PostgreSQL

- Определение структуры данных
  - Внешние ключи
- Отладка
  - Отладочные запросы
  - Выгрузка результатов запросов
  - Простое обнаружение проблем производительности в PostgreSQL
  - Слоты репликации
- Функции и операторы
  - Оператор DISTINCT
- Управляющие подключениями
  - PgBouncer
- Блокировки
  - Блокировки отношений
- Динамическая компиляция SQL-запросов
  - Динамическая компиляция SQL-запросов в PostgreSQL с использованием LLVM JIT

?????????? ??????????
??????

??????? ?????

Для связи между таблицами применяются внешние ключи. Внешний ключ устанавливается для столбца из зависимой, подчиненной таблицы (**referencing table**), и указывает на один из столбцов из главной таблицы (**referenced table**). Как правило, внешний ключ указывает на первичный ключ из связанной главной таблицы.

????? ?????????? ??????????
???????? ????? ?? ?????? ???????

```
REFERENCES главная_таблица (столбец_главной_таблицы)
    [ON DELETE {CASCADE|RESTRICT}]
    [ON UPDATE {CASCADE|RESTRICT}]
```

Чтобы установить связь между таблицами, после ключевого слова **REFERENCES** указывается имя связанной таблицы и далее в скобках имя столбца из этой таблицы, на который будет указывать внешний ключ. После выражения **REFERENCES** может идти выражение **ON DELETE** и **ON UPDATE**, которые уточняют поведение при удалении или обновлении данных.

????? ?????????? ??????????
???????? ????? ?? ?????? ???????

```
FOREIGN KEY (стобец1, столбец2, ... столбецN)
    REFERENCES главная_таблица (столбец_главной_таблицы1, столбец_главной_таблицы2, ...
столбец_главной_таблицыN)
    [ON DELETE {CASCADE|RESTRICT}]
    [ON UPDATE {CASCADE|RESTRICT}]
```

Например, определим две таблицы и свяжем их посредством внешнего ключа:

```
CREATE TABLE Customers
(
```

```
    Id SERIAL PRIMARY KEY,
    Age INTEGER,
    FirstName VARCHAR(20) NOT NULL
);

CREATE TABLE Orders
(
    Id SERIAL PRIMARY KEY,
    CustomerId INTEGER REFERENCES Customers (Id),
    Quantity INTEGER
);
```

Здесь определены таблицы **Customers** и **Orders**. **Customers** является главной и представляет клиента. **Orders** является зависимой и представляет заказ, сделанный клиентом. Эта таблица через столбец **CustomerId** связана с таблицей **Customers** и ее столбцом **Id**. То есть столбец **CustomerId** является внешним ключом, который указывает на столбец **Id** из таблицы **Customers**.

Определение внешнего ключа на уровне таблицы выглядело бы следующим образом:

```
CREATE TABLE Customers
(
    Id SERIAL PRIMARY KEY,
    Age INTEGER,
    FirstName VARCHAR(20) NOT NULL
);

CREATE TABLE Orders
(
    Id SERIAL PRIMARY KEY,
    CustomerId INTEGER,
    Quantity INTEGER,
    FOREIGN KEY (CustomerId) REFERENCES Customers (Id)
);
```

ON DELETE ? ON UPDATE

С помощью выражений **ON DELETE** и **ON UPDATE** можно установить действия, которые выполняются соответственно при удалении и изменении связанной строки из главной таблицы. Для установки подобного действия можно использовать следующие опции:

- **CASCADE**: автоматически удаляет или изменяет строки из зависимой таблицы при удалении или изменении связанных строк в главной таблице.
- **RESTRICT**: предотвращает какие-либо действия в зависимой таблице при удалении или изменении связанных строк в главной таблице. То есть фактически какие-либо действия отсутствуют.
- **NO ACTION**: действие по умолчанию, предотвращает какие-либо действия в зависимой таблице при удалении или изменении связанных строк в главной таблице. И генерирует ошибку. В отличие от **RESTRICT** выполняет отложенную проверку на связанность между таблицами.
- **SET NULL**: при удалении связанной строки из главной таблицы устанавливает для столбца внешнего ключа значение **NULL**.
- **SET DEFAULT**: при удалении связанной строки из главной таблицы устанавливает для столбца внешнего ключа значение по умолчанию, которое задается с помощью атрибуты **DEFAULT**. Если для столбца не задано значение по умолчанию, то в качестве него применяется значение **NULL**.

????????? ????????

По умолчанию, если на строку из главной таблицы по внешнему ключу ссылается какая-либо строка из зависимой таблицы, то мы не сможем удалить эту строку из главной таблицы. Вначале нам необходимо будет удалить все связанные строки из зависимой таблицы. И если при удалении строки из главной таблицы необходимо, чтобы были удалены все связанные строки из зависимой таблицы, то применяется каскадное удаление, то есть опция **CASCADE** :

```
CREATE TABLE Orders
(
    Id SERIAL PRIMARY KEY,
    CustomerId INTEGER,
    Quantity INTEGER,
    FOREIGN KEY (CustomerId) REFERENCES Customers (Id) ON DELETE CASCADE
);
```

Аналогично работает выражение **ON UPDATE CASCADE**. При изменении значения первичного ключа автоматически изменится значение связанного с ним внешнего ключа. Но так как первичные ключи, как правило, изменяются очень редко, да и с принципе не рекомендуется использовать в качестве первичных ключей столбцы с изменяемыми значениями, то на практике выражение **ON UPDATE** используется редко.

????????? NULL

При установки для внешнего ключа опции **SET NULL** необходимо, чтобы столбец внешнего ключа допускал значение **NULL**:

```
CREATE TABLE Orders
(
    Id SERIAL PRIMARY KEY,
    CustomerId INTEGER,
    Quantity INTEGER,
    FOREIGN KEY (CustomerId) REFERENCES Customers (Id) ON DELETE SET NULL
);
```

?????????? ????????? ?? ??????????

```
CREATE TABLE Orders
(
    Id SERIAL PRIMARY KEY,
    CustomerId INTEGER DEFAULT 1,
    Quantity INTEGER,
    FOREIGN KEY (CustomerId) REFERENCES Customers (Id) ON DELETE SET DEFAULT
);
```

Если для столбца значение по умолчанию не задано через параметр **DEFAULT**, то в качестве такового используется значение **NULL** (если столбец допускает **NULL**).

???????

Отладка

?????????? ???????

????? ?????? ???????? ???????? ? ??? ???????????

```
SELECT
    pid,
    datname,
    usename,
    client_addr,
    xact_start,
    query_start,
    (state_change-query_start) AS query_duration,
    wait_event_type,
    wait_event,
    state,
    query
FROM
    pg_stat_activity
WHERE
    state != 'idle'
AND
    usename NOT IN ('postgres','replicator','zabbix');
```

20 ????? ?????????? ????????

```
SELECT
    substring(query, 1, 50) AS short_query,
    query,
    round(total_time::numeric, 2) AS total_time,
    calls,
    round(mean_time::numeric, 2) AS mean,
    round((100 * total_time / sum(total_time::numeric) OVER ())::numeric, 2) AS percentage_cpu
FROM
```

```
    pg_stat_statements
ORDER BY
    total_time DESC LIMIT 20;
```

????? ?????????? ???????? ? ?????????? ??????????

```
SELECT
  max_conn,
  used,
  res_for_super,
  max_conn-used-res_for_super res_for_normal
FROM
  (SELECT count(*) used FROM pg_stat_activity) t1,
  (SELECT setting::int res_for_super FROM pg_settings WHERE
name=$$superuser_reserved_connections$$) t2,
  (SELECT setting::int max_conn FROM pg_settings WHERE name=$$max_connections$$) t3;
```

?????????? ???????????? ? ??????? ??? ?????? ? ??????? ????????

```
SELECT count(*), client_addr, datname FROM pg_stat_activity GROUP BY client_addr, datname;
```

?????????? ???????????? ? ???? ?????? ? ??????? ??????? ????????

```
SELECT count(*), client_addr, datname FROM pg_stat_activity WHERE datname = 'db_name' GROUP BY
client_addr, datname;
```

???????? ???????????? ????????

? html

```
$ psql -h <host> -p <port> -U postgres -H -c "SQL command"
```

Отладка

??????? ???????????? ??????? ???????????????????? ? PostgreSQL

Существует ли в мире очень большая и крупная база данных, которая время от времени не страдает от проблем с производительностью? Держу пари, что их не так уж много. Поэтому каждый DBA (администратор базы данных), отвечающий за PostgreSQL, должен знать, как отслеживать потенциальные проблемы производительности, чтобы выяснить, что на самом деле происходит.

?????????? ???????????????? PostgreSQL ????? ?????????? ??????????

Многие думают, что изменение параметров в `postgresql.conf` — это реальный путь к успеху. Однако это не всегда так. Конечно, чаще всего хорошие параметры конфигурации базы данных очень полезны. Тем не менее, во многих случаях реальные проблемы будут возникать из-за странного запроса, скрытого глубоко в некоторой логике приложения. Даже вполне вероятно, что запросы, вызывающие реальные проблемы, не являются теми, на которые вы обратили внимание. Возникает естественный вопрос: как мы можем отследить эти запросы и выяснить, что на самом деле происходит? Мой любимый инструмент для этого — `pg_stat_statements`, который всегда должен быть включен по моему мнению, если вы используете PostgreSQL 9.2 или выше (пожалуйста, не используйте его в более старых версиях).

????????? pg_stat_statements

Чтобы включить `pg_stat_statements` на вашем сервере, измените следующую строку в `postgresql.conf` и перезапустите PostgreSQL:

```
shared_preload_libraries = 'pg_stat_statements'
```

После загрузки этого модуля на сервер PostgreSQL автоматически начнет собирать информацию. Хорошо то, что накладные расходы модуля действительно очень низкие (накладные расходы в основном просто «шум»).

Затем выполните следующую команду для создания представления, необходимого для доступа к данным:

```
CREATE EXTENSION pg_stat_statements;
```

Прелесть здесь в том, что тип запроса, который наиболее трудоемкий, естественно будет отображаться в верхней части списка. Лучший способ — пройтись от первого до, скажем, 10-го запроса и посмотреть, что там происходит.

По моему мнению, невозможно настроить систему без просмотра наиболее трудоемких запросов на сервере базы данных.

?????????? ?????? ??????????????????? PostgreSQL

`pg_stat_statements` может предложить гораздо больше, чем просто запрос и время, которое он занял. Вот структура представления:

```
test=# \d pg_stat_statements
View "public.pg_stat_statements"
Column               |     Type         | Collation | Nullable | Default
---------------------+------------------+-----------+----------+---------
 userid              | oid              |           |          |
 dbid                | oid              |           |          |
 queryid             | bigint           |           |          |
 query               | text             |           |          |
 calls               | bigint           |           |          |
 total_time          | double precision |           |          |
 min_time            | double precision |           |          |
 max_time            | double precision |           |          |
 mean_time           | double precision |           |          |
 stddev_time         | double precision |           |          |
 rows                | bigint           |           |          |
 shared_blks_hit     | bigint           |           |          |
 shared_blks_read    | bigint           |           |          |
 shared_blks_dirtied | bigint           |           |          |
```

```
 shared_blks_written | bigint           |           |          |
 local_blks_hit      | bigint           |           |          |
 local_blks_read     | bigint           |           |          |
 local_blks_dirtied  | bigint           |           |          |
 local_blks_written  | bigint           |           |          |
 temp_blks_read      | bigint           |           |          |
 temp_blks_written   | bigint           |           |          |
 blk_read_time       | double precision |           |          |
 blk_write_time      | double precision |           |          |
```

Вполне полезно посмотреть и на столбец `stddev_time`. Если стандартное отклонение велико, можно ожидать, что некоторые из этих запросов будут быстрыми, а некоторые — медленными, что может привести к ухудшению работы пользователей.

Столбец `rows` также может быть достаточно информативным. Предположим, что 1000 вызовов вернули 1.000.000.000 строк: фактически это означает, что каждый вызов в среднем возвращал 1 миллион строк. Легко понять, что возвращать столько данных все время не слишком хорошо.

Если требуется проверить, показывает ли определенный тип запроса плохую производительность при кэшировании, будет интересен `shared_*`. Вкратце: PostgreSQL может сообщить вам частоту обращений к кешу для каждого отдельного типа запроса в случае, если включен `pg_stat_statements`.

Также имеет смысл взглянуть на поля `temp_blks_*`. Каждый раз, когда PostgreSQL должен обратиться к диску для сортировки или материализации, потребуются временные блоки.

Наконец-то есть `blk_read_time` и `blk_write_time`. Обычно эти поля пусты, если не включен `track_io_timing`. Идея здесь заключается в том, чтобы иметь возможность измерять количество времени, которое определенный тип запроса тратит на ввод-вывод. Это позволит вам ответить на вопрос, привязана ли ваша система к вводу/выводу или к ЦП. В большинстве случаев рекомендуется включить **I/O timing**, поскольку это даст вам важную информацию.

?????? ? Java ? Hibernate

`pg_stat_statements` дает хорошую информацию. Однако в некоторых случаях запрос может быть прерван из-за переменной конфигурации:

```
test=# SHOW track_activity_query_size;
track_activity_query_size
---------------------------
```

```
1024
(1 row)
```

Для большинства приложений 1024 байта абсолютно достаточно. Однако обычно это не тот случай, если вы используете Hibernate или Java. Hibernate имеет тенденцию посылать безумно длинные запросы к базе данных, и поэтому код SQL может быть обрезан задолго до запуска соответствующих частей (например, предложение FROM и т.д.). Поэтому имеет смысл увеличить `track_activity_query_size` до более высокого значения (возможно 32.786).

???????? ???????? ??? ????????? ????? ???? ? PostgreSQL

Есть один запрос, который я нашел особенно полезным в этом контексте: Следующий запрос показывает 20 запросов, занимающих много времени:

```
test=# SELECT substring(query, 1, 50) AS short_query,
              round(total_time::numeric, 2) AS total_time,
              calls,
              round(mean_time::numeric, 2) AS mean,
              round((100 * total_time / sum(total_time::numeric) OVER ())::numeric, 2) AS
percentage_cpu
FROM  pg_stat_statements
ORDER BY total_time DESC
LIMIT 20;
           short_query                               | total_time | calls | mean |
percentage_cpu
-----------------------------------------------------+------------+-------+------
+----------------
SELECT name FROM (SELECT pg_catalog.lower(name) A   | 11.85      | 7     | 1.69 | 38.63
DROP SCHEMA IF EXISTS performance_check CASCADE;    | 4.49       | 4     | 1.12 | 14.64
CREATE OR REPLACE FUNCTION performance_check.pg_st  | 2.23       | 4     | 0.56 | 7.27
SELECT pg_catalog.quote_ident(c.relname) FROM pg_c  | 1.78       | 2     | 0.89 | 5.81
SELECT a.attname, +                                 | 1.28       | 1     | 1.28 | 4.18
SELECT substring(query, ?, ?) AS short_query,roun   | 1.18       | 3     | 0.39 | 3.86
CREATE OR REPLACE FUNCTION performance_check.pg_st  | 1.17       | 4     | 0.29 | 3.81
SELECT query FROM pg_stat_activity LIMIT ?;         | 1.17       | 2     | 0.59 | 3.82
CREATE SCHEMA performance_check;                    | 1.01       | 4     | 0.25 | 3.30
SELECT pg_catalog.quote_ident(c.relname) FROM pg_c  | 0.92       | 2     | 0.46 | 3.00
SELECT query FROM performance_check.pg_stat_activi  | 0.74       | 1     | 0.74 | 2.43
```

```
SELECT * FROM pg_stat_statements ORDER BY total_ti  | 0.56       | 1     | 0.56 | 1.82
SELECT query FROM pg_stat_statements LIMIT ?;       | 0.45       | 4     | 0.11 | 1.45
GRANT EXECUTE ON FUNCTION performance_check.pg_sta  | 0.35       | 4     | 0.09 | 1.13
SELECT query FROM performance_check.pg_stat_statem  | 0.30       | 1     | 0.30 | 0.96
SELECT query FROM performance_check.pg_stat_activi  | 0.22       | 1     | 0.22 | 0.72
GRANT ALL ON SCHEMA performance_check TO schoenig_  | 0.20       | 3     | 0.07 | 0.66
SELECT query FROM performance_check.pg_stat_statem  | 0.20       | 1     | 0.20 | 0.67
GRANT EXECUTE ON FUNCTION performance_check.pg_sta  | 0.19       | 4     | 0.05 | 0.62
SELECT query FROM performance_check.pg_stat_statem  | 0.17       | 1     | 0.17 | 0.56
(20 rows)
```

Последний столбец особенно примечателен: он показывает процент общего времени, потраченного на один запрос. Это поможет вам выяснить, насколько влияет запрос на общую производительность или не влияет.

?????????????? ??????????

- Обнаружение медленных запросов

????? ???????????

??????? ?????? ?????? ???????????

```
SELECT * FROM pg_replication_slots;
```

??????? ???? ???????????

```
SELECT pg_create_physical_replication_slot('slot_name');
```

??????? ???? ???????????

```
SELECT pg_drop_replication_slot('slot_name')
```

?????? ???????????

- Функции для системного администрирования

??????? ? ??????????

???????? DISTINCT

PostgreSQL оператор `DISTINCT` используется для удаления дубликатов из набора результатов. `DISTINCT` может использоваться только с операторами `SELECT`.

??????????

Синтаксис для оператора DISTINCT в PostgreSQL:

```
SELECT DISTINCT | DISTINCT ON (distinct_expressions) expressions FROM tables [WHERE conditions];
```

Параметры и аргументы

- `distinct_expressions` - выражения, используемые для удаления дубликатов.
- `expressions` - столбцы или вычисления, которые вы хотите получить.
- `tables` - таблицы, из которых вы хотите получить записи. В операторе FROM должна быть указана хотя бы одна таблица.
- `WHERE conditions` - необязательный. Условия, которые должны быть выполнены для записей, которые будут выбраны.

???????????

- Если в `DISTINCT` указано только одно выражение, запрос возвратит уникальные значения для этого выражения.
- Если в `DISTINCT` указано несколько выражений, запрос извлекает уникальные комбинации для перечисленных выражений.
- Если заданы ключевые слова `DISTINCT ON` , запрос возвратит уникальные значения для `distinct_expressions` и вернет другие поля для выбранных записей на основе предложения `ORDER BY (limit 1)` .
- В `PostgreSQL DISTINCT` не игнорирует значения `NULL`. Поэтому при использовании `DISTINCT` в вашем операторе `SQL` ваш результирующий набор будет содержать значение `NULL` как отдельное значение.

?????? ? ????? ???????????

Рассмотрим на простейший пример **DISTINCT** в PostgreSQL. Мы можем использовать оператор `DISTINCT`, чтобы вернуть одно поле, которое удаляет дубликаты из набора результатов.
Например:

```
SELECT DISTINCT last_name
FROM contacts
ORDER BY last_name;
```

В этом PostgreSQL примере **DISTINCT** будут возвращены все уникальные значения `last_name` из таблицы `contacts`.

?????? ? ??????????? ???????????

Давайте посмотрим, как вы можете использовать оператор PostgreSQL `DISTINCT` для удаления дубликатов из более чем одного поля в вашем операторе `SELECT`.
Например:

```
SELECT DISTINCT last_name, city, state
FROM contacts
ORDER BY last_name, city, state;
```

Этот пример будет возвращать каждую уникальную комбинацию `last_name`, `city` и `state` из таблицы `contacts`. В этом случае `DISTINCT` применяется к каждому полю, указанному после ключевого слова `DISTINCT`, и, следовательно, возвращает различные комбинации.

?????? DISTINCT ON

Одна вещь, которая уникальна в PostgreSQL, по сравнению с другими базами данных, заключается в том, что у вас есть еще одна опция при использовании оператора `DISTINCT`, которая называется `DISTINCT ON`.
`DISTINCT ON` вернет только первую строку для `DISTINCT ON (diver_expressions)` на основе оператора `ORDER BY`, предоставленного в запросе. Любые другие поля, перечисленные в операторе `SELECT`, будут возвращены для этой первой строки. Это похоже на выполнение `LIMIT` в 1 для каждой комбинации `DISTINCT ON (different_expressions)`.
Давайте подробнее рассмотрим, как использовать DISTINCT ON в операторе DISTINCT и что он возвращает.
Таким образом, мы могли бы изменить пример выше:

```
SELECT DISTINCT ON (last_name) last_name, city, state
FROM contacts
```

```
ORDER BY last_name, city, state;
```

Этот пример `DISTINCT`, в котором используются ключевые слова `DISTINCT ON`, вернет все уникальные значения `last_name`. Но в этом случае для каждого уникального значения `last_name` он будет возвращать только первую уникальную запись `last_name`, с которой он сталкивается, на основе оператора `ORDER BY` вместе с `city` и `state` значениями из этой записи. Он не возвращает уникальные комбинации `last_name`, `city` и `state`. По сути, он выполняет `LIMIT`, равный 1, для каждого `DISTINCT ON (last_name)` и возвращает соответствующие значения `city` и `state` после того, как он выбрал возвращенные записи.

?????? ????????????? ???????????

- PostgreSQL оператор DISTINCT

?????????? ??????????

Управляющие подключениями

PgBouncer

В PostgreSQL для обработки каждого соединения клиента создается отдельный процесс. Чем больше число соединений, тем больше процессов, которые используют оперативную память. Максимальное число соединений с процессом PostgreSQL определяется параметром `max_connections`.

Чтобы оптимизировать потребление ресурсов, можно использовать управляющего соединениями. Клиенты подключаются не напрямую к PostgreSQL, а к управляющему соединениями. При этом поддерживается небольшое количество соединений между управляющим и сервером PostgreSQL — управляющий создает новое соединение или повторно использует одно из существующих. Количество соединений между управляющим и базой данных на каждой из узлов кластера определяется размером пула (параметр `pool_size`).

?????? ???? (pool_size)?

Размер пула (параметр `pool_size`) — это максимальное число соединений между управляющим соединениями и каждой базой данных PostgreSQL на каждом из узлов кластера.

?????? ???????????

PgBouncer поддерживает три режима подключения:

- режим транзакции (transaction);
- режим сессии (session);
- режим оператора (statement).

????? ?????????? (transaction)?

Соединение с PostgreSQL поддерживается до тех пор, пока не завершится транзакция. Когда транзакция завершается, управляющий соединениями возвращает соединение в пул. Позже это соединение может повторно использоваться этим же клиентом для других подключений или другим клиентом.

Общее количество клиентских подключений к PgBouncer может достигать 10 000, но количество активных транзакций определяет размер пула. Например, если размер пула

равен 30, то активных транзакций будет 30.

Количество соединений между управляющим соединениями и каждой базой данных PostgreSQL на каждом из узлов кластера также определяется размером пула.

Клиент может одновременно выполнять несколько транзакций на разных соединениях. При этом каждое соединение между управляющим соединениями и сервером PostgreSQL в течение своего жизненного цикла может выполнять транзакции разных клиентов.

Режим транзакции снижает нагрузку на ресурсы СУБД, если есть большое количество клиентских подключений с низкой нагрузкой.



??????????? ?????? transaction?

Режим transaction нарушает работу некоторых механизмов PostgreSQL. Выберите другой режим, если клиенты используют эти опции. Некоторые флаги подключения могут распределяться между разными клиентами — это может привести к непредсказуемому поведению и некорректным результатам.

В режиме transaction не работают:

- команды SET/RESET и LISTEN/NOTIFY;
- WITH HOLD CURSOR;
- PRESERVE/DELETE ROWS во временных таблицах;
- подготовленные операторы (prepared statements): protocol-level prepared plans, PREPARE, DEALLOCATE;
- оператор LOAD;
- рекомендательные блокировки Session-level advisory locks.

Подробнее о несовместимых опциях в документации PgBouncer.

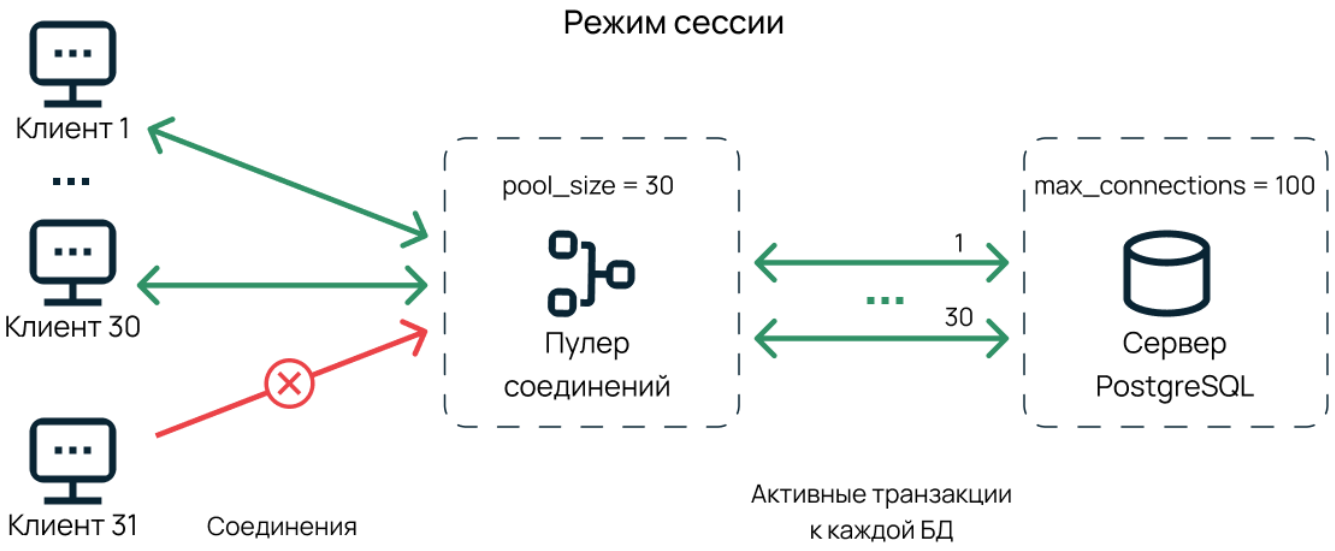
????? ?????? (session)?

В режиме сессии клиент может продолжать отправлять запросы до тех пор, пока продолжается сессия — соединение между управляющим соединениями и сервером PostgreSQL будет поддерживаться до отключения клиента от базы данных.

Количество подключений между управляющим соединениями и сервером PostgreSQL определяется размером пула. На каждое подключение клиента используется подключение между управляющим соединениями и сервером PostgreSQL. Соединение возвращается в пул и может быть повторно использовано только после отключения предыдущего клиента от базы данных.

В отличие от режима транзакции (transaction), этот режим безопасен, повторяет прямое подключение к PostgreSQL, поддерживает все механизмы и подходит для всех клиентов PostgreSQL. При использовании этого режима нагрузка на ресурсы не снижается.

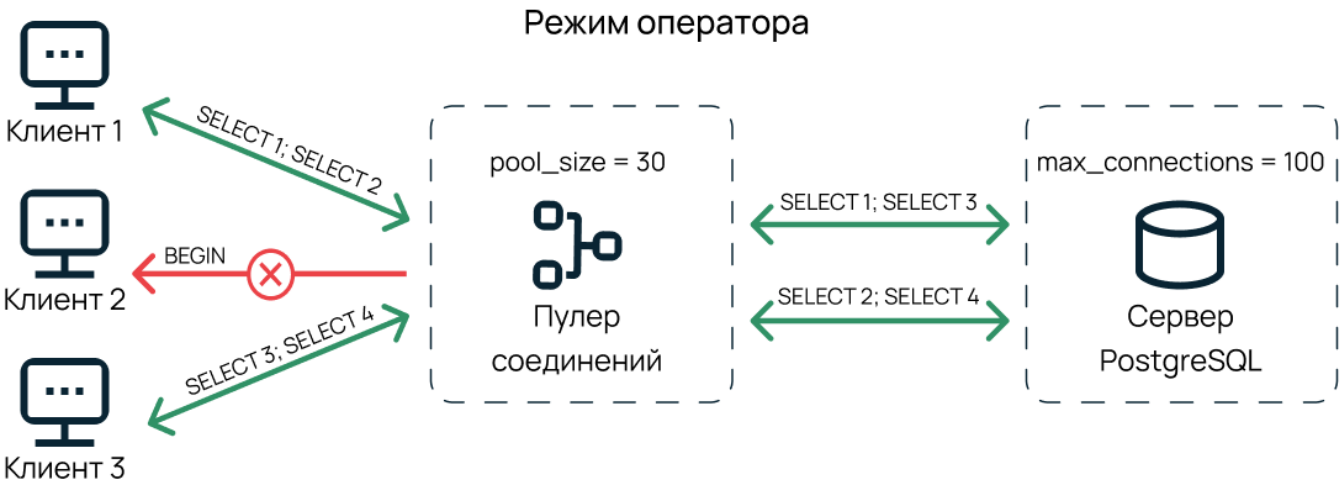
Этот режим соединений полезен для клиентов, у которых много короткоживущих подключений к базе данных, потому что в таком режиме увеличивается скорость подключения к СУБД.



????? ????????? (statement)?

Управляющий подключениями вернет соединение в пул, как только будет обработан первый запрос — транзакции с несколькими операторами прервутся, управляющий подключениями вернет ошибку.

Этот режим позволяет использовать больше клиентских подключений, чем в режиме транзакции. Режим подойдет, если известно, что каждая транзакция ограничена только одним запросом (включен режим AUTOCOMMIT).



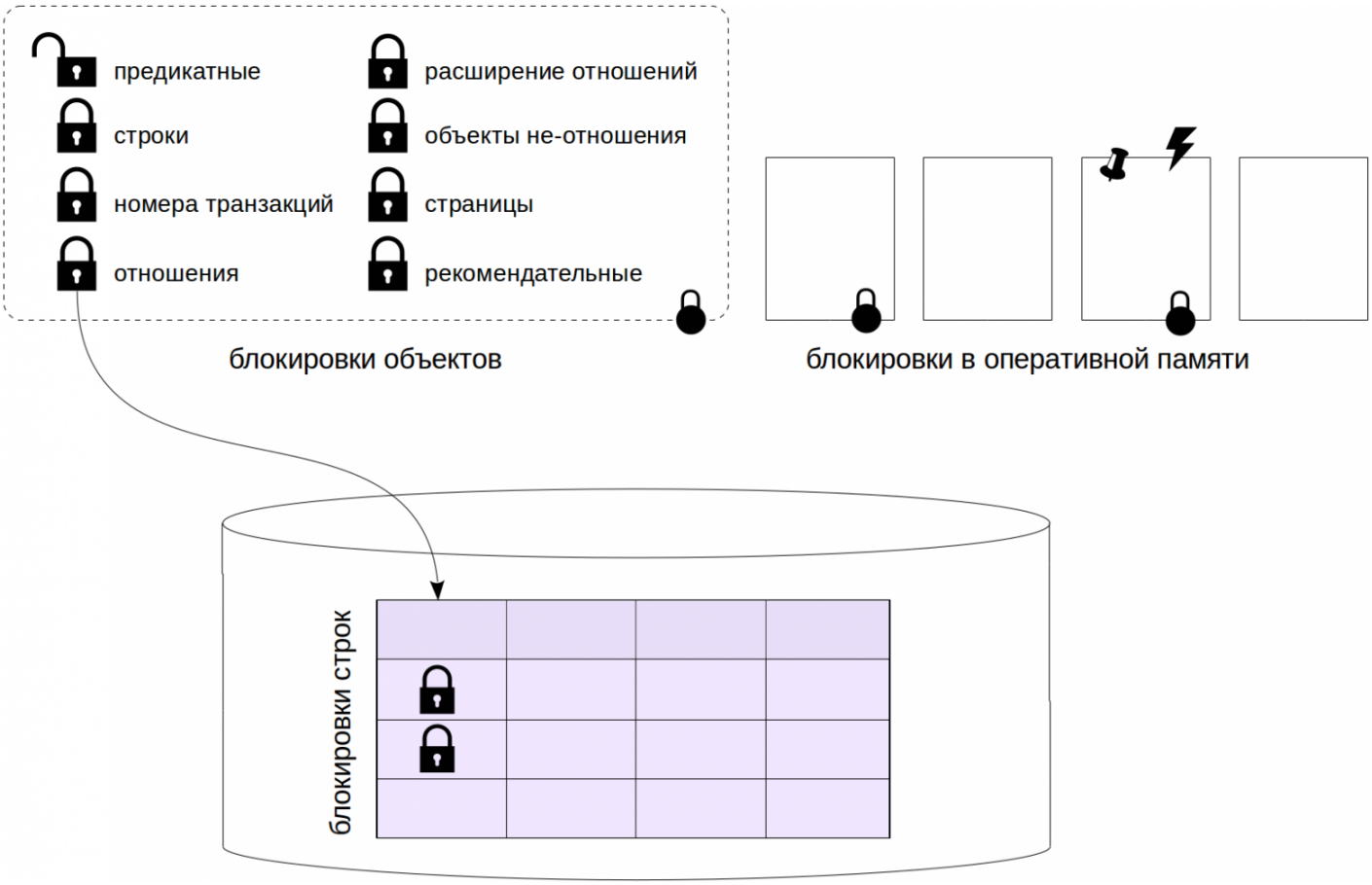
?????? ????????????? ??????????

- Серверные приложения: pgbouncer

??????????

Понимание того как работают блокировки в PostgreSQL является ключом к написанию правильных запросов способных выполняться параллельно.

?????????? ?????????



????? ?????????? ? ??????????

В PostgreSQL используется множество самых разных механизмов, которые служат для блокировки чего-либо (или по крайней мере так называются). Я поэтому начну с самых общих слов о том, зачем вообще нужны блокировки, какие они бывают и чем отличаются друг от друга. Затем мы посмотрим, что из этого разнообразия встречается в PostgreSQL и только после этого начнем разбираться с разными видами блокировок подробно.

Блокировки используются, чтобы упорядочить конкурентный доступ к разделяемым ресурсам.

Под конкурентным доступом понимается одновременный доступ нескольких процессов. Сами процессы могут при этом выполняться как параллельно (если позволяет аппаратура), так и последовательно в режиме разделения времени — это не важно.

Если нет конкуренции, то нет нужды и в блокировках (например, общий буферный кеш требует блокировок, а локальный – нет).

Перед тем, как обратиться к ресурсу, процесс обязан *захватить* (acquire) блокировку, ассоциированную с этим ресурсом. То есть речь идет об определенной дисциплине: все работает до тех пор, пока все процессы выполняют установленные правила обращения к разделяемому ресурсу. Если блокировками управляет СУБД, то она сама следит за порядком; если блокировки устанавливает приложение, то эта обязанность ложится на него.

На низком уровне блокировка представляется участком разделяемой памяти, в котором некоторым образом отмечается, свободна ли блокировка или захвачена (и, возможно, записывается дополнительная информация: номер процесса, время захвата и т. п.).

> Можно заметить, что такой участок разделяемой памяти сам по себе является ресурсом, к которому возможен конкурентный доступ. Если спуститься на уровень ниже, мы увидим, что для упорядочения доступа используются специальные примитивы синхронизации (такие, как семафоры или мьютексы), предоставляемые ОС. Смысл их в том, чтобы код, обращающийся к разделяемому ресурсу, одновременно выполнялся только в одном процессе. На самом низком уровне эти примитивы реализуются на основе атомарных инструкций процессора (таких, как test-and-set или compare-and-swap).

После того, как ресурс больше не нужен процессу, он *освобождает* (release) блокировку, чтобы ресурсом могли воспользоваться другие.

Конечно, захват блокировки возможен не всегда: ресурс может оказаться уже занятым кем-то другим. Тогда процесс либо встает в очередь ожидания (если механизм блокировки дает такую возможность), либо повторяет попытку захвата блокировки через определенное время. Так или иначе это приводит к тому, что процесс вынужден простаивать в ожидании освобождения ресурса.

> Иногда удается применить другие, неблокирующие, стратегии. Например, механизм многоверсионности позволяет нескольким процессам в ряде случаев работать одновременно с разными версиями данных, не блокируя друг друга.

Защищаемым ресурсом в принципе может быть все, что угодно, лишь бы этот ресурс можно

было однозначно идентифицировать и сопоставить ему адрес блокировки.

Например, ресурсом может быть объект, с которым работает СУБД, такой как страница данных (идентифицируется именем файла и позицией внутри файла), таблица (oid в системном каталоге), табличная строка (страница и смещение внутри страницы). Ресурсом может быть структура в памяти, такая как хеш-таблица, буфер и т. п. (идентифицируется заранее присвоенным номером). Иногда даже бывает удобно использовать абстрактные ресурсы, не имеющие никакого физического смысла (идентифицируются просто уникальным числом).

На эффективность блокировок оказывают влияние много факторов, из которых выделим два.

- **Гранулярность** (степень детализации) важна, если ресурсы образуют иерархию.

  Например, таблица состоит из страниц, которые содержат табличные строки. Все эти объекты могут выступать в качестве ресурсов. Если процессы обычно заинтересованы всего в нескольких строках, а блокировка устанавливается на уровне таблицы, то другие процессы не смогут одновременно работать с разными строками. Поэтому чем выше гранулярность, тем лучше для возможности распараллеливания.

  Но это приводит к увеличению числа блокировок (информацию о которых надо хранить в памяти). В таком случае может применяться *повышение уровня* (эскалация) блокировок: когда количество низкоуровневых, высокогранулярных блокировок превышает определенный предел, они заменяются на одну блокировку более высокого уровня.
- Блокировки могут захватываться в разных **режимах**.

  Имена режимов могут быть абсолютно произвольными, важна лишь матрица их совместимости друг с другом. Режим, несовместимый ни с каким режимом (в том числе с самим собой), принято называть *исключительным* или *монопольным* (exclusive). Если режимы совместимы, то блокировка может захватываться несколькими процессами одновременно; такие режимы называют *разделяемыми* (shared). В целом, чем больше можно выделить разных режимов, совместимых друг с другом, тем больше создается возможностей для параллелизма.

По времени использования блокировки можно разделить на длительные и короткие.

- **Долговременные** блокировки захватываются на потенциально большое время (обычно до конца транзакции) и чаще всего относятся к таким ресурсам, как таблицы (отношения) и строки. Как правило, PostgreSQL управляет такими блокировками автоматически, но пользователь, тем не менее, имеет определенный

контроль над этим процессом.

Для длительных блокировок характерно большое число режимов, чтобы можно было выполнять как можно больше одновременных действий над данными. Обычно для таких блокировок имеется развитая инфраструктура (например, поддержка очередей ожидания и обнаружение взаимоблокировок) и средства мониторинга, поскольку затраты на поддержание всех этих удобств все равно оказываются несравнимо меньшими по сравнению со стоимостью операций над защищаемыми данными.

- **Краткосрочные** блокировки захватываются на небольшое время (от нескольких инструкций процессора до долей секунд) и обычно относятся к структурам данных в общей памяти. Такими блокировками PostgreSQL управляет полностью автоматически — об их существовании надо просто знать.

  Для коротких блокировок характерны минимум режимов (исключительный и разделяемый) и простая инфраструктура. В ряде случаев могут отсутствовать даже средства мониторинга.

В PostgreSQL используются разные виды блокировок.

Блокировки на уровне объектов относятся к длительным, «тяжеловесным». В качестве ресурсов здесь выступают отношения и другие объекты. Если слово блокировка встречается в тексте без уточнений, то оно обозначает именно такую, «обычную» блокировку.

Среди длительных блокировок отдельно выделяются **блокировки на уровне строк**. Их реализация отличается от остальных длительных блокировок из-за потенциально огромного их количества (представьте обновление миллиона строк в одной транзакции). Такие блокировки будут рассмотрены в следующей статье.

Третья статья цикла будет посвящена оставшимся блокировкам на уровне объектов, а также **предикатным блокировкам** (поскольку информация о всех этих блокировках хранится в оперативной памяти однотипным образом).

К коротким блокировкам относятся различные **блокировки структур оперативной памяти** . Их мы рассмотрим в последней статье цикла.

?????????? ????????

Итак, мы начинаем с блокировок уровня объектов. Под объектом здесь понимаются в первую очередь *отношения* (relations), то есть таблицы, индексы, последовательности, материализованные представления, но также и некоторые другие сущности. Эти

блокировки обычно защищают объекты от одновременного изменения или от использования в то время, когда объект изменяется, но также и для других нужд.

Расплывчатая формулировка? Так и есть, потому что блокировки из этой группы используются для самых разных целей. Объединяет их лишь то, как они устроены.

??????????

Блокировки объектов располагаются в общей памяти сервера. Их количество ограничено произведением значений двух параметров: *max_locks_per_transaction* × *max_connections*.

Пул блокировок — общий для всех транзакций, то есть одна транзакция может захватить больше блокировок, чем *max_locks_per_transaction*: важно лишь, чтобы общее число блокировок в системе не превысило установленный предел. Пул создается при запуске, так что изменение любого из двух указанных параметров требует перезагрузки сервера.

Все блокировки можно посмотреть в представлении pg_locks.

Если ресурс уже заблокирован в несовместимом режиме, транзакция, пытающаяся захватить этот ресурс, ставится в очередь и ожидает освобождения блокировки. Ожидающие транзакции не потребляют ресурсы процессора: соответствующие обслуживающие процессы «засыпают» и пробуждаются операционной системой при освобождении ресурса.

Возможна ситуация *взаимоблокировки* или *тупика* (deadlock), при которой одной транзакции для продолжения работы требуется ресурс, занятый второй транзакцией, а второй необходим ресурс, занятый первой (в общем случае может произойти взаимоблокировка и более двух транзакций). В таком случае ожидание будет продолжаться бесконечно, поэтому PostgreSQL автоматически определяет такие ситуации и аварийно прерывает одну из транзакций, чтобы остальные могли продолжить работу. (Более подробно мы поговорим про взаимоблокировки в следующей статье.)

???? ????????

Вот список типов блокировок (или, если угодно, типов объектов), с которыми мы будем иметь дело в этой и следующей статьях. Названия приведены в соответствии со столбцом locktype представления pg_locks.

- **relation**

Блокировки отношений.

- **transactionid** и **virtualxid**

  Блокировка номера транзакции (настоящего или виртуального). Каждая транзакция сама удерживает исключительную блокировку своего собственного номера, поэтому такие блокировки удобно использовать, когда нужно дождаться окончания другой транзакции.
- **tuple**

  Блокировка версии строки. Используется в некоторых случаях для установки приоритета среди нескольких транзакций, ожидающих блокировку одной и той же строки.

Разговор об остальных типах блокировок мы отложим до третьей статьи цикла. Все они захватываются либо только в исключительном режиме, либо в исключительном и разделяемом.

- **extend**

  Используется при добавлении страниц к файлу какого-либо отношения.
- **object**

  Блокировка объектов, которые не являются отношениями (баз данных, схем, подписок и т. п.).
- **page**

  Блокировка страницы, используется нечасто и только некоторыми типами индексов.
- **advisory**

  Рекомендательная блокировка, устанавливается пользователем вручную.

?????????? ??????????

Чтобы не терять контекст, я буду отмечать на такой картинке те типы блокировок, о которых дальше пойдет речь.

odwcnenfvjv_ikwmqw8x-vaexqi.png

??????

Если не самая важная, то уж точно самая «развесистая» блокировка — блокировка отношений. Для нее определено целых 8 различных режимов. Такое количество нужно для того, чтобы как можно большее количество команд, относящихся к одной таблице, могло выполняться одновременно.

Нет никакого смысла учить эти режимы наизусть или пытаться вникнуть в смысл их названий; главное — иметь в нужный момент перед глазами матрицу, которая показывает, какие блокировки конфликтуют друг с другом. Для удобства она воспроизведена здесь вместе с примерами команд, которые требуют соответствующие уровни блокировки:

режим блокировки	AS	RS	RE	SUE	S	SRE	E	AE	пример SQL-команд
Access Share								X	SELECT
Row Share							X	X	SELECT FOR UPDATE/ SHARE
Row Exclusive					X	X	X	X	INSERT, UPDATE, DELETE
Share Update Exclusive				X	X	X	X	X	VACUUM, ALTER TABLE*, CREATE INDEX CONCURRENTLY
Share			X	X		X	X	X	CREATE INDEX
Share Row Exclusive			X	X	X	X	X	X	CREATE TRIGGER, ALTER TABLE*
Exclusive		X	X	X	X	X	X	X	REFRESH MAT. VIEW CONCURRENTLY

Access Exclusive	X	X	X	X	X	X	X	X	DROP, TRUNCATE, VACUUM FULL, LOCK TABLE, ALTER TABLE*, REFRESH MAT. VIEW

Некоторые комментарии:

- Первые 4 режима допускают одновременное изменение данных в таблице, а следующие 4 — нет.
- Первый режим (Access Share) — самый слабый, он совместим с любым другим, кроме последнего (Access Exclusive). Этот последний режим — монопольный, он не совместим ни с одним режимом.
- Команда ALTER TABLE имеет много вариантов, разные из которых требуют разных уровней блокировки. Поэтому в матрице эта команда появляется в разных строках и отмечена звездочкой.

??? ??????? ????????

приведем такой пример. Что произойдет, если выполнить команду CREATE INDEX?

Находим в документации, что эта команда устанавливает блокировку в режиме Share. По матрице определяем, что команда совместима сама с собой (то есть можно одновременно создавать несколько индексов) и с читающими командами. Таким образом, команды SELECT продолжат работу, а вот команды UPDATE, DELETE, INSERT будут заблокированы.

И наоборот — незавершенные транзакции, изменяющие данные в таблице, будут блокировать работу команды CREATE INDEX. Поэтому и существует вариант команды — CREATE INDEX CONCURRENTLY. Он работает дольше (и может даже упасть с ошибкой), зато допускает одновременное изменение данных.

В сказанном можно убедиться и на практике. Для экспериментов мы будем использовать знакомую по первому циклу таблицу «банковских» счетов, в которой будем хранить номер счета и сумму.

```
=> CREATE TABLE accounts(
  acc_no integer PRIMARY KEY,
  amount numeric
);
=> INSERT INTO accounts
  VALUES (1,1000.00), (2,2000.00), (3,3000.00);
```

Во втором сеансе начнем транзакцию. Нам понадобится номер обслуживающего процесса.

```
|  => SELECT pg_backend_pid();
```

```
|   pg_backend_pid
|  ----------------
|              4746
|  (1 row)
```

Какие блокировки удерживает только что начавшаяся транзакция? Смотрим в pg_locks:

```
=> SELECT locktype, relation::REGCLASS, virtualxid AS virtxid, transactionid AS xid, mode,
granted
```

```
FROM pg_locks WHERE pid = 4746;
```

```
  locktype  | relation | virtxid | xid |     mode      | granted
------------+----------+---------+-----+---------------+---------
 virtualxid |          | 5/15    |     | ExclusiveLock | t
(1 row)
```

Как я уже говорил, транзакция всегда удерживает исключительную (ExclusiveLock) блокировку собственного номера, а данном случае — виртуального. Других блокировок у этого процесса нет.

Теперь обновим строку таблицы. Как изменится ситуация?

```
|  => UPDATE accounts SET amount = amount + 100 WHERE acc_no = 1;
```

```
=> \g
```

```
   locktype    |   relation    | virtxid |  xid   |       mode       | granted
---------------+---------------+---------+--------+------------------+---------
 relation      | accounts_pkey |         |        | RowExclusiveLock | t
 relation      | accounts      |         |        | RowExclusiveLock | t
 virtualxid    |               | 5/15    |        | ExclusiveLock    | t
 transactionid |               |         | 529404 | ExclusiveLock    | t
(4 rows)
```

Теперь появились блокировки изменяемой таблицы и индекса (созданного для первичного ключа), который используется командой UPDATE. Обе блокировки взяты в режиме RowExclusiveLock. Кроме того, добавилась исключительная блокировка настоящего номера транзакции (который появился, как только транзакция начала изменять данные).

Теперь попробуем в еще одном сеансе создать индекс по таблице.

```
||     => SELECT pg_backend_pid();
```

```
||      pg_backend_pid
||     ----------------
```

```
||                4782
||     (1 row)
```

```
||     => CREATE INDEX ON accounts(acc_no);
```

Команда «подвисает» в ожидании освобождения ресурса. Какую именно блокировку она пытается захватить? Проверим:

```
=> SELECT locktype, relation::REGCLASS, virtualxid AS virtxid, transactionid AS xid, mode,
granted
FROM pg_locks WHERE pid = 4782;
```

```
  locktype  | relation | virtxid | xid |     mode      | granted
------------+----------+---------+-----+---------------+---------
 virtualxid |          | 6/15    |     | ExclusiveLock | t
 relation   | accounts |         |     | ShareLock     | f
(2 rows)
```

Видим, что транзакция пытается получить блокировку таблицы в режиме ShareLock, но не может (granted = f).

Находить номер блокирующего процесса, а в общем случае — несколько номеров, удобно с помощью функции, которая появилась в версии 9.6 (до того приходилось делать выводы, вдумчиво разглядывая все содержимое pg_locks):

```
=> SELECT pg_blocking_pids(4782);
```

```
 pg_blocking_pids
------------------
 {4746}
(1 row)
```

И затем, чтобы разобраться в ситуации, можно получить информацию о сеансах, к которым относятся найденные номера:

```
=> SELECT * FROM pg_stat_activity
WHERE pid = ANY(pg_blocking_pids(4782)) \gx
```

```
-[ RECORD 1 ]----+------------------------------------------------------------
datid            | 16386
datname          | test
pid              | 4746
usesysid         | 16384
usename          | student
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2019-08-07 15:02:53.811842+03
xact_start       | 2019-08-07 15:02:54.090672+03
query_start      | 2019-08-07 15:02:54.10621+03
state_change     | 2019-08-07 15:02:54.106965+03
wait_event_type  | Client
wait_event       | ClientRead
state            | idle in transaction
backend_xid      | 529404
backend_xmin     |
query            | UPDATE accounts SET amount = amount + 100 WHERE acc_no = 1;
backend_type     | client backend
```

После завершения транзакции блокировки снимаются и индекс создается.

```
|  => COMMIT;
```

```
|  COMMIT
```

```
||     CREATE INDEX
```

? ???????!..

Чтобы лучше представить, к чему приводит появление несовместимой блокировки, посмотрим, что произойдет, если в процессе работы системы выполнить команду VACUUM FULL.

Пусть вначале на нашей таблице выполняется команда SELECT. Она получает блокировку самого слабого уровня Access Share. Чтобы управлять временем освобождения блокировки, мы выполняем эту команду внутри транзакции — пока транзакция не окончится, блокировка не будет снята. В реальности таблицу могут читать (и изменять) несколько команд, и какие-то из запросов могут выполняться достаточно долго.

```
=> BEGIN;
=> SELECT * FROM accounts;
```

```
 acc_no | amount
--------+---------
      2 | 2000.00
      3 | 3000.00
      1 | 1100.00
(3 rows)
```

```
=> SELECT locktype, mode, granted, pid, pg_blocking_pids(pid) AS wait_for
FROM pg_locks WHERE relation = 'accounts'::regclass;
```

```
 locktype |      mode       | granted | pid  | wait_for
----------+-----------------+---------+------+----------
 relation | AccessShareLock | t       | 4710 | {}
(1 row)
```

Затем администратор выполняет команду VACUUM FULL, которой требуется блокировка уровня Access Exclusive, несовместимая ни с чем, даже с Access Share. (Такую же блокировку требует и команда LOCK TABLE.) Транзакция встает в очередь.

```
|  => BEGIN;
|  => LOCK TABLE accounts; -- аналогично VACUUM FULL
```

```
=> SELECT locktype, mode, granted, pid, pg_blocking_pids(pid) AS wait_for
FROM pg_locks WHERE relation = 'accounts'::regclass;
```

```
 locktype |        mode         | granted | pid  | wait_for
----------+---------------------+---------+------+----------
 relation | AccessShareLock     | t       | 4710 | {}
 relation | AccessExclusiveLock | f       | 4746 | {4710}
(2 rows)
```

Но приложение продолжает выдавать запросы, и вот в системе появляется еще команда SELECT. Чисто теоретически она могла бы «проскочить», пока VACUUM FULL ждет, но нет — она честно занимают место в очереди за VACUUM FULL.

```
||     => SELECT * FROM accounts;
```

```
=> SELECT locktype, mode, granted, pid, pg_blocking_pids(pid) AS wait_for
FROM pg_locks WHERE relation = 'accounts'::regclass;
```

```
 locktype |        mode         | granted | pid  | wait_for
----------+---------------------+---------+------+----------
 relation | AccessShareLock     | t       | 4710 | {}
 relation | AccessExclusiveLock | f       | 4746 | {4710}
 relation | AccessShareLock     | f       | 4782 | {4746}
(3 rows)
```

После того, как первая транзакция с командой SELECT завершается и освобождает блокировку, начинает выполняться команда VACUUM FULL (которую мы сымитировали командой LOCK TABLE).

```
=> COMMIT;
```

```
COMMIT
```

```
|  LOCK TABLE
```

```
=> SELECT locktype, mode, granted, pid, pg_blocking_pids(pid) AS wait_for
FROM pg_locks WHERE relation = 'accounts'::regclass;
```

```
 locktype |        mode         | granted | pid  | wait_for
----------+---------------------+---------+------+----------
 relation | AccessExclusiveLock | t       | 4746 | {}
 relation | AccessShareLock     | f       | 4782 | {4746}
(2 rows)
```

И только после того, как VACUUM FULL завершит свою работу и снимет блокировку, все накопившиеся в очереди команды (SELECT в нашем примере) смогут захватить соответствующие блокировки (Access Share) и выполниться.

```
|  => COMMIT;
```

```
|  COMMIT
```

```
||      acc_no | amount
||     --------+---------
||           2 | 2000.00
||           3 | 3000.00
||           1 | 1100.00
||     (3 rows)
```

Таким образом, неаккуратно выполненная команда может парализовать работу системы на время, значительно превышающее время выполнение самой команды.

???????? ????????????

Безусловно, блокировки необходимы для корректной работы, но могут приводить к нежелательным ожиданиям. Такие ожидания можно отслеживать, чтобы разобраться в их причине и по возможности устранить (например, изменив алгоритм работы приложения).

С одним способом мы уже познакомились: в момент возникновения долгой блокировки мы можем выполнить запрос к представлению pg_locks, посмотреть на блокируемые и блокирующие транзакции (функция pg_blocking_pids) и расшифровывать их при помощи pg_stat_activity.

Другой способ состоит в том, чтобы включить параметр *log_lock_waits*. В этом случае в журнал сообщений сервера будет попадать информация, если транзакция ждала дольше, чем *deadlock_timeout* (несмотря на то, что используется параметр для взаимоблокировок, речь идет об обычных ожиданиях).

Попробуем.

```
=> ALTER SYSTEM SET log_lock_waits = on;
=> SELECT pg_reload_conf();
```

Значение параметра *deadlock_timeout* по умолчанию равно одной секунде:

```
=> SHOW deadlock_timeout;
```

```
 deadlock_timeout
------------------
 1s
(1 row)
```

Воспроизведем блокировку.

```
=> BEGIN;
=> UPDATE accounts SET amount = amount - 100.00 WHERE acc_no = 1;
```

```
UPDATE 1
```

```
|  => BEGIN;
|  => UPDATE accounts SET amount = amount + 100.00 WHERE acc_no = 1;
```

Вторая команда UPDATE ожидает блокировку. Подождем секунду и завершим первую транзакцию.

```
=> SELECT pg_sleep(1);
=> COMMIT;
```

```
COMMIT
```

Теперь и вторая транзакция может завершиться.

```
|  UPDATE 1
```

```
|  => COMMIT;
```

```
|  COMMIT
```

И вся важная информация попала в журнал:

```
postgres$ tail -n 7 /var/log/postgresql/postgresql-11-main.log
```

```
2019-08-07 15:26:30.827 MSK [5898] student@test LOG:  process 5898 still waiting for ShareLock on transaction 529427 after 1000.186 ms
2019-08-07 15:26:30.827 MSK [5898] student@test DETAIL:  Process holding the lock: 5862. Wait queue: 5898.
2019-08-07 15:26:30.827 MSK [5898] student@test CONTEXT:  while updating tuple (0,4) in relation "accounts"
2019-08-07 15:26:30.827 MSK [5898] student@test STATEMENT:  UPDATE accounts SET amount = amount + 100.00 WHERE acc_no = 1;
```

```
2019-08-07 15:26:30.836 MSK [5898] student@test LOG:  process 5898 acquired ShareLock on transaction 529427 after 1009.536 ms
```

```
2019-08-07 15:26:30.836 MSK [5898] student@test CONTEXT:  while updating tuple (0,4) in relation "accounts"
2019-08-07 15:26:30.836 MSK [5898] student@test STATEMENT:  UPDATE accounts SET amount = amount + 100.00 WHERE acc_no = 1;
```

?????? ????????????? ???????????

- Блокировки в PostgreSQL: 1. Блокировки отношений

?????????? ??????????
SQL-????????

?????????????? ??????????? SQL-????????? ? PostgreSQL ? ?????????????? LLVM JIT