблицу csv-файл
-c "COPY gemini FROM 'E:/.../gemini_ethusd_d.csv' WITH (HEADER true, FORMAT csv);"
```

Замечание

Лучше для нескольких команд использовать несколько ключей `-c`

Прочитать команды из файла `sql_query.sql`, а не из стандартного ввода. По большому счету ключ `-f` равнозначен метакоманде `\i`

```
$ psql -U postgres -f sql_query.sql
```

Вывести первые 3 строки таблицы в HTML-формате (`-H`)

```
$ psql -U postgres -d postgres -H -c '! chcp 1251' -c 'table family limit 3;'
```

Записать вывод результатов всех запросов в файл с именем `psql_output.txt`

```
$ psql -U postgres -d postgres -H -c '! chcp 1251' -c 'table family limit 3;' \
-o psql_output.txt
```

Вывести таблицу в формате `LATEX` и вывод запроса записать в файл `for_latex_output.txt`

```
$ psql -U postgres -d postgres -P format=latex \
-c '! chcp 1251' -c 'table family limit 3;' \
-o for_latex_output.txt
```

Вывести путь до каталога данных

```
$ psql -U postgres -c 'show data_directory;'
# /Users/leor.finkelberg/Library/Application Support/Postgres/var-11
```

Вывести версию Postgres

```
psql -U postgres -c 'select version();'
```

Создать базу данных из `sql`-файла с перенаправлением стандартного потока вывода и стандартного потока вывода ошибок в `log`-файл

```
psql -U postgres -f demo_small.sql &> demo.log
# то же, но создание базы данных происходит в фоновом режиме
psql -U postgres -f demo_small.sql &> demo.log &
```

Вывести список текущих настроек (пусть нас интересует номер порта)

```
psql -U postgres -c "show all;" | grep -inE port
```

44. Специальные функции и операторы

44.1. Предикаты ANY, ALL

Использование IN эквивалентно = ANY, а использование NOT IN эквивалентно <> ALL. Пример

```
-- есть ли совпадение хотя бы с одним элементом массива?
SELECT 'test'::text = ANY('{ "test", "non-test" }'::text[]); -- true
-- есть ли совпадение со всеми элементами массива?
SELECT 'test'::text = ALL('{ "test", "non-test" }'::text[]); -- false
```

44.2. Оператор конкатенации

Соединить два массива можно с помощью оператора ||

```
SELECT '{2,3,10}'::int[] || '{5}'::int[]; -- {2,3,10,5}
```

С помощью этого же оператора можно «склеивать» строки

```
SELECT name || '[' || job || ']' AS record FROM employees; -- Ben[ Junior Developer ]
```

К слову, у PostgreSQL нестрогая типизация. То есть PostgreSQL проводит неявное преобразование типов, например, когда используется оператор конкатенации, склеивающий строку и число, то выполняется неявное преобразование числового типа в строковый.

44.3. Диапазонные операторы

Подробнее вопрос обсуждается в документации PostgreSQL [9.19. Диапазонные функции и операторы](#).

С помощью оператора @> удобно проверять покрывает ли диапазон слева от оператора диапазон справа от оператора

```
SELECT '{2,3,5}'::int[] @> '{3}'::int[] -- true
```

Можно проверить содержит ли диапазон слева от оператора элемент справа от оператора

```
SELECT '[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp; --true
```

Еще можно с помощью оператора @> работать с json-объектами

```
-- здесь предполагается, что атрибут 'model' имеет тип 'jsonb'
-- вывести значения по указанному ключу в соответствующих строках
SELECT
    aircraft_code,
    model->'en' AS model
FROM
    aircrafts_data
WHERE
    model->'en' @> '"Airbus A321-200"'::jsonb;
```

Выведет

```
+-----+-----+
| aircraft_code |      model      |
+-----+-----+
| 321          | "Airbus A321-200" |
+-----+-----+
```


То есть во фразе **WHERE** мы выбираем те строки, у которых в json-объектах атрибута **model**, есть ключ **'en'**, имеющий значение **'Airbus A321-200'**. А затем во фразе **SELECT** выбираем только значения по указанному ключу в этих строках.

Если требуется выяснить содержится ли элемент в диапазоне, то используется оператор **<@**

```
SELECT 42 <@ int4range(2,10); --false
-- принадлежит ли 3 полуотрезку [2,10)
SELECT 3 || ' in ' || int4range(2,10) AS cond, 3 <@ int4range(2,10) AS bool; --true
```

Общие элементы (пересечение) удобно выявлять с помощью оператора **&&**

```
SELECT '{3}':int[] && '{2,3,5}':int[]; --true
```

44.4. Функции, генерирующие ряды значений

Очень полезна бывает функция **generate_series(start, stop, step)**, которая возвращает последовательность в общем случае вещественных чисел от *start* до *stop* с заданным шагом *step*

```
SELECT generate_series(1.1, 4, 1.3) AS real_num;
```

ВЫВЕДЕТ

```
+-----+
| real_num |
+-----+
|      1.1 |
|      2.4 |
|      3.7 |
+-----+
(3 строки)
```

Сгенерировать несколько временных меток с шагом в 5 часов

```
SELECT generate_series('2020-05-01 00:00':timestamp,
                       '2020-05-01 22:00':timestamp,
                       '5 hours':interval) AS dates;
```

ВЫВЕДЕТ

```
+-----+
|      dates      |
+-----+
| 2020-05-01 00:00:00 |
| 2020-05-01 05:00:00 |
| 2020-05-01 10:00:00 |
| 2020-05-01 15:00:00 |
| 2020-05-01 20:00:00 |
+-----+
(5 строк)
```

Все функции, упомянутые во фразе **FROM**, могут использовать результаты любых предшествующих функций или подзапросов (указанных в этой же фразе **FROM**)

```
SELECT a, b
FROM
    generate_series(1,3) AS a,
    generate_series(a,a+2) AS b;
```

ВЫВЕДЕТ

```

+---+---+
| a | b |
+---+---+
| 1 | 1 |
| 1 | 2 |
| 1 | 3 |
| 2 | 2 |
| 2 | 3 |
| 2 | 4 |
| 3 | 3 |
| 3 | 4 |
| 3 | 5 |
+---+---+
(9 строк)

```

Когда после функции в предложении FROM добавляется WITH ORDINALITY, в выходные данные добавляется столбец типа bigint, числа в котором начинаются с 1 и увеличиваются на 1 для каждой строки, выданной функцией. Пример

```
SELECT * FROM pg_ls_dir('.') WITH ORDINALITY AS t(ls, n) LIMIT 5;
```

ВЫВЕДЕТ

```

+-----+---+
|      ls      | n |
+-----+---+
| base         | 1 |
| current_logfiles | 2 |
| global       | 3 |
| log          | 4 |
| pg_commit_ts | 5 |
+-----+---+

```

Результаты нескольких табличных функций можно объединить, как если бы они были соединены по позиции строки. Для этого служит конструкция ROWS FROM. Эта конструкция возвращает *отношение*, поэтому ее можно использовать во фразе FROM, как любое другое отношение. Количество строк равно размеру самого большого результата перечисленных функций. Столбцы соответствуют функциям во фразе ROWS FROM. Если какая-то функция возвращает меньше строк, чем другие, то отсутствующие значения будут равны <NULL>

```

SELECT idx, number FROM
  ROWS FROM (
    generate_series(
      '2020-07-06'::timestamp,
      '2020-10-05'::timestamp,
      '14 days'::interval
    ),
    generate_series(100,105)
  ) AS t(idx, number);

```

ВЫВЕДЕТ

```

+-----+-----+
|      idx      | number |
+-----+-----+
| 2020-07-06 00:00:00 |    100 |
| 2020-07-20 00:00:00 |    101 |
| 2020-08-03 00:00:00 |    102 |

```

2020-08-17 00:00:00	103
2020-08-31 00:00:00	104
2020-09-14 00:00:00	105
2020-09-28 00:00:00	<NULL>

44.5. Функции для работы с массивами

Пример использования функции `unnest`, которая преобразует заданный массив в набор строк, каждая запись которого соответствует элементу массива

```
SELECT unnest('{2,3,4}'::int[]) AS idx, 5 AS number;
```

Выведет

idx	number
2	5
3	5
4	5

Работа с несколькими массивами

```
SELECT * FROM unnest('{2,3,4}'::int[], '{10,20,30,40}'::int[]);
```

44.6. Операторы и функции даты/времени

Вычислить возраст, например, пользователя приложения по дате его рождения удобно с помощью функции `age`

```
SELECT age('1986-08-18'::timestamp); -- 33 years 10 mons 16 days
```

В этом же запросе можно извлечь, например, только полное число лет

```
SELECT extract('year' from age('1986-08-18'::timestamp)); -- 33
```

Перекрытие временных диапазонов можно определить с помощью SQL-оператора `OVERLAPS`.

Пример

```
SELECT
  ('2001-02-16'::date, '2001-12-21'::date)
OVERLAPS
  ('2001-10-30'::date, '2002-10-30'::date); -- t
```

«Усечь» дату до заданной точности можно так

```
SELECT date_trunc('year', '2001-02-16 20:38:40'::timestamp); -- 2001-01-01 00:00:00
```

44.7. Выполнение динамически формируемых команд

Часто требуется динамически формировать команды внутри функции на PL/pgSQL, то есть такие команды, в которых при каждом выполнении могут использоваться разные таблицы или типы данных. Обычно PL/pgSQL кеширует планы выполнения, но в случае с динамическими командами это не будет работать.

Для исполнения динамических команд предусмотрен оператор `EXECUTE`

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ...] ];
```

где `command-string` – это выражение, формирующее строку (типа `text`) с текстом команды, которую нужно выполнить. Необязательная цель – это переменная-запись, переменная-кортеж или разделенный запятыми список простых переменных и полей записи/кортежа, куда будут помещены результаты команды. Необязательные выражения в `USING` формируют значения, которые будут вставлены в команду.

В сформированном тексте команды замена имен переменных PL/pgSQL на их значения проводиться не будет. Все необходимые значения переменных должны быть вставлены в командную строку при ее построении, либо нужно использовать параметры, как описано ниже.

Также, *нет никакого плана кеширования* для команд, выполняемых с помощью `EXECUTE`. Вместо этого план создается каждый раз при выполнении. Таким образом, строка команды может динамически создаваться внутри функции для выполнения действий с различными таблицами и столбцами.

Предложение `INTO` указывает, куда должны быть помещены результаты SQL-команды, возвращающей строки. Если используется переменная строкового типа или список переменных, то они должны в точности соответствовать структуре результата запроса (когда используется переменная типа `record`, она автоматически приводится к строковому типу результата запроса). Если возвращается несколько строк, то только первая будет присвоена переменной(ым) в `INTO`. Если не возвращается ни одной строки, то присваивается `NULL`. Без предложения `INTO` результаты запроса отбрасываются.

С указанием `STRICT` запрос должен вернуть ровно одну строку, иначе выдается сообщение об ошибке.

В тексте команды можно использовать значения параметров. Ссылки на эти параметры обозначаются как `$1`, `$2` и т.д. Эти символы указывают на значения, находящиеся в предложении `USING`. Такой метод зачастую предпочтительнее, чем вставка значений в команду в виде текста: он позволяет исключить во время исполнения дополнительные расходы на преобразования значений в текст и обратно, и не открывает возможности для SQL-инъекций, не требуя применять экранирование или кавычки для спецсимволов.

Пример

```
...
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
  INTO c -- число строк связывается с переменной 'c'
  USING checked_user, checked_date;
...
```

Обратите внимание, что *символы параметров* можно использовать *только* вместо значений данных. Если же требуется *динамически* формировать имена таблиц или столбцов, их необходимо вставлять в виде текста. Например, если в предыдущем запросе необходимо динамически задавать имя таблицы, можно сделать следующее

```
...
EXECUTE 'SELECT count(*) FROM '
  || quote_ident(tabname)
  || ' WHERE inserted_by = $1 AND inserted <= $2'
  INTO c
  USING checked_user, checked_date;
...
```

В качестве более аккуратного решения, вместо имени таблиц или столбцов можно использовать указание формата %I с функцией `format()` (текст, разделенный символами новой строки, соединяется вместе)

```
...
-- самый аккуратный вариант
-- символы $1 и $2 указывают на значения в предложении USING
EXECUTE format('SELECT count(*) FROM %I '
               'WHERE inserted_by = $1 AND inserted <= $2', tablename)
INTO c
USING checked_user, checked_date;
...
```

Замечание

Имена таблиц и столбцов для динамического построения запроса следует вставлять с помощью функции `format()`, а значения атрибутов – с помощью предложения `USING` (к атрибутам можно обращаться как \$1, \$2 и т.д.)

Команда `EXECUTE` с неизменяемым текстом и параметрами `USING`, функционально эквивалентна команде, записанной напрямую в PL/pgSQL, в которой переменные PL/pgSQL автоматически заменяются значениями. Важное отличие в том, что `EXECUTE` при каждом исполнении заново строит план команды с учетом текущих значений параметров, тогда как PL/pgSQL строит *общий план* выполнения и *кеширует* его при повторном использовании. В тех случаях, когда наилучший план выполнения сильно зависит от значений параметров, может быть полезно использовать `EXECUTE` для *гарантии* того, что не будет выбран общий план.

Указание %I равнозначно вызову `quote_ident`, а %L – вызову `quote_nullable`.

44.8. Функции и операторы JSON

Рассмотрим примеры использования операторов для типов `json` и `jsonb`

```
-- можно указывать имя ключа в json-объекте или номер этого элемента
-- нумерация начинается с 0
SELECT '[
  { "solver type" : ["direct", "iterative"] },
  { "package name" : "Ansys" }
]':::json->0->'solver type'->>1; -- iterative
```

Использование нотации пути

```
SELECT '[
  { "solver type" : ["direct", "iterative"] },
  { "package name" : "Ansys" }
]':::json #>> '{0, "solver type", 1}'; -- iterative
```

Примеры использования дополнительных операторов `jsonb`

```
-- содержит ли левый json-объект правый json-объект
SELECT '{
  "solver type" : [ "iterative", "direct" ],
  "package name" : "Ansys"
}':::jsonb @> '{ "solver type" : [ "iterative" ] }':::jsonb; -- t
```

```
-- содержится ли левый json-объект в правом
SELECT '{ "package name" : "Ansys" }':::jsonb <@ '{ "package name" : "Ansys", "solver type" : "
  direct" }':::jsonb;
```

```
-- присутствует ли ключ в формате СТРОКИ в данном json-объекте
SELECT '{ "package name" : "Nastran",
        "solver type" : [ "iterative", "direct" ] }'::jsonb ? 'solver type'; -- t
```

```
-- содержит ли данный json-объект ХОТЯ БЫ один элемент из указанного массива СТРОК
SELECT '{
    "package name" : "Abaqus",
    "solver type" : "direct"
}'::jsonb ?| '{ "package name", "solver type" }'::text[];
```

```
-- содержит ли данный json-объект ВСЕ элемент из указанного массива СТРОК
SELECT '{
    "package name" : "Abaqus",
    "solver type" : "direct"
}'::jsonb ?& '{ "package name", "solver type" }'::text[];
```

```
-- удаляет пару ключ/значение или элемент-СТРОКУ из левого операнда
SELECT '{
    "package name" : "Abaqus",
    "solver type" : "direct"
}'::jsonb - 'solver type';
```

```
-- удаляет поле или элемент с заданным путем
SELECT '{
    "package name" : "Abaqus",
    "solver type" :
        {
            "direct" : "Java",
            "iterative" : "IronPython"
        }
}'::jsonb #- '{"solver type", "iterative"}'::text[];
```

Чтобы убедиться в том, что найден правильный элемент json-объект, следует сначала воспользоваться оператором #>, а уже потом оператором #-.

Функции json_each() и jsonb_each() разворачивают внешний объект JSON в набор пар ключ/значение

```
SELECT * FROM jsonb_each(
    '{
        "package name" : "Abaqus",
        "solver type" : { "direct" : "Java", "iterative" : "IronPython" }
    }'::jsonb #- '{"solver type", "direct"}'::text[]
);
```

ВЫВЕДЕТ

key	value
solver type	{ "iterative": "IronPython" }
package name	"Abaqus"

Для того чтобы развернуть внешний объект JSON в набор пар ключ/значение в текстовом формате, следует воспользоваться функциями json_each_text() и jsonb_each_text().

Ключи JSON-объекта можно вернуть так

```
-- выведет ключи JSON-объекта верхнего уровня
SELECT jsonb_object_keys(
  '{
    "package name" : "Nastran",
    "solver type" : { "direct" : "Java", "iterative" : "IronPython" }
  }'::jsonb #- '{"solver type",direct}'::text[]
);
```

Обновить или создать новый элемент JSON-объект можно так

```
-- заменим 'direct' на 'DIRECT', так как 'create_missing' = false
SELECT jsonb_set(
  '{
    "package name" : "Ansys",
    "solver type" : [ "direct", "iterative" ]
  }'::jsonb, -- где меняем
  '{"solver type",0}'::text[], -- путь в json-объекте
  '"DIRECT"'::jsonb, -- на что меняем
  false -- обновляем значение
  -- true -- добавляем новое значение
);
```

```
-- добавит новое значение 'NEW TYPE' в 'solver type', так как в этом ключе нет элемента с номером
  позиции 2, а 'create_missing' = true
SELECT jsonb_set(
  '{
    "package name" : "Ansys",
    "solver type" : [ "direct", "iterative" ]
  }'::jsonb, -- где меняем
  '{"solver type",2}'::text[], -- путь в json-объекте
  '"NEW TYPE"'::jsonb, -- на что меняем
  true -- добавляем новое значение
);
```

Для того чтобы упростить восприятие сложных JSON-объектов служит функция json_pretty()

```
SELECT * FROM jsonb_pretty(
  '{ "pretty name" : "Ansys",
    "solver type" : [ "direct", "iterative" ]
  }'::jsonb
) AS "json object";
```

ВЫВЕДЕТ

```
+-----+
|          json object          |
+-----+
| {                               +|
|   "pretty name": "Ansys",      +|
|   "solver type": [            +|
|       "direct",               +|
|       "iterative"            +|
|   ]                           +|
| }                               |
+-----+
```

45. Приемы работы с JSONPATH

45.1. Элементы языка путей SQL/JSON

Подробности в документации <https://postgrespro.ru/docs/postgresql/12/functions-json>.

В PostgreSQL «путевые выражения» реализованны как тип данных `jsonpath` и потому может использоваться для описания объектов.

Рассмотрим пример

```
SELECT jsonb_path_query( -- специальная функция для построения JSON-запроса
    '{
      "track" : {
        "segments" : [
          {
            "location" : [ 47.763, 13.4034 ],
            "start time" : "2018-10-14 10:05:14",
            "HR" : 73
          },
          {
            "location" : [ 47.706, 13.2635 ],
            "start time" : "2018-10-14 10:39:21",
            "HR" : 135
          }
        ]
      }
    }'::jsonb, -- целевой JSONB-объект
    '$.track.segments[*] ? (@.HR > 100).start time' -- "2018-10-14 10:39:21"
);
```

Можно задавать цепочку условий

```
select jsonb_path_query(
    '{
      "track" : {
        "segments" : [
          {
            "location" : [ 47.763, 13.4034 ],
            "start time" : "2018-10-14 10:05:14",
            "HR" : 73
          },
          {
            "location" : [ 47.706, 13.2635 ],
            "start time" : "2018-10-14 10:39:21",
            "HR" : 135
          }
        ]
      }
    }'::jsonb,
    '$.track.segments[*] ? (@.location[1] > 13.35) ? (@.HR > 70).start time' -- "2018-10-14
    10:05:14"
);
```

Также допускается использование выражений фильтра на разных уровнях вложенности. Следующий пример сначала фильтрует все сегменты по местоположению, а затем возвращает высокие значения частоты сердечных сокращений для этих сегментов, если они доступны

```
select jsonb_path_query(
    '{
      "track" : {
```



```

    "segments" : [
      {
        "location" : [ 47.763, 13.4034 ],
        "start time" : "2018-10-14 10:05:14",
        "HR" : 73
      },
      {
        "location" : [ 47.706, 13.2635 ],
        "start time" : "2018-10-14 10:39:21",
        "HR" : 135
      }
    ]
  }
}'::jsonb,
'$$.track.segments[*] ? (@.location[1] > 13.35).HR ? (@ > 70)'
);

```

В JSON-путях поддерживается два режима:

- **lax** – неявно преобразует запрашиваемые данные к указанному пути, подавляя структурные ошибки,
- **strict** – в случае появления структурной ошибки, выводит сообщение об ошибке.

Пример

```

-- выведет сообщение об ошибке, т.к. индекс массива вне диапазона
SELECT jsonb_path_query(
  '{ "a" : [ 10, 20, 30 ] }'::jsonb,
  'strict $.a[3] ? (@ > 20)'
);

```

С помощью фильтра `like_regex` поддерживаются регулярные выражения

```

SELECT jsonb_path_query(
  '{
    "track" : {
      "segments" : [
        {
          "location" : [ 47.763, 13.4034 ],
          "start time" : "2018-10-14 10:05:14",
          "HR" : 73
        },
        {
          "location" : [ 47.706, 13.2635 ],
          "start time" : "2018-10-14 10:39:21",
          "HR" : 135
        }
      ]
    },
    "package name" : [ "Ansys", "Nastran", "Abaqus" ]
  }'::jsonb,
  '$$.track."package name" ? (@ like_regex "^a[bn].*" flag "i")'
); -- "Ansys", "Abaqus"

```

Разумеется результаты работы этой функции можно комбинировать с JSON-операторами

```

SELECT test_json.obj->'package name' AS "package name",
  jsonb_path_query(
    obj,
    '$."solver type"[*] ? (@ like_regex "^iter.*" flag "i")'
  ) AS "solver type"
FROM test_json;

```

45.2. Элементы языка JQuery

jQuery – язык запросов к типу данных JSONB. Основное предназначение – предоставить дополнительную функциональность для JSONB, например, простой и эффективный способ поиска во вложенных объектах и массивах, а также дополнительные операторы сравнения с поддержкой индексов.

Установить расширение `jquery` можно так

```
CREATE EXTENSION jquery;
```

45.3. Примеры конструкций с JSONPATH

В PostgreSQL 12 появились несколько новых функций и операторов, упрощающих работу с JSON-объектами.

Пример на использование оператора `@?` (*оператора существования*⁹)

```
-- проверка на условие в круглых скобках
SELECT '{ "a" : [ 1,2,3,4,5], "b" : 100 }'::jsonb @? '$.a[*] ? (@ < 1)'; -- t
```

Здесь, если хотя бы для одного элемента массива [...] выполняется условие (`@ < 1`), то возвращается `true`.

Похожий пример с использованием регулярных выражений

```
SELECT '{ "language" : [ "Python", "Java", "IronPython" ] }'::jsonb @? '$."language"[*] ? (@ like_regex ".*python$" flag "i")'; -- true
```

Чтобы вернуть элементы, отвечающие условию удобнее использовать функцию `jsonb_path_query()`

```
SELECT jsonb_path_query(
  '{ "language" : [ "Python", "Java", "IronPython" ] }'::jsonb,
  '$.language[*] ? ( @ like_regex ".*python" flag "i" )'
);
```

Существует еще *оператор совпадения*¹⁰ `@@`

```
SELECT '[ 1,2,3 ]'::jsonb @@ '$[*] == 3' -- true
```

Подобные задачи еще можно решить с помощью специальной функции `jsonb_path_exists`

```
SELECT jsonb_path_exists(
  '{ "a" : [ 1,2,3,4,5 ] }'::jsonb, -- целевой JSON-объект
  '$.a[*] ? (@ => $min && @ <= $max)', -- JSON-путь
  '{ "min" : 2, "max" : 4 }' -- пользовательские переменные
);
-- или с помощью @?
SELECT '{ "a" : [ 1,2,3,4,5 ] }'::jsonb @? '$.a[*] ? ( @ >= 2 && @ <= 4)';
```

Еще пример на использование оператора `@?`

```
SELECT '{ "package name" : "Ansys",
  "solver type" : [ "direct", "iterative" ]
}'::jsonb @? '$."solver type"[*] ? (@ == "direct")'; -- true
```

Эту задачу можно было бы решить и так

⁹Еще этот оператор можно прочитать так: выдает ли путь JSON какой-либо элемент для заданного значения JSON?

¹⁰Оператор возвращает результат проверки предиката пути JSON для заданного значения JSON. При этом учитывается только первый элемент результата. Если результат не является булевым, то возвращается `null`

```
SELECT jsonb_path_exists(
  '{ "package name" : "Ansys",
    "solver type" : [ "direct", "iterative " ]
  }'::jsonb, -- целевой JSON-объект
  '$."solver type"[*] ? (@ == $var)', -- JSON-путь
  '{ "var" : "direct" }'::jsonb -- пользовательская переменная
);
```

Вывести элементы JSON-объекта, отвечающие заданному условию, можно так

```
SELECT jsonb_path_query AS res FROM jsonb_path_query(
  '{ "a" : [ 1,2,3,4,5 ] }'::jsonb,
  '$.a[*] ? (@ >= $min && @ <= $max)',
  '{ "min" : 2, "max" : 4 }'::jsonb
);
```

ВЫВЕДЕТ

```
res |
--- |
2   |
3   |
4   |
```

Получить сгруппированный вариант в формате JSONB можно так

```
SELECT jsonb_path_query_array AS res FROM jsonb_path_query_array(
  '{ "a" : [ 1,2,3,4,5 ] }'::jsonb,
  '$.a[*] ? (@ >= $min && @ <= $max)',
  '{ "min" : 2, "max" : 4 }'::jsonb
);
```

Аналогично можно использовать различные математические операторы

```
-- оператор взятия остатка от деления
SELECT jsonb_path_query(''[9]''::jsonb, '$[0] % 2'); -- 1
```

Округление в меньшую сторону

```
SELECT jsonb_path_query(
  '{ "x" : [ 2.85, -14.7, -9.4 ] }'::jsonb,
  '+ $.x.floor()
);
```

То же самое с обращением знака

```
SELECT jsonb_path_query(
  '{ "x" : [ 2.85, -14.7, -9.4 ] }'::jsonb,
  '- $.x.floor()
);
```

Выбрать из JSON-объекта только те CAE-пакеты, имена которых без учета регистра начинаются с символа a, затем идет либо символ b, либо n, а затем произвольная последовательность символов

```
-- шаблон: структура JSON-объекта ? (условие)
SELECT jsonb_path_query(
  '{ "package name" : [ "Ansys", "Nastran", "Abaqus", "Comsol" ] }'::jsonb,
  '$."package name" ? (@ like_regex "^a[bn].*" flag "i")'
);
```

46. Наиболее полезные команды PostgreSQL

46.1. Команда VACUUM

Команда `VACUUM` предназначена для сборки мусора и, возможно, для анализа базы данных. `VACUUM` высвобождает пространство, занятое «мертвыми» кортежами. При обычных операциях кортежи, удаленные или устаревшие в результате обновления, физически не удаляются из таблицы. Они сохраняются в ней, и ждут выполнения команды `VACUUM`. Таким образом, периодически необходимо выполнять `VACUUM`, особенно для часто изменяемых таблиц.

Без параметра команда `VACUUM` обрабатывает *все таблицы* текущей базы данных, которые может очистить текущий пользователь. Если в параметре передается *имя таблицы*, то `VACUUM` обрабатывает *только эту таблицу*.

Конструкция `VACUUM ANALYZE` выполняет очистку, а затем анализ всех указанных таблиц. Параметр `ANALYZE` обновляет статистическую информацию по таблицам, которую использует планировщик при выборе наиболее эффективного способа выполнения запроса. Это удобная комбинация для регулярного обслуживания БД.

Пример

```
VACUUM (VERBOSE, ANALYZE) onek;
```

47. Обработка временных рядов в TimescaleDB

TimescaleDB <https://www.timescale.com/> – расширение PostgreSQL предназначенное для работы с временными рядами. Временные ряды можно хранить и PostgreSQL и просто так, но TimescaleDB обеспечивает большую производительность на том же железе. Кроме того TimescaleDB предлагает набор фичей адаптированных для эффективной работы с временными рядами.

С TimescaleDB поддерживается множество языков программирования, в частности Python <https://docs.timescale.com/quick-start/latest/python/>, с которым можно работать через библиотеку `psycopg2`

```
import psycopg2

CONNECTION = "postgres://username:password@host:port/dbname"
# CONNECTION = "postgres://username:password@host:port/dbname?sslmode=require"
# или так
# CONNECTION = "dbname=tsdb user=tsdbadmin password=secret host=host.com port=5432 sslmode=require"
```

Создать объект соединения и объект курсора можно так

```
CONNECTION = "postgres://username:password@host:port/dbname"
with psycopg2.connect(CONNECTION) as conn:
    cursor = conn.cursor()
    # use the cursor to interact with your database
    # cursor.execute("SELECT * FROM table")
```

Создать таблицу можно так

```
query_create_sensors_table = """CREATE TABLE sensors (
    id SERIAL PRIMARY KEY,
    type VARCHAR(50),
    location VARCHAR(50)
);
```

```
"""
```

Выполняем запрос, сохраняем изменения (commit), закрываем объект курсора

```
cursor = conn.cursor()
# see definition in Step 1
cursor.execute(query_create_sensors_table)
conn.commit()
cursor.close()
```

Теперь можно создать гипертаблицу

```
# create sensor data hypertable
query_create_sensordata_table = """CREATE TABLE sensor_data (
    time TIMESTAMPTZ NOT NULL,
    sensor_id INTEGER,
    temperature DOUBLE PRECISION,
    cpu DOUBLE PRECISION,
    FOREIGN KEY (sensor_id) REFERENCES sensors (id)
);
"""

# требуется указать имя таблицы, которую нужно преобразовать в гипертаблицу и имя столбца с временной меткой
query_create_sensordata_hypertable = "SELECT create_hypertable('sensor_data', by_range('time'));"

cursor = conn.cursor()
cursor.execute(query_create_sensordata_table)
cursor.execute(query_create_sensordata_hypertable)
# commit changes to the database to make changes persistent
conn.commit()
cursor.close()
```

Вставка данных

```
sensors = [('a', 'floor'), ('a', 'ceiling'), ('b', 'floor'), ('b', 'ceiling')]
cursor = conn.cursor()
for sensor in sensors:
    try:
        cursor.execute(
            "INSERT INTO sensors (type, location) VALUES (%s, %s);",
            (sensor[0], sensor[1])
        )
    except (Exception, psycopg2.Error) as error:
        print(error.pgerror)
conn.commit()

cursor = conn.cursor()
query = "SELECT * FROM sensor_data;"
cursor.execute(query)
for row in cursor.fetchall():
    print(row)
cursor.close()
```

Если требуется получить список словарей, то можно сделать так

```
cursor = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)
```

47.1. Установка TimescaleDB

Инструкции по установке TimescaleDB можно найти на странице <https://docs.timescale.com/self-hosted/latest/install/> (раздел «Other deployment options»)

47.1.1. На Linux

47.1.2. На MacOS

47.1.3. Сборка из исходников

<https://eax.me/timescaledb/>

Список литературы

1. *Джуба С., Волков А.* Изучаем PostgreSQL 10. – М.: ДМК Пресс, 2019. – 400 с.
2. *Домбровская Г.* Оптимизация запросов в PostgreSQL. Полное руководство по созданию эффективных запросов, 2022
3. *Чакон С., Штрауб Б.* Git для профессионального программиста. – СПб.: Питер, 2020. – 496 с.
4. *Собель М.* Linux. Администрирование и системное программирование. 2-е изд. – СПб.: Питер, 2011. – 880 с.

Листинги

- 1 [Функция на PL/pgSQL, возвращающая подвыборку по условию](#) 87