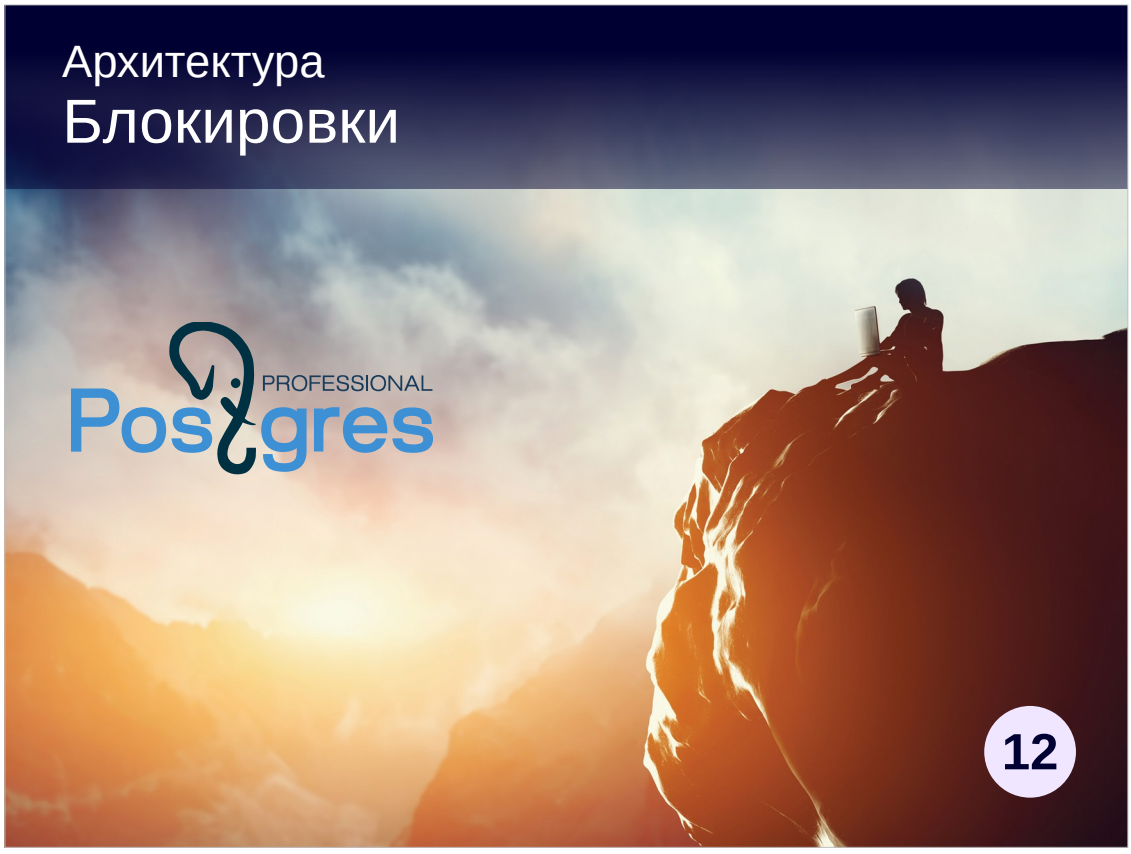


Авторы: Егор Рогов, Павел Лузанов

Обратная связь
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

Темы

Общая информация о блокировках

Блокировки отношений и других объектов

Блокировки на уровне строк

Блокировки

Задача: упорядочение конкурентного доступа к разделяемым ресурсам

Механизм

- перед обращением к данным процесс захватывает блокировку
- после обращения — освобождает (обычно в конце транзакции)
- несовместимые блокировки приводят к очередям

Блокировки используются, чтобы упорядочить конкурентный доступ к разделяемым ресурсам.

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

Блокировки не нужны, если нет конкуренции (одновременно к данным обращается только один процесс) или если нет разделяемого ресурса (например, общий буферный кеш нуждается в блокировках, а локальный — нет).

Перед тем, как обратиться к ресурсу, защищенному блокировкой, процесс должен захватить эту блокировку. После того, как ресурс больше не нужен процессу, он освобождает блокировку, чтобы ресурсом могли воспользоваться другие процессы.

Захват блокировки возможен не всегда: ресурс может оказаться уже занятым кем-то другим в несовместимом режиме. В простом случае используют два режима: исключительный (несовместим ни с чем) и разделяемый (совместим сам с собой). Но их может быть и больше, в этом случае совместимость определяется матрицей.

Если ресурс занят, процесс должен встать в очередь ожидания. Это, конечно, уменьшает производительность системы. Поэтому для создания высокоэффективных приложений важно понимать механизм блокировок.

Блокировки объектов

Блокировки номера транзакции

Блокировки отношений

Очередь ожидания

Из всего многообразия блокировок на уровне объектов мы рассмотрим только блокировки номера транзакции и блокировки отношений.

Еще один вид блокировок того же типа – рекомендательные блокировки – рассматривается в курсе DEV1.

Номер транзакции

Типы ресурсов в pg_locks

- virtualxid — виртуальный номер транзакции
- transactionid — номер транзакции

Режимы

- исключительный
- разделяемый

Транзакции удерживают исключительную блокировку собственного номера

- способ дождаться завершения транзакции

Рассмотрение блокировок на уровне объектов («обычные», «тяжелые» блокировки) мы начнем с блокировок номера транзакции.

Блокировки объектов располагаются в общей памяти сервера, поэтому их общее количество ограничено. Все такие блокировки можно посмотреть в представлении pg_locks: они устроены одинаково и отличаются только типом ресурса и режимами блокирования. Блокировки номера транзакции отображаются в pg_locks с типами ресурса **Transactionid** и **Virtualxid**.

Каждой транзакции при старте назначается сначала виртуальный номер, который используется, пока транзакция только читает данные. Это сделано для оптимизации. Виртуальный номер никак не учитывается в правилах видимости. Кроме того, виртуальный номер не обязан быть полностью уникальным (аналогично номеру процесса в Unix) и его можно выдать быстро, без обращения к общей памяти.

Настоящий номер присваивается в тот момент, когда транзакция изменяет какие-либо данные. Реальный уникальный номер нужен, чтобы отслеживать статус транзакции (зафиксирована или оборвана); такой номер можно записывать на диск в полях заголовка версий строк.

Каждая транзакция всегда сама удерживает исключительную блокировку своего собственного номера (и виртуального, и — если есть — реального). Это дает простой способ дождаться окончания какой-либо транзакции: надо запросить блокировку ее номера. Блокировку получить не удастся, и процесс будет разбужен только когда блокировка освободится, а произойдет это при завершении транзакции.

Блокировка номера транзакции

```
=> CREATE DATABASE arch_locks;
```

```
CREATE DATABASE
```

```
=> \c arch_locks
```

```
You are now connected to database "arch_locks" as user "student".
```

Начнем в другом сеансе новую транзакцию.

```
=> \c arch_locks
```

```
You are now connected to database "arch_locks" as user "student".
```

```
=> BEGIN;
```

```
BEGIN
```

Нам понадобится номер обслуживающего процесса:

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

```
 pg_backend_pid
----------------
          62824
(1 row)
```

Все «обычные» блокировки видны в представлении pg_locks. Какие блокировки удерживает только что начатая транзакция?

```
=> SELECT locktype, virtualxid AS virtxid, transactionid AS xid,
    mode, granted
FROM pg_locks
WHERE pid = 62824;
```

```
  locktype  | virtxid | xid |     mode      | granted
------------+---------+-----+---------------+---------
 virtualxid | 3/26    |     | ExclusiveLock | t
(1 row)
```

- locktype — тип ресурса,
- mode — режим блокировки,
- granted — удалось ли получить блокировку.

Каждой транзакции выдается виртуальный номер, и транзакция удерживает его исключительную блокировку.

Как только транзакция начинает менять какие-либо данные, ей выдается настоящий номер, который учитывается в правилах видимости. Номер можно получить и явно:

```
=> SELECT txid_current();
```

```
 txid_current
--------------
          540
(1 row)
```

```
=> SELECT locktype, virtualxid AS virtxid, transactionid AS xid,
    mode, granted
FROM pg_locks
WHERE pid = 62824;
```

```
   locktype    | virtxid | xid |     mode      | granted
---------------+---------+-----+---------------+---------
 virtualxid    | 3/26    |     | ExclusiveLock | t
 transactionid |         | 540 | ExclusiveLock | t
(2 rows)
```

Теперь транзакция удерживает исключительную блокировку обоих номеров.

Блокировки отношений

Тип ресурса в pg_locks

relation — таблицы, индексы и т. п.

Режимы

Access Share	SELECT	допускают параллельное изменение данных в таблице
Row Share	SELECT FOR UPDATE/SHARE	
Row Exclusive	UPDATE, DELETE, INSERT	
Share Update Exclusive	VACUUM, ALTER TABLE, CREATE INDEX CONCURRENTLY	
Share	CREATE INDEX	
Share Row Exclusive	CREATE TRIGGER, ALTER TABLE	
Exclusive	REFRESH MAT. VIEW CONCURRENTLY	
Access Exclusive	DROP, TRUNCATE, VACUUM FULL, LOCK TABLE, ALTER TABLE, REFRESH MAT. VIEW	

Второй важный случай блокировок объектов — блокировки отношений (таблиц, индексов, последовательностей и т. п.). Такие блокировки имеют тип **Relation** в pg_locks.

Для них определено целых 8 различных режимов, которые показаны на слайде вместе с примерами команд SQL, которые используют эти режимы. Матрица совместимости здесь не показана из-за ее большого размера, но она приведена в документации:

https://postgrespro.ru/docs/postgresql/12/explicit-locking#LOCKING-TABLES

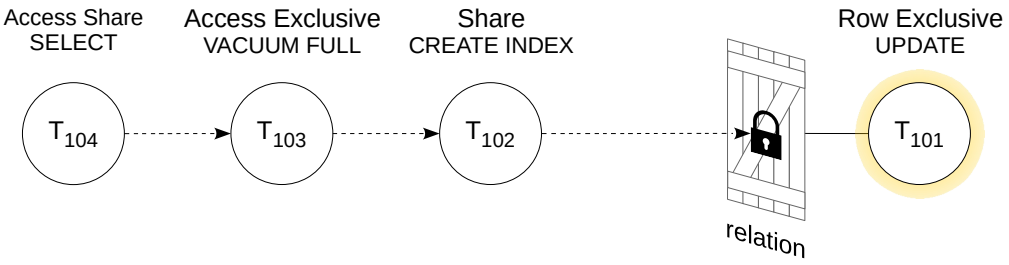
Такое количество режимов существуют для того, чтобы позволить выполнять одновременно как можно большее количество команд, относящихся к одной таблице (индексу и т. п.).

Самый слабый режим — Access Share, он захватывается командой SELECT и совместим с любым режимом, кроме самого сильного — Access Exclusive. Это означает, что запрос не мешает ни другим запросам, ни изменению данных в таблице, ни чему-либо другому, но не дает, например, удалить таблицу в то время, как из нее читаются данные.

Другой пример: режим Share (как и другие более сильные режимы) не совместимы с изменением данных в таблице. Например, команда CREATE INDEX заблокирует команды INSERT, UPDATE и DELETE (и наоборот). Поэтому есть команда CREATE INDEX CONCURRENTLY, использующая режим Share Update Exclusive, который совместим с такими изменениями (за счет этого команда выполняется дольше).

Очередь ожидания

Честная очередь для несовместимых режимов



Блокировки объектов предоставляют «честную» очередь ожидания. Это означает, что операции с несовместимыми режимами выстраиваются в очередь и вне очереди никто не проходит.

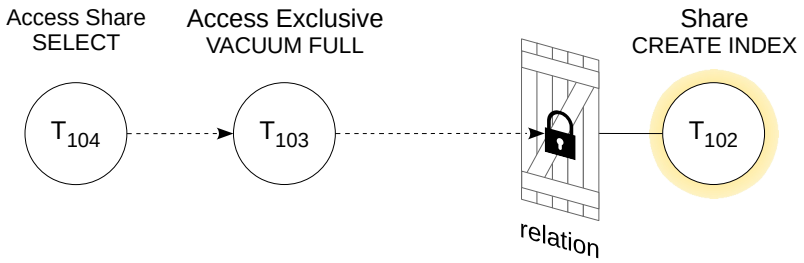
Например, на рисунке выполняется команда UPDATE. Операция CREATE INDEX встает в очередь, поскольку использует несовместимый режим. Если следом придет команда VACUUM FULL (использующая самый сильный режим, не совместимый ни с чем), она встанет в очередь за CREATE INDEX.

Обычный запрос SELECT, хоть и совместим с выполняющейся сейчас командой UPDATE, тоже честно встанет в очередь за VACUUM FULL.

На рисунке транзакции показаны кружками, сплошные стрелки обозначают захваченную блокировку, а пунктирные стрелки — попытки захватить блокировку, уже занятую в несовместимом режиме.

Очередь ожидания

Честная очередь для несовместимых режимов



После того, как первая транзакция завершается, блокировку захватывает следующая транзакция, стоящая в очереди.

Блокировка отношений

Создадим таблицу

```
=> CREATE TABLE accounts(acc_no integer, amount numeric);
```

```
CREATE TABLE
```

```
=> INSERT INTO accounts VALUES (1,100.00), (2,200.00), (3,300.00);
```

```
INSERT 0 3
```

Вторая транзакция, которую мы не завершали, продолжает работу. Выполним в ней запрос к таблице:

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

```
 acc_no | amount
--------+--------
      1 | 100.00
      2 | 200.00
      3 | 300.00
(3 rows)
```

Как изменятся блокировки?

```
=> SELECT locktype, relation::regclass, virtualxid AS virtxid,
    transactionid AS xid, mode, granted
FROM pg_locks
WHERE pid = 62824;
```

```
   locktype    | relation | virtxid | xid |      mode       | granted
---------------+----------+---------+-----+-----------------+---------
 relation      | accounts |         |     | AccessShareLock | t
 virtualxid    |          | 3/26    |     | ExclusiveLock   | t
 transactionid |          |         | 540 | ExclusiveLock   | t
(3 rows)
```

Добавилась блокировка таблицы в режиме Access Share. Она совместима с блокировкой любого режима, кроме Access Exclusive, поэтому не мешает практически никаким операциям, но не дает, например, удалить таблицу. Попробуем.

```
    => \c arch_locks
```

```
    You are now connected to database "arch_locks" as user "student".
```

```
    => BEGIN;
```

```
    BEGIN
```

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

```
     pg_backend_pid
    ----------------
              63155
    (1 row)
```

```
    => DROP TABLE accounts;
```

Команда не выполняется — ждет освобождения блокировки. Какой?

```
=> SELECT locktype, relation::regclass, virtualxid AS virtxid,
    transactionid AS xid, mode, granted
FROM pg_locks
WHERE pid = 63155;
```

```
   locktype    | relation | virtxid | xid |        mode         | granted
---------------+----------+---------+-----+---------------------+---------
 virtualxid    |          | 4/9     |     | ExclusiveLock       | t
 transactionid |          |         | 543 | ExclusiveLock       | t
 relation      | accounts |         |     | AccessExclusiveLock | f
(3 rows)
```

Транзакция пыталась получить блокировку таблицы в режиме Access Exclusive, но не смогла (granted = f).

Информацию о том, что транзакция ожидает чего-то для продолжения работы, можно получить и так:

```
=> SELECT state, wait_event_type, wait_event
FROM pg_stat_activity
WHERE pid = 63155;
```

```
 state  | wait_event_type | wait_event
--------+-----------------+------------
 active | Lock            | relation
(1 row)
```

- state — состояние: активная транзакция, выполняющая команду;
- wait_event_type — тип ожидания: блокировка (бывают и другие, например, ожидание ввода-вывода);
- wait_event — конкретное ожидание: ожидание блокировки отношения.

Мы можем найти номер блокирующего процесса (в общем виде — несколько номеров):

```
=> SELECT pg_blocking_pids(63155);
```

```
 pg_blocking_pids
------------------
 {62824}
(1 row)
```

И посмотреть информацию о сеансах, к которым они относятся:

```
=> SELECT * FROM pg_stat_activity
WHERE pid = ANY(pg_blocking_pids(63155)) \gx
```

```
-[ RECORD 1 ]----+------------------------------
datid            | 16466
datname          | arch_locks
pid              | 62824
usesysid         | 16384
usename          | student
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2023-07-26 10:58:57.890441+03
xact_start       | 2023-07-26 10:58:57.959182+03
query_start      | 2023-07-26 10:58:58.216269+03
state_change     | 2023-07-26 10:58:58.216535+03
wait_event_type  | Client
wait_event       | ClientRead
state            | idle in transaction
backend_xid      | 540
backend_xmin     |
query            | SELECT * FROM accounts;
backend_type     | client backend
```

После завершения транзакции все блокировки снимаются и таблица удаляется:

```
=> COMMIT;
```

```
COMMIT
```

```
DROP TABLE
```

```
=> COMMIT;
```

```
COMMIT
```

Блокировки строк

Разделяемые и исключительные блокировки

«Нечестная» очередь ожидания

Блокировки строк

Используются совместно с многоверсионностью, для чтения данных не требуются

Информация *только* в страницах данных

- в оперативной памяти ничего не хранится

Неограниченное количество

- большое число не влияет на производительность

Инфраструктура

- очередь ожидания организована с помощью блокировок объектов

Благодаря многоверсионности, блокировки уровня строк нужны только при изменении данных (или для того, чтобы не допустить изменения), но не нужны при обычном чтении.

Если в случае блокировок объектов каждый ресурс представлен собственной блокировкой в оперативной памяти, то со строками так не получается: отдельная блокировка для каждой табличной строки (которых могут быть миллионы и миллиарды) потребует непомерных накладных расходов и огромного объема оперативной памяти.

Один известный вариант решения состоит в повышении уровня: если уже заблокировано N строк таблицы, то при блокировке еще одной блокируется вся таблица целиком, а блокировки на уровне строк снимаются. Но в этом случае страдает пропускная способность.

Поэтому в PostgreSQL сделано иначе. Информация о том, что строка заблокирована, хранится только и исключительно в версии строки внутри страницы данных. Там она представлена номером блокирующей транзакции (xmax) и дополнительными информационными битами.

За счет этого может быть неограниченное количество блокировок уровня строки. Это не приводит к потреблению каких-либо ресурсов и не снижает производительность системы.

Обратная сторона такого подхода — сложность организации очереди ожидания. Для этого все-таки приходится использовать блокировки уровня объектов, но удается обойтись очень небольшим их количеством (пропорциональным числу процессов, а не числу заблокированных строк).

Блокировки строк

Режимы

Update — удаление строки или изменение всех полей
No Key Update — изменение любых полей, кроме ключевых
Share — запрет изменения любых полей строки
Key Share — запрет изменения ключевых полей строки

	Key Share	Share	No Key Update	Update	
Key Share				×	разделяемые
Share			×	×	
No Key Update		×	×	×	исключительные
Update	×	×	×	×	

Всего существует 4 режима, в которых можно заблокировать строку (в версии строки режим проставляется с помощью дополнительных информационных битов).

Два режима представляют **исключительные** (exclusive) блокировки, которые одновременно может удерживать только одна транзакция. Режим UPDATE предполагает полное изменение (или удаление) строки, а режим NO KEY UPDATE — изменение только тех полей, которые не входят в уникальные индексы (иными словами, при таком изменении все внешние ключи остаются без изменений). Команда UPDATE сама выбирает минимальный подходящий режим блокировки; обычно строки блокируются в режиме NO KEY UPDATE.

Еще два режима представляют **разделяемые** (shared) блокировки, которые могут удерживаться несколькими транзакциями.

Режим SHARE применяется, когда нужно прочитать строку, но при этом нельзя допустить, чтобы она как-либо изменилась другой транзакцией. Режим KEY SHARE допускает изменение строки, но только неключевых полей. Этот режим, в частности, автоматически используется PostgreSQL при проверке внешних ключей.

Напомним, что обычное чтение (SELECT) вообще не блокирует строки.

Общая матрица совместимости режимов приведена внизу слайда. Из нее видно, что разделяемый режим KEY SHARE совместим с исключительным режимом NO KEY UPDATE (то есть можно обновлять неключевые поля и быть уверенным в том, что ключ не изменится).

https://postgrespro.ru/docs/postgresql/12/explicit-locking#LOCKING-ROWS

Блокировка строк

Снова создадим таблицу счетов, и сделаем номер счета первичным ключом:

```
=> CREATE TABLE accounts(acc_no integer PRIMARY KEY, amount numeric);

CREATE TABLE

=> INSERT INTO accounts VALUES (1,100.00), (2,200.00), (3,300.00);

INSERT 0 3
```

В новой транзакции обновим сумму первого счета (при этом ключ не меняется):

```
=> BEGIN;

BEGIN

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

UPDATE 1

=> SELECT txid_current();

 txid_current
--------------
          546
(1 row)
```

Как правило, признаком блокировки строки служит номер блокирующей транзакции, записанный в поле xmax (и еще ряд информационных битов, определяющих режим блокировки):

```
=> SELECT xmax, * FROM accounts;

 xmax | acc_no | amount
------+--------+--------
  546 |      1 | 100.00
    0 |      2 | 200.00
    0 |      3 | 300.00
(3 rows)
```

Но в случае разделяемых блокировок такой способ не годится, поскольку в xmax нельзя записать несколько номеров транзакций. Чтобы не вдаваться в детали внутреннего устройства, воспользуемся расширением pgrowlocks:

```
=> CREATE EXTENSION pgrowlocks;

CREATE EXTENSION

=> SELECT * FROM pgrowlocks('accounts') \gx

-[ RECORD 1 ]-----------------
locked_row | (0,1)
locker     | 546
multi      | f
xids       | {546}
modes      | {"No Key Update"}
pids       | {62824}
```

Чтобы показать блокировки, расширение читает табличные страницы (в отличие от обращения к pg_locks, которое читает данные из оперативной памяти).

Теперь изменим номер второго счета (при этом меняется ключ):

```
=> UPDATE accounts SET acc_no = 20 WHERE acc_no = 2;

UPDATE 1

=> SELECT * FROM pgrowlocks('accounts') \gx
```

```
-[ RECORD 1 ]----------------
locked_row | (0,1)
locker     | 546
multi      | f
xids       | {546}
modes      | {"No Key Update"}
pids       | {62824}
-[ RECORD 2 ]----------------
locked_row | (0,2)
locker     | 546
multi      | f
xids       | {546}
modes      | {Update}
pids       | {62824}
```

Чтобы продемонстрировать разделяемые блокировки, начнем еще одну транзакцию. Все запрашиваемые блокировки будут совместимы друг с другом.

```
    => BEGIN;

    BEGIN

    => SELECT * FROM accounts WHERE acc_no = 1 FOR KEY SHARE;

     acc_no | amount
    --------+--------
          1 | 100.00
    (1 row)


    => SELECT * FROM accounts WHERE acc_no = 3 FOR SHARE;

     acc_no | amount
    --------+--------
          3 | 300.00
    (1 row)


    => SELECT txid_current();

     txid_current
    --------------
              548
    (1 row)
```

```
=> SELECT * FROM pgrowlocks('accounts') \gx

-[ RECORD 1 ]----------------------------
locked_row | (0,1)
locker     | 1
multi      | t
xids       | {546,548}
modes      | {"No Key Update","Key Share"}
pids       | {62824,63155}
-[ RECORD 2 ]----------------------------
locked_row | (0,2)
locker     | 546
multi      | f
xids       | {546}
modes      | {Update}
pids       | {62824}
-[ RECORD 3 ]----------------------------
locked_row | (0,3)
locker     | 548
multi      | f
xids       | {548}
modes      | {"For Share"}
pids       | {63155}
```

```
  => ROLLBACK;

  ROLLBACK

    => ROLLBACK;

    ROLLBACK
```

Как не ждать блокировку?

Иногда удобно не ждать освобождения блокировки, а сразу получить ошибку, если необходимый ресурс занят. Приложение может перехватить и обработать такую ошибку.

Для этого ряд команд SQL (такие, как SELECT и некоторые варианты ALTER) позволяют указать ключевое слово NOWAIT. Заблокируем таблицу, обновив первую строку:

```
=> BEGIN;
```

```
BEGIN
```

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

```
UPDATE 1
```

```
=> BEGIN;
```

```
BEGIN
```

```
=> LOCK TABLE accounts NOWAIT; -- IN ACCESS EXCLUSIVE MODE
```

```
ERROR:  could not obtain lock on relation "accounts"
```

Транзакция сразу же получает ошибку.

```
=> ROLLBACK;
```

```
ROLLBACK
```

Команды UPDATE и DELETE не позволяют указать NOWAIT. Но можно сначала выполнить команду

```
SELECT ... FOR UPDATE NOWAIT; -- или FOR NO KEY UPDATE NOWAIT
```

а затем, если строки успешно заблокированы, изменить или удалить их. Например:

```
=> BEGIN;
```

```
BEGIN
```

```
=> SELECT * FROM accounts WHERE acc_no = 1 FOR UPDATE NOWAIT;
```

```
ERROR:  could not obtain lock on row in relation "accounts"
```

Снова тут же получаем ошибку.

```
=> ROLLBACK;
```

```
ROLLBACK
```

Другой способ блокировки строк предоставляет предложение SKIP LOCKED. Заблокируем одну строку, но без указания конкретного номера счета:

```
=> BEGIN;
```

```
BEGIN
```

```
=> SELECT * FROM accounts ORDER BY acc_no
FOR UPDATE SKIP LOCKED LIMIT 1;
```

```
 acc_no | amount
--------+--------
      2 | 200.00
(1 row)
```

В этом случае команда пропускает уже заблокированную первую строку и мы немедленно получаем блокировку второй строки. Этот прием уже использовался в практике темы «Очистка» при выборе пакета строк для обновления. Еще одно применение — в организации очередей — будет рассмотрено в теме «Асинхронная обработка».

```
=> ROLLBACK;
```

```
ROLLBACK
```

Для команд, не связанных с блокировкой строк, использовать NOWAIT не получится. В этом случае можно установить небольшой тайм-аут ожидания (который по умолчанию не задан):

```
=> SET lock_timeout = '1s';
```

```
SET
```

```
=> ALTER TABLE accounts DROP COLUMN amount;
```

```
ERROR:  canceling statement due to lock timeout
```

Получаем ошибку без длительного ожидания освобождения ресурса.

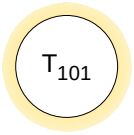
```
=> RESET lock_timeout;
```

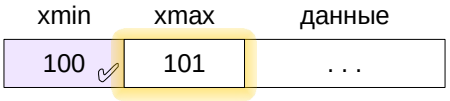
```
   RESET
=> ROLLBACK;
ROLLBACK
```

Очередь ожидания

Порядок действий

1. захватить исключительную блокировку типа tuple нужной версии строки
2. если версия строки занята, дождаться ее освобождения
3. записать номер своей транзакции в xmax
4. освободить блокировку типа tuple





Чтобы заблокировать строку, транзакция выполняет следующие действия:

1. Захватывает исключительную блокировку типа **tuple** для интересующей версии строки.

2. Если xmax и информационные биты говорят о том, что строка заблокирована в конфликтующем режиме, то ждет освобождения строки.

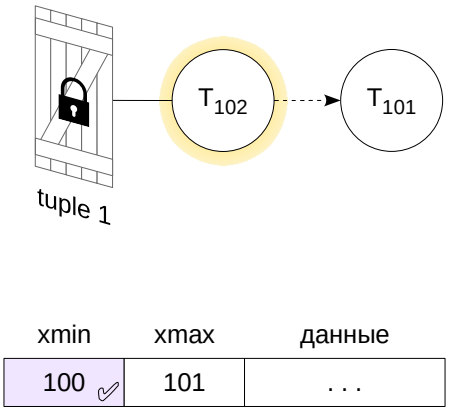
3. Прописывает свой xmax и необходимые информационные биты в версию строки, таким образом блокируя ее.

4. Освобождает блокировку tuple версии строки.

Разберем этот процесс.

В примере, приведенном на слайде, транзакция 101 захватила блокировку tuple, затем прописала свой номер в версии строки и тут же освободила блокировку tuple.

Очередь ожидания

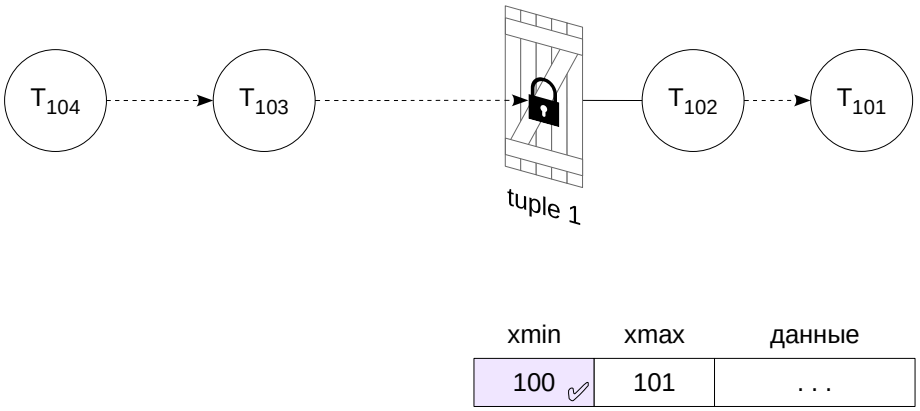


Транзакция 102 захватила блокировку tuple для первой версии строки (п. 1), но затем (п. 2) обнаружила, что версия строки уже заблокирована транзакцией 101.

Теперь ей надо дождаться, пока строка освободится, но как это сделать? Для этого используется своего рода трюк. Как мы уже говорили, каждая транзакция удерживает исключительную блокировку своего номера. Поскольку транзакция 102 фактически должна дождаться завершения транзакции 101 (ведь блокировка строки освобождается только при завершении транзакции), она запрашивает блокировку номера 101.

Обратите внимание, что блокировка tuple при этом остается, так как снимается только в п. 3, до которого дело еще не дошло.

Очередь ожидания

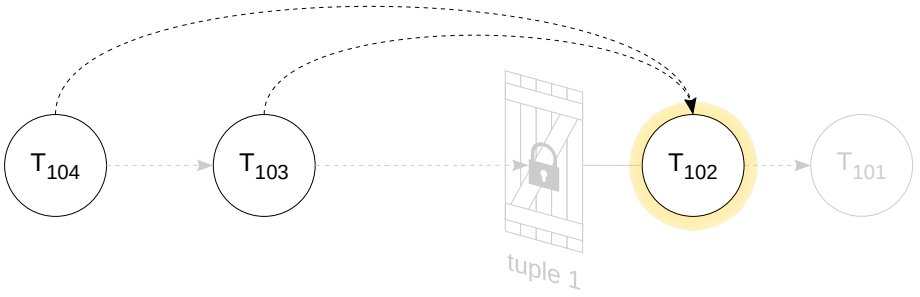


Если появляются другие транзакции, конфликтующие с текущей блокировкой строки, первым делом они пытаются захватить блокировку типа tuple для этой строки — и выстраиваются в очередь.

В нашем примере, поскольку блокировка tuple уже удерживается транзакцией 102, транзакции 103, 104 ждут освобождения этой блокировки.

(Заметим, что транзакции, желающие получить строку в совместимом режиме, проходит без очереди. Обычно это не представляет проблемы, поскольку разделяемые блокировки строк используются по умолчанию только при проверке внешних ключей.)

Очередь ожидания



xmin	xmax	данные
100 ✓	101	. . .
101 ✓	102	. . .

Получается двухуровневая очередь. Одна из транзакций (102 в примере) — «крайняя», и за ней — все остальные.

Когда транзакция 101 завершится, именно транзакция 102 получит возможность первой записать свой номер в поле xmax, после чего она освободит блокировку tuple.

Но дело в том, что при фиксации транзакции 101 блокировка tuple первой версии не просто освобождается — она становится ненужной, ведь теперь актуальной является уже вторая версия строки. (При обрыве транзакции 101 это не так, но обрывы происходят значительно реже.)

Поэтому все транзакции, стоявшие ранее в очереди, превращаются в «толпу» и выстраиваются непосредственно за транзакцией 102. Когда транзакция 102 освободит версию строки, среди ожидающих транзакций возникает гонка за право захватить блокировку.

Если бы не двухуровневая схема блокирования, то такая гонка со случайным победителем возникала бы всегда. Это могло бы приводить к ситуации вечного ожидания «невезучей» транзакции. Наличие блокировки типа tuple решает — хотя бы отчасти — эту проблему.

Стоит избегать проектных решений, которые предполагают массовые изменения одной и той же строки. В этом случае возникает «горячая точка», которая на высоких нагрузках может привести к снижению производительности.

Очередь ожидания

Начинаем транзакцию и обновляем строку.

```
=> BEGIN;
```

```
BEGIN
```

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

```
UPDATE 1
```

Будем смотреть только на блокировки, связанные с номерами транзакций и версиями строк:

```
=> SELECT pid, locktype, page, tuple, transactionid AS xid,
    mode, granted
FROM pg_locks WHERE locktype IN ('transactionid','tuple')
ORDER BY pid, granted DESC, locktype;
```

```
  pid  |   locktype    | page | tuple | xid |     mode      | granted
-------+---------------+------+-------+-----+---------------+---------
 62787 | transactionid |      |       | 554 | ExclusiveLock | t
(1 row)
```

Другая транзакция пытается обновить ту же строку:

```
=> BEGIN;
```

```
BEGIN
```

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

Какие при этом возникают блокировки?

```
=> SELECT pid, locktype, page, tuple, transactionid AS xid,
    mode, granted
FROM pg_locks WHERE locktype IN ('transactionid','tuple')
ORDER BY pid, granted DESC, locktype;
```

```
  pid  |   locktype    | page | tuple | xid |     mode      | granted
-------+---------------+------+-------+-----+---------------+---------
 62787 | transactionid |      |       | 554 | ExclusiveLock | t
 62824 | transactionid |      |       | 555 | ExclusiveLock | t
 62824 | tuple         |    0 |     1 |     | ExclusiveLock | t
 62824 | transactionid |      |       | 554 | ShareLock     | f
(4 rows)
```

Транзакция захватила блокировку, связанную с версией строки 1 на странице 0, и ждет завершения первой транзакции:

```
=> SELECT pid, pg_blocking_pids(pid) FROM pg_stat_activity
WHERE pid IN (62787,62824) ORDER BY pid;
```

```
  pid  | pg_blocking_pids
-------+------------------
 62787 | {}
 62824 | {62787}
(2 rows)
```

Теперь следующая транзакция пытается обновить ту же строку:

```
    => BEGIN;
```

```
    BEGIN
```

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

Что увидим в pg_locks на этот раз?

```
=> SELECT pid, locktype, page, tuple, transactionid AS xid,
    mode, granted
FROM pg_locks WHERE locktype IN ('transactionid','tuple')
ORDER BY pid, granted DESC, locktype;
```

```
  pid  |   locktype    | page | tuple | xid |     mode      | granted
-------+---------------+------+-------+-----+---------------+---------
 62787 | transactionid |      |       | 554 | ExclusiveLock | t
 62824 | transactionid |      |       | 555 | ExclusiveLock | t
 62824 | tuple         |    0 |     1 |     | ExclusiveLock | t
 62824 | transactionid |      |       | 554 | ShareLock     | f
 63155 | transactionid |      |       | 556 | ExclusiveLock | t
 63155 | tuple         |    0 |     1 |     | ExclusiveLock | f
(6 rows)
```

Транзакция встала в очередь за блокировкой версии строки, очередь выросла:

```
=> SELECT pid, pg_blocking_pids(pid) FROM pg_stat_activity
WHERE pid IN (62787,62824,63155) ORDER BY pid;
```

```
  pid  | pg_blocking_pids
-------+------------------
 62787 | {}
 62824 | {62787}
 63155 | {62824}
(3 rows)
```

Если теперь первая транзакция успешно завершается...

```
=> COMMIT;
```

```
COMMIT
```

...то первая версия становится не актуальной.

```
=> SELECT pid, locktype, page, tuple, transactionid AS xid,
    mode, granted
FROM pg_locks WHERE locktype IN ('transactionid','tuple')
ORDER BY pid, granted DESC, locktype;
```

```
  pid  |   locktype    | page | tuple | xid |     mode      | granted
-------+---------------+------+-------+-----+---------------+---------
 62824 | transactionid |      |       | 555 | ExclusiveLock | t
 63155 | transactionid |      |       | 556 | ExclusiveLock | t
 63155 | transactionid |      |       | 555 | ShareLock     | f
(3 rows)
```

Все транзакции, которые стояли в очереди, будут теперь ожидать завершения второй транзакции — и будут обрабатываться в произвольном порядке. Вот как выглядит очередь в нашем случае:

```
=> SELECT pid, pg_blocking_pids(pid) FROM pg_stat_activity
WHERE pid IN (62824,63155) ORDER BY pid;
```

```
  pid  | pg_blocking_pids
-------+------------------
 62824 | {}
 63155 | {62824}
(2 rows)
```

```
=> COMMIT;
```

```
UPDATE 1
COMMIT
```

```
=> COMMIT;
```

```
UPDATE 1
COMMIT
```

Итоги

Блокировки отношений и других объектов БД используются для организации конкурентного доступа к общим ресурсам

- хранятся в разделяемой памяти сервера
- имеется механизм очередей

Блокировки строк реализованы иначе

- хранятся в страницах данных из-за потенциально большого количества
- используют блокировки уровня объектов для организации очереди

Практика

1. Какие блокировки на уровне изоляции Read Committed удерживает транзакция, прочитавшая одну строку таблицы по первичному ключу? Проверьте на практике.
2. Посмотрите, как в представлении pg_locks отображаются рекомендательные блокировки.
3. Убедитесь на практике, что проверка внешнего ключа и обновление строки могут выполняться одновременно. Изучите возникающие при этом блокировки уровня строки.
4. Воспроизведите ситуацию *взаимоблокировки* двух транзакций и проверьте, как она обрабатывается сервером.

2. Рекомендательные блокировки рассматриваются в курсе DEV1.

https://postgrespro.ru/docs/postgresql/12/functions-admin#FUNCTIONS-ADVISORY-LOCKS

Посмотрите все столбцы представления pg_locks чтобы определить, в каком из них отображается идентификатор ресурса.

3. Для этого потребуется создать две таблицы, связанные ограничением внешнего ключа.

Для анализа блокировок используйте расширение pgrowlocks.

4. Взаимоблокировка двух транзакций возникает, когда

- первая транзакция удерживает блокировку объектов, необходимых второй транзакции для продолжения работы,

- а вторая транзакция удерживает блокировку объектов, необходимых первой транзакции для продолжения работы.

В общем случае может произойти взаимоблокировка более двух транзакций.

1. Блокировки при чтении строки по первичному ключу

```
=> CREATE DATABASE locks_overview;

CREATE DATABASE

=> \c locks_overview

You are now connected to database "locks_overview" as user "student".

=> SELECT pg_backend_pid();

 pg_backend_pid
----------------
          96126
(1 row)
```

Создадим таблицу как в демонстрации:

```
=> CREATE TABLE accounts(acc_no integer PRIMARY KEY, amount numeric);

CREATE TABLE

=> INSERT INTO accounts VALUES (1,100.00), (2,200.00), (3,300.00);

INSERT 0 3
```

Прочитаем строку, начав транзакцию:

```
=> \c locks_overview

You are now connected to database "locks_overview" as user "student".

=> SELECT pg_backend_pid();

 pg_backend_pid
----------------
          96235
(1 row)


=> BEGIN;

BEGIN

=> SELECT * FROM accounts WHERE acc_no = 1;

 acc_no | amount
--------+--------
      1 | 100.00
(1 row)
```

Блокировки включают блокировку индекса, поддерживающего ограничение первичного ключа, в режиме Access Share:

```
=> SELECT locktype, relation::REGCLASS, virtualxid AS virtxid,
    transactionid AS xid, mode, granted
FROM pg_locks
WHERE pid = 96235;

  locktype  |   relation    | virtxid | xid |      mode       | granted
------------+---------------+---------+-----+-----------------+---------
 relation   | accounts_pkey |         |     | AccessShareLock | t
 relation   | accounts      |         |     | AccessShareLock | t
 virtualxid |               | 3/22    |     | ExclusiveLock   | t
(3 rows)


=> COMMIT;

COMMIT
```

2. Рекомендательные блокировки

Захватим рекомендательную блокировку уровня сеанса:

```
=> BEGIN;

BEGIN
```

```
=> SELECT pg_advisory_lock(42);
```

```
 pg_advisory_lock
------------------

(1 row)
```

Блокировка:

```
=> SELECT locktype, virtualxid AS virtxid, objid, mode, granted
FROM pg_locks
WHERE pid = 96235;
```

```
  locktype  | virtxid | objid |     mode      | granted
------------+---------+-------+---------------+---------
 virtualxid | 3/23    |       | ExclusiveLock | t
 advisory   |         |    42 | ExclusiveLock | t
(2 rows)
```

Идентификатор ресурса для блокировки типа advisory отображается в столбце objid.

```
=> COMMIT;
```

```
COMMIT
```

3. Проверка внешнего ключа

Нам понадобится расширение для анализа блокировок на уровне строк:

```
=> CREATE EXTENSION pgrowlocks;
```

```
CREATE EXTENSION
```

Создадим таблицу клиентов:

```
=> CREATE TABLE clients(
    client_id integer PRIMARY KEY,
    name text
);
```

```
CREATE TABLE
```

```
=> INSERT INTO clients VALUES (10,'alice'), (20,'bob');
```

```
INSERT 0 2
```

В таблицу счетов добавим столбец для идентификатора клиента и внешний ключ:

```
=> ALTER TABLE accounts
    ADD client_id integer REFERENCES clients(client_id);
```

```
ALTER TABLE
```

Внутри транзакции выполним какое-нибудь действие с таблицей счетов, вызывающее проверку внешнего ключа. Например, вставим строку:

```
=> BEGIN;
```

```
BEGIN
```

```
=> INSERT INTO accounts(acc_no, amount, client_id)
    VALUES (4,400.00,20);
```

```
INSERT 0 1
```

Проверка внешнего ключа приводит к появлению блокировки строки в таблице клиентов в режиме KeyShare:

```
=> SELECT * FROM pgrowlocks('clients') \gx
```

```
-[ RECORD 1 ]-----------------
locked_row | (0,2)
locker     | 86527
multi      | f
xids       | {86527}
modes      | {"For Key Share"}
pids       | {96235}
```

Это не мешает изменять неключевые столбцы этой строки:

```
=> UPDATE clients SET name = 'brian' WHERE client_id = 20;
```

```
UPDATE 1
```

```
=> COMMIT;
```

```
COMMIT
```

4. Взаимоблокировка двух транзакций

Обычная причина возникновения взаимоблокировок — разный порядок блокирования строк таблиц в приложении.

Первая транзакция намерена перенести 100 рублей с первого счета на второй. Для этого она сначала уменьшает первый счет:

```
=> BEGIN;
```

```
BEGIN
```

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

```
UPDATE 1
```

В это же время вторая транзакция намерена перенести 10 рублей со второго счета на первый. Она начинает с того, что уменьшает второй счет:

```
=> BEGIN;
```

```
BEGIN
```

```
=> UPDATE accounts SET amount = amount - 10.00 WHERE acc_no = 2;
```

```
UPDATE 1
```

Теперь первая транзакция пытается увеличить второй счет...

```
=> UPDATE accounts SET amount = amount + 100.00 WHERE acc_no = 2;
```

...но обнаруживает, что строка заблокирована.

Затем вторая транзакция пытается увеличить первый счет...

```
=> UPDATE accounts SET amount = amount + 10.00 WHERE acc_no = 1;
```

...но тоже блокируется.

Возникает циклическое ожидание, которое никогда не завершится само по себе. Поэтому если какая-либо блокировка не получена за время, указанное в параметре deadlock_timeout (по умолчанию — 1 секунда), сервер проверяет наличие циклов ожидания. Обнаружив такой цикл, он прерывает одну из транзакций, чтобы остальные могли продолжить работу.

```
UPDATE 1
```

```
=> COMMIT;
```

```
COMMIT
```

```
ERROR:  deadlock detected
DETAIL:  Process 96235 waits for ShareLock on transaction 86529; blocked by process 96126.
Process 96126 waits for ShareLock on transaction 86530; blocked by process 96235.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,1) in relation "accounts"
```

```
=> COMMIT;
```

```
ROLLBACK
```

Взаимоблокировки обычно означают, что приложение спроектировано неправильно. Правильный способ выполнения таких операций — блокирование ресурсов в одном и том же порядке. Например, в данном случае можно блокировать счета в порядке возрастания их номеров.

Тем не менее взаимоблокировки могут возникать и при нормальной работе (например, могут взаимозаблокироваться две команды UPDATE одной и той же таблицы). Но это очень редкие ситуации.