Блокировки

Блокировки строк

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов

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

Темы

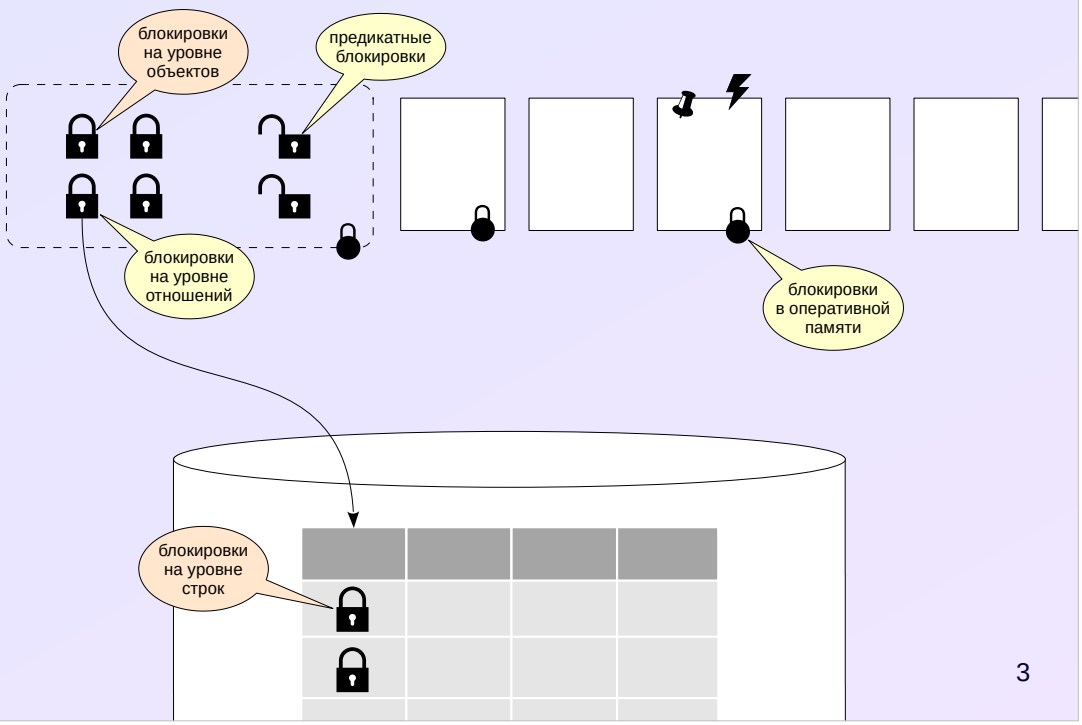
Исключительные и разделяемые блокировки строк

Мультитранзакции и заморозка

Реализация очереди ожидания

Взаимоблокировки

Блокировки строк



Устройство

Информация *только* в страницах данных

- поле xmax заголовка версии строки + информационные биты
- в оперативной памяти ничего не хранится

Неограниченное количество

- большое число не влияет на производительность

Инфраструктура

- очередь ожидания организована с помощью блокировок объектов

В случае блокировок объектов, рассмотренных в предыдущей теме этого модуля, каждый ресурс представлен собственной блокировкой в оперативной памяти. Со строками так не получается: заведение отдельной блокировки для каждой табличной строки (которых могут быть миллионы и миллиарды) потребует непомерных накладных расходов и огромного объема оперативной памяти. А если повышать уровень блокировок — будет страдать пропускная способность.

Поэтому в PostgreSQL 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 количество блокировок уровня строки ничем не ограничено. 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 более высокого уровня, но удается обойтись очень небольшим их количеством (пропорциональным числу процессов, а не числу заблокированных строк). И это также не снижает производительность системы.

Исключительные режимы

Update

удаление строки
или изменение всех полей

`SELECT FOR UPDATE`

`UPDATE`
(с изменением ключевых полей)

`DELETE`

No Key Update

изменение любых полей,
кроме ключевых

`SELECT FOR NO KEY UPDATE`

`UPDATE`
(без изменения ключевых полей)

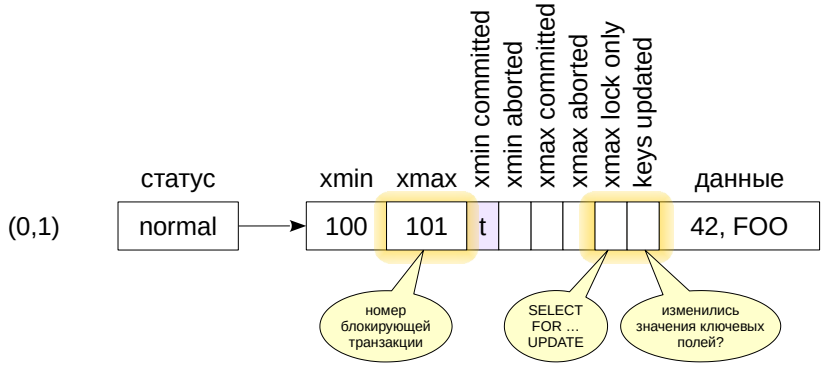
Всего существует 4 режима, в которых можно заблокировать строку (в версии строки режим проставляется с помощью дополнительных информационных битов).

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.

https://postgrespro.ru/docs/postgresql/13/explicit-locking#LOCKING-ROWS

Исключительные режимы



При изменении или удалении строки в поле xmax актуальной версии записывается номер текущей транзакции. Установленное значение xmax, соответствующее активной транзакции, выступает в качестве блокировки.

То же самое происходит и при явном блокировании строки командой SELECT FOR UPDATE, но проставляется дополнительный информационный бит (xmax lock only), который говорит о том, что версия строки по-прежнему актуальна, хоть и заблокирована.

Режим блокировки определяется еще одним информационным битом (keys updated).

(На самом деле используется большее количество битов с более сложными условиями и проверками, что связано с поддержкой совместимости с предыдущими версиями. Но это не принципиально.)

Если другая транзакция намерена обновить или удалить заблокированную строку в несовместимом режиме, она будет вынуждена дождаться завершения транзакции с номером xmax.

Разделяемые режимы

Share

запрет изменения
любых полей строки

`SELECT FOR SHARE`

Key Share

запрет изменения
ключевых полей строки

`SELECT FOR KEY SHARE`
и проверка внешних ключей

Матрица совместимости режимов

	Key Share	Share	No Key Update	Update
Key Share				×
Share			×	×
No Key Update		×	×	×
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.

Общая матрица совместимости режимов приведена внизу слайда. Из нее видно, что:

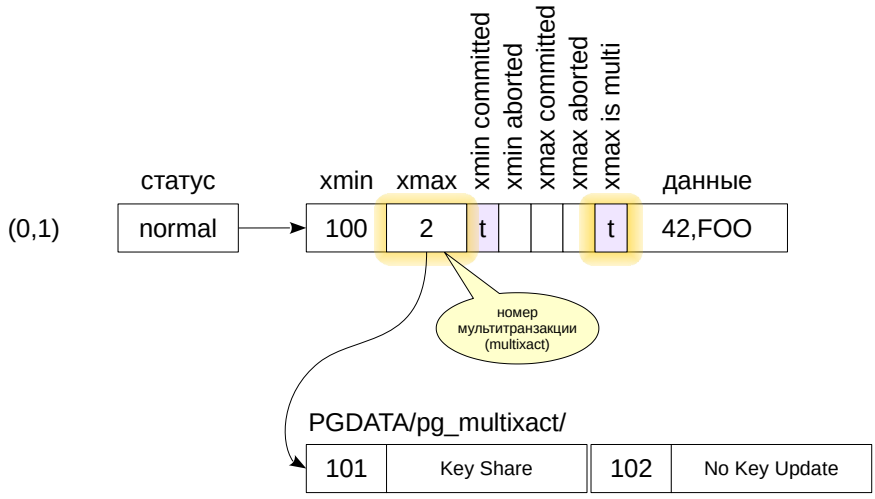
- исключительные режимы конфликтуют между собой;

- разделяемые режимы совместимы между собой;

- 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/13/explicit-locking#LOCKING-ROWS

Разделяемые режимы



Мы говорили о том, что блокировка представляется номером блокирующей транзакции в поле xmax. Разделяемые блокировки могут удерживаться несколькими транзакциями, но в одно поле xmax нельзя записать несколько номеров.

Поэтому для разделяемых блокировок применяются так называемые *мультитранзакции* (MultiXact). Им выделяются отдельные номера, которые соответствуют не одной транзакции, а целой группе. Чтобы отличить мультитранзакцию от обычной, используется еще один информационный бит (xmax is multi), а детальная информация об участниках такой группы и режимах блокировки находятся в каталоге PGDATA/pg_multixact/. Естественно, последние использованные данные хранятся в буферах в общей памяти сервера для ускорения доступа.

Настройка заморозки

Параметры для мультитранзакций

vacuum_multixact_freeze_min_age = 5 000 000

vacuum_multixact_freeze_table_age = 150 000 000

autovacuum_multixact_freeze_max_age = 400 000 000

Параметры хранения таблиц

autovacuum_multixact_freeze_min_age
toast.autovacuum_multixact_freeze_min_age

vacuum_multixact_freeze_table_age
toast.vacuum_multixact_freeze_table_age

autovacuum_multixact_freeze_max_age
toast.autovacuum_multixact_freeze_max_age

Поскольку для мультитранзакций выделяются отдельные номера, которые записываются в поле xmax версий строк, из-за ограничения разрядности счетчика с ними возникают такие же сложности, как и с обычным номером транзакции. Речь идет о проблеме переполнения (xid wraparound), которая рассматривалась в теме «Заморозка» модуля «Многоверсионность».

Поэтому для номеров мультитранзакций тоже необходимо выполнять аналог заморозки — старые номера multixact id заменяются на новые (или на обычный номер, если в текущий момент блокировка уже удерживается только одной транзакцией).

Заметим, что обычная заморозка версий строк выполняется для поля xmin (если у версии строки непустое поле xmax, то либо это уже неактуальная версия и она будет очищена, либо транзакция xmax отменена и ее номер нас не интересует). А для мультитранзакций речь идет о поле xmax актуальной версии строки, которая может оставаться актуальной, но при этом постоянно блокироваться разными транзакциями в разделяемом режиме.

За заморозку мультитранзакций отвечают параметры, аналогичные параметрам обычной заморозки.

Блокировки строк

Наиболее частый случай блокировок — блокировки, возникающие на уровне строк.

Создадим таблицу счетов, как в прошлой теме.

```
=> CREATE DATABASE locks_rows;
```

```
CREATE DATABASE
```

```
=> \c locks_rows
```

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

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

```
CREATE TABLE
```

```
=> INSERT INTO accounts VALUES (1,1000.00),(2,2000.00),(3,3000.00);
```

```
INSERT 0 3
```

Поскольку информация о блокировке строк хранится только в самих версиях строк, воспользуемся знакомым расширением pageinspect.

```
=> CREATE EXTENSION pageinspect;
```

```
CREATE EXTENSION
```

Для удобства создадим представление, расшифровывающее интересующие нас информационные биты в первых трех версиях строк.

```
=> CREATE VIEW accounts_v AS
SELECT '(0,'||lp||')' AS ctid,
       t_xmax as xmax,
       CASE WHEN (t_infomask & 1024) > 0  THEN 't' END AS committed,
       CASE WHEN (t_infomask & 2048) > 0  THEN 't' END AS aborted,
       CASE WHEN (t_infomask & 128) > 0   THEN 't' END AS lock_only,
       CASE WHEN (t_infomask & 4096) > 0  THEN 't' END AS is_multi,
       CASE WHEN (t_infomask2 & 8192) > 0 THEN 't' END AS keys_upd
FROM heap_page_items(get_raw_page('accounts',0))
WHERE lp <= 3
ORDER BY lp;
```

```
CREATE VIEW
```

Обновляем сумму первого счета (ключ не меняется) и номер второго счета (ключ меняется):

```
=> \c locks_rows
```

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

```
=> BEGIN;
```

```
BEGIN
```

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

```
UPDATE 1
```

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

```
UPDATE 1
```

Заглянем в представление. Напомним, что оно показывает только первые три (то есть исходные) версии строк.

```
=> SELECT * FROM accounts_v;
```

```
 ctid  |  xmax  | committed | aborted | lock_only | is_multi | keys_upd
-------+--------+-----------+---------+-----------+----------+----------
 (0,1) | 515366 |           |         |           |          |
 (0,2) | 515366 |           |         |           |          | t
 (0,3) |      0 |           | t       |           |          |
(3 rows)
```

По столбцу keys_upd видно, что строки заблокированы в разных режимах.

Теперь в другом сеансе запросим разделяемые блокировки первого и третьего счетов:

```
    => \c locks_rows
```

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

```
=> BEGIN;
```

```
BEGIN
```

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

```
 acc_no | amount
--------+---------
      1 | 1000.00
(1 row)
```

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

```
 acc_no | amount
--------+---------
      3 | 3000.00
(1 row)
```

Все запрошенные блокировки совместимы друг с другом. В версиях строк видим:

```
=> SELECT * FROM accounts_v;
```

```
 ctid  |  xmax  | committed | aborted | lock_only | is_multi | keys_upd
-------+--------+-----------+---------+-----------+----------+----------
 (0,1) |      5 |           |         |           | t        |
 (0,2) | 515366 |           |         |           |          | t
 (0,3) | 515367 |           |         | t         |          |
(3 rows)
```

Столбец lock_only позволяет отличить просто блокировку от обновления или удаления. В первой строке видим, что обычный номер заменен на номер мультитранзакции — об этом говорит столбец is_multi.

Чтобы не вникать в детали информационных битов и реализацию мультитранзакций, можно воспользоваться еще одним расширением, которое позволяет увидеть всю информацию о блокировках строк в удобном виде.

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

```
CREATE EXTENSION
```

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

```
-[ RECORD 1 ]-------------------------------
locked_row | (0,1)
locker     | 5
multi      | t
xids       | {515366,515367}
modes      | {"No Key Update","Key Share"}
pids       | {99909,100070}
-[ RECORD 2 ]-------------------------------
locked_row | (0,2)
locker     | 515366
multi      | f
xids       | {515366}
modes      | {Update}
pids       | {99909}
-[ RECORD 3 ]-------------------------------
locked_row | (0,3)
locker     | 515367
multi      | f
xids       | {515367}
modes      | {"For Share"}
pids       | {100070}
```

Расширение дает информацию о режимах всех блокировок.

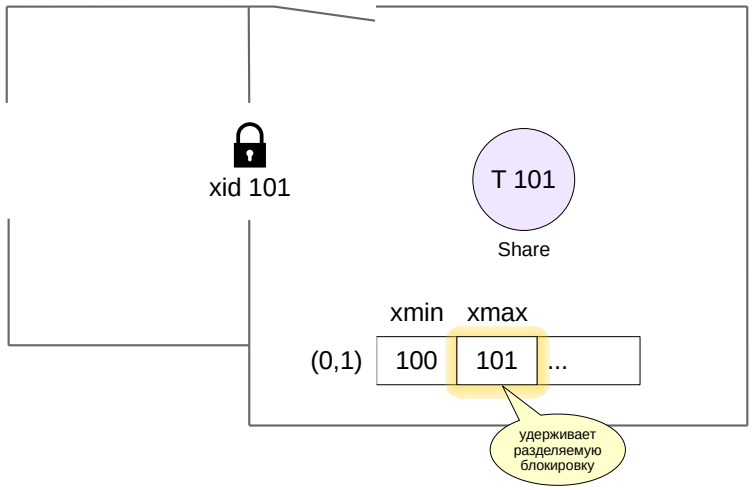
```
=> ROLLBACK;
```

```
ROLLBACK
```

```
=> ROLLBACK;
```

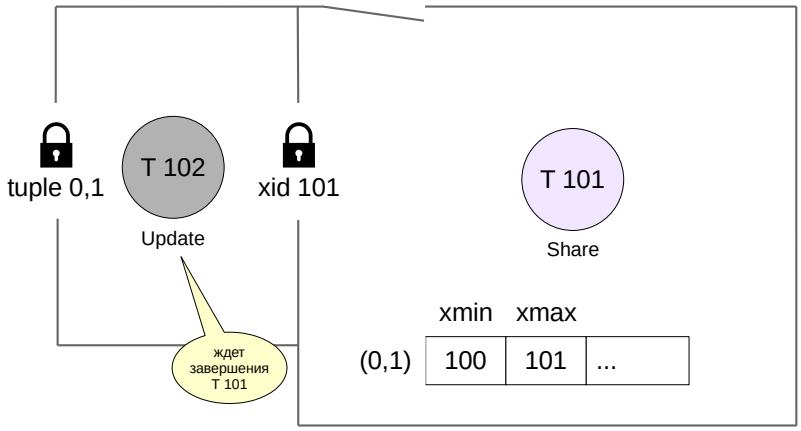
```
ROLLBACK
```

«Очередь» ожидания



Как мы обсудили, транзакция блокирует строку, проставляя в поле xmax свой номер (101 в примере на слайде).

«Очередь» ожидания



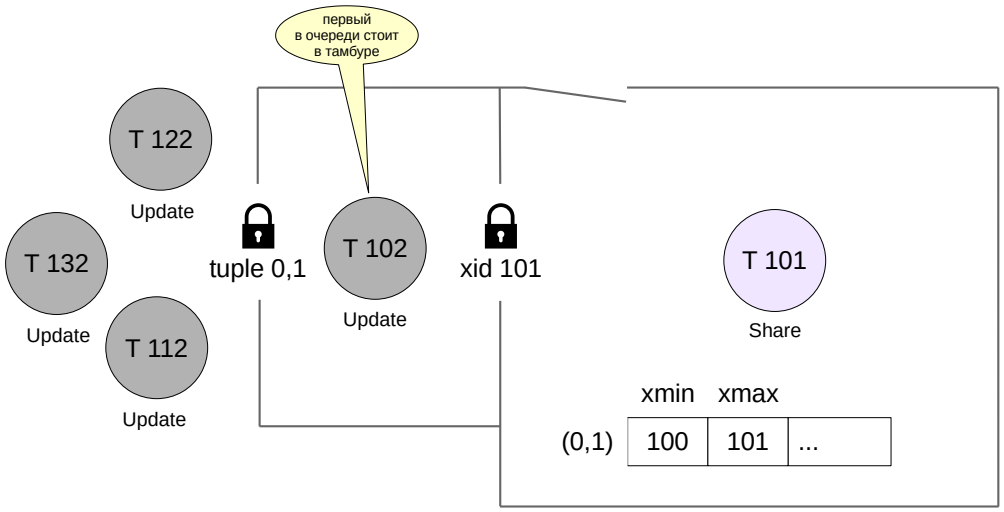
Другая транзакция (102), желая получить блокировку, обращается к строке и видит, что строка заблокирована. Если режимы блокировок конфликтуют, она должна каким-то образом встать в очередь, чтобы система «разбудила» ее, когда блокировка освободится. Но блокировки на уровне строк не предоставляют такой возможности — они никак не представлены в оперативной памяти, это просто байты внутри страницы данных.

Как мы видели в теме «Блокировки объектов», 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.

Когда транзакция 101 завершится, заблокированный ресурс освободится (при фиксации — просто исчезнет), транзакция 102 будет разбужена и сможет заблокировать строку (установить xmax = 102 в соответствующей версии строки).

Кроме того, ожидающая транзакция удерживает блокировку ресурса типа tuple (версия строки), показывая, что она первая в очереди.

«Очередь» ожидания



Если появляются другие транзакции, конфликтующие с текущей блокировкой версии строки (в нашем примере — 112, 122, 132), первым делом они пытаются захватить блокировку типа tuple для этой версии.

Поскольку блокировка tuple уже удерживается транзакцией 102, другие транзакции ждут освобождения этой блокировки. Получается своеобразная «очередь», в которой есть *первый* и *все остальные*.

Идея такой двухуровневой схемы блокирования состоит в том, чтобы избегать ситуации вечного ожидания «невезучей» транзакции. Если бы все прибывающие транзакции занимали очередь за транзакцией 101, при освобождении блокировки возникала бы ситуация гонки. Наличие второй блокировки немного упорядочивает ситуацию. Тем не менее, стоит избегать появления «горячих» строк при высокой нагрузке.

«Очередь» ожидания

Для удобства создадим представление над pg_locks, «свернув» в одно поле идентификаторы разных типов блокировок:

```
=> CREATE VIEW locks AS
SELECT pid,
       locktype,
       CASE locktype
         WHEN 'relation' THEN relation::regclass::text
         WHEN 'virtualxid' THEN virtualxid::text
         WHEN 'transactionid' THEN transactionid::text
         WHEN 'tuple' THEN relation::REGCLASS::text||':'||page::text||','||tuple::text
       END AS lockid,
       mode,
       granted
FROM pg_locks;
```

```
CREATE VIEW
```

Пусть одна транзакция заблокирует строку в разделяемом режиме...

```
S1=> BEGIN;
```

```
BEGIN
```

```
S1=> SELECT txid_current(), pg_backend_pid();
```

```
 txid_current | pg_backend_pid
--------------+----------------
       515370 |          99766
(1 row)
```

```
S1=> SELECT * FROM accounts WHERE acc_no = 1 FOR SHARE;
```

```
 acc_no | amount
--------+---------
      1 | 1000.00
(1 row)
```

...а другая попробует выполнить обновление:

```
U1=> BEGIN;
```

```
BEGIN
```

```
U1=> SELECT txid_current(), pg_backend_pid();
```

```
 txid_current | pg_backend_pid
--------------+----------------
       515371 |          99909
(1 row)
```

```
U1=> UPDATE accounts SET amount = amount + 100.00 WHERE acc_no = 1;
```

В представлении pg_locks можно увидеть, что вторая транзакция ожидает завершения первой (granted = f), удерживая при этом блокировку версии строки (locktype = tuple):

```
=> SELECT * FROM locks WHERE pid = 99909; -- U1
```

```
  pid  |   locktype    |    lockid     |       mode       | granted
-------+---------------+---------------+------------------+---------
 99909 | relation      | accounts_pkey | RowExclusiveLock | t
 99909 | relation      | accounts      | RowExclusiveLock | t
 99909 | virtualxid    | 3/16          | ExclusiveLock    | t
 99909 | tuple         | accounts:0,1  | ExclusiveLock    | t
 99909 | transactionid | 515371        | ExclusiveLock    | t
 99909 | transactionid | 515370        | ShareLock        | f
(6 rows)
```

Чтобы не разбираться, кто кого блокирует, по представлению pg_locks, можно узнать номер (или номера) процесса блокирующего сеанса с помощью функции:

```
=> SELECT pg_blocking_pids(99909); -- U1
```

```
 pg_blocking_pids
------------------
 {99766}
(1 row)
```

Теперь появляется транзакция, желающая получить несовместимую блокировку.

```
U2=> BEGIN;
```

```
BEGIN
```

```
U2=> SELECT txid_current(), pg_backend_pid();
```

```
 txid_current | pg_backend_pid
--------------+----------------
       515372 |         100070
(1 row)
```

```
U2=> UPDATE accounts SET amount = amount - 100.00 WHERE acc_no = 1;
```

Она встает в очередь за транзакцией, удерживающей блокировку версии строки:

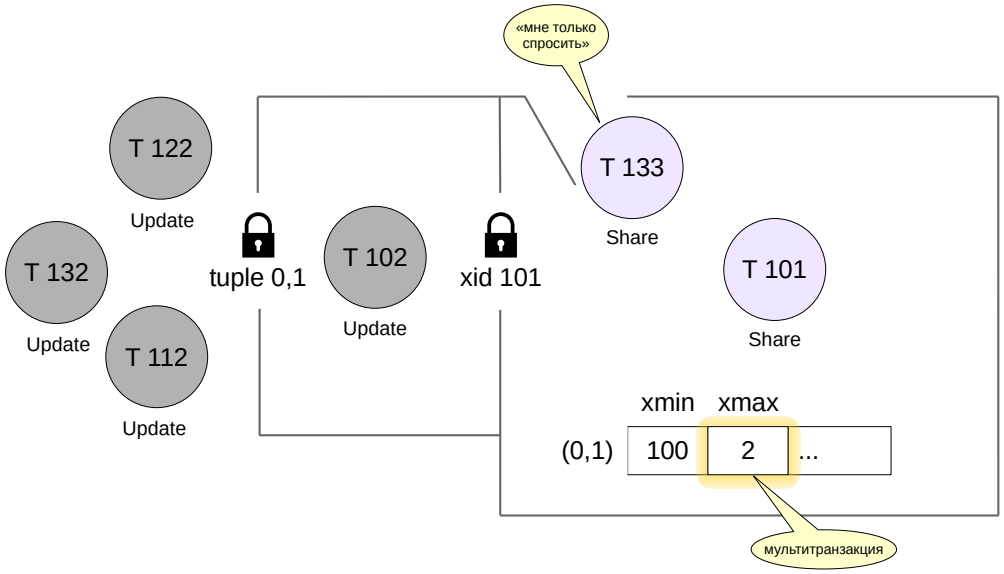
```
=> SELECT * FROM locks WHERE pid = 100070; -- U2
```

```
  pid   |   locktype    |    lockid     |       mode       | granted
--------+---------------+---------------+------------------+---------
 100070 | relation      | accounts_pkey | RowExclusiveLock | t
 100070 | relation      | accounts      | RowExclusiveLock | t
 100070 | virtualxid    | 4/6           | ExclusiveLock    | t
 100070 | tuple         | accounts:0,1  | ExclusiveLock    | f
 100070 | transactionid | 515372        | ExclusiveLock    | t
(5 rows)
```

```
=> SELECT pg_blocking_pids(100070); -- U2
```

```
 pg_blocking_pids
------------------
 {99909}
(1 row)
```

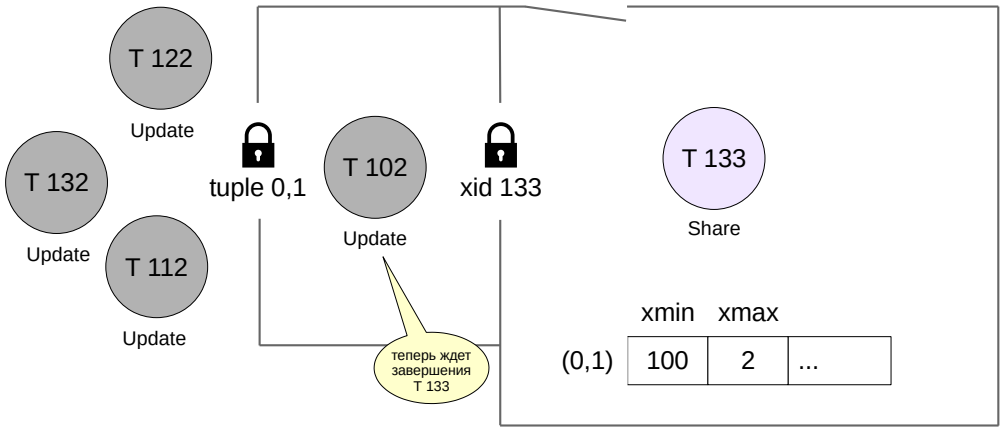
«Очередь» ожидания



В нашем примере транзакция 101 заблокировала строку в разделяемом (Share) режиме. Пока транзакция 102 ожидает завершения транзакции 101, может появиться еще транзакция 133, желающая получить блокировку строки в разделяемом режиме, совместимом с текущей блокировкой транзакции 101.

Такая транзакция не стоит в очереди. Она сразу получает блокировку: формируется мультитранзакция, состоящая из транзакций 101 и 133, и номер мультитранзакции записывается в поле xmax.

«Очередь» ожидания



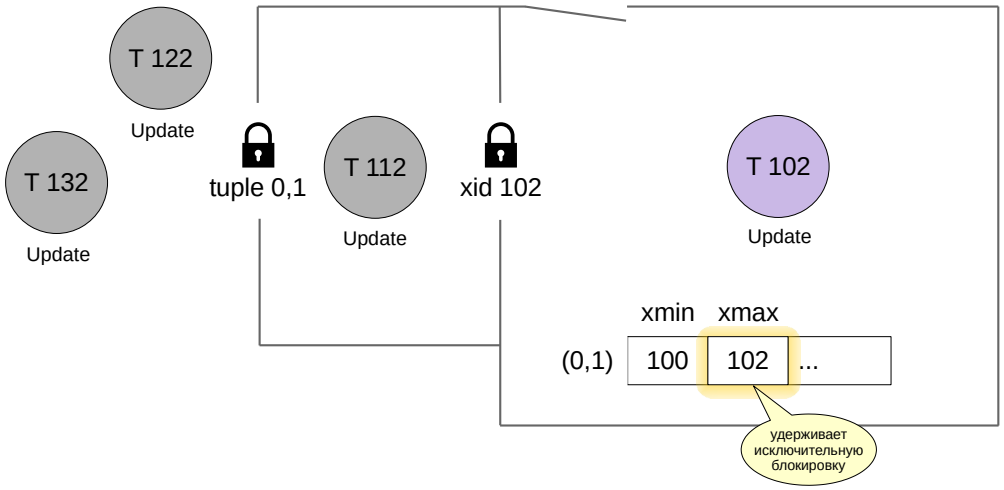
Теперь, когда транзакция 101 завершится, транзакция 102 будет разбужена, но не сможет заблокировать строку, поскольку ее теперь блокирует транзакция 133. Транзакция 102 будет вынуждена снова «заснуть».

При постоянном потоке разделяемых блокировок пишущая транзакция может ждать своей очереди бесконечно. Это ситуация называется по-английски locker starvation.

Заметим, что такой проблемы в принципе не возникает при блокировках объектов (таких, как отношения). В этом случае каждый ресурс представлен собственной блокировкой в оперативной памяти, и все ждущие процессы выстраиваются в «честную» очередь.

https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob;f=src/backend/access/heap/README.tuplock;hb=HEAD

«Очередь» ожидания



Когда транзакция 133 завершится, транзакция 102 получит возможность первой записать свой номер в поле xmax, после чего она освободит блокировку tuple. Тогда одна случайная транзакция из *всех остальных* успеет захватить блокировку tuple и станет *первой*.

Транзакция, желающая заблокировать строку в разделяемом режиме, проходит в нашем примере без очереди.

```
S2=> \c locks_rows
```

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

```
S2=> BEGIN;
```

```
BEGIN
```

```
S2=> SELECT txid_current(), pg_backend_pid();
```

```
 txid_current | pg_backend_pid
--------------+----------------
       515373 |         100801
(1 row)
```

```
S2=> SELECT * FROM accounts WHERE acc_no = 1 FOR SHARE;
```

```
 acc_no | amount
--------+---------
      1 | 1000.00
(1 row)
```

В версии строки теперь мультитранзакция:

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

```
-[ RECORD 1 ]---------------
locked_row | (0,1)
locker     | 6
multi      | t
xids       | {515370,515373}
modes      | {Share,Share}
pids       | {99766,100801}
```

После того как одна из транзакций, удерживающих строку в разделяемом режиме, завершится, другая продолжит удерживать блокировку.

```
S1=> COMMIT;
```

```
COMMIT
```

Транзакция, стоящая первой в очереди, теперь ждет завершения оставшейся транзакции.

```
=> SELECT * FROM locks WHERE pid = 99909; -- U1
```

```
  pid  |   locktype    |    lockid     |       mode       | granted
-------+---------------+---------------+------------------+---------
 99909 | relation      | accounts_pkey | RowExclusiveLock | t
 99909 | relation      | accounts      | RowExclusiveLock | t
 99909 | virtualxid    | 3/16          | ExclusiveLock    | t
 99909 | tuple         | accounts:0,1  | ExclusiveLock    | t
 99909 | transactionid | 515373        | ShareLock        | f
 99909 | transactionid | 515371        | ExclusiveLock    | t
(6 rows)
```

Обратите снимание, что в поле xmax остался номер мультитранзакции, хотя одна из указанных транзакций уже завершилась. Этот номер может быть заменен на другой (новой мультитранзакции или обычной транзакции) при очистке.

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

```
-[ RECORD 1 ]---------------
locked_row | (0,1)
locker     | 6
multi      | t
xids       | {515370,515373}
modes      | {Share,Share}
pids       | {0,100801}
```

Теперь завершается и вторая транзакция, удерживавшая разделяемую блокировку.

```
S2=> COMMIT;
```

```
COMMIT
```

Транзакция, стоявшая первой в очереди, получает доступ к версии строки:

```
UPDATE 1
```

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

```
-[ RECORD 1 ]-----------------
locked_row | (0,1)
locker     | 515371
multi      | f
xids       | {515371}
modes      | {"No Key Update"}
pids       | {99909}
```

Оставшаяся транзакция захватывает блокировку tuple версии строки и становится первой в очереди:

```
=> SELECT * FROM locks WHERE pid = 100070; -- U2
```

```
  pid   |   locktype    |    lockid     |       mode       | granted
--------+---------------+---------------+------------------+---------
 100070 | relation      | accounts_pkey | RowExclusiveLock | t
 100070 | relation      | accounts      | RowExclusiveLock | t
 100070 | virtualxid    | 4/6           | ExclusiveLock    | t
 100070 | transactionid | 515371        | ShareLock        | f
 100070 | tuple         | accounts:0,1  | ExclusiveLock    | t
 100070 | transactionid | 515372        | ExclusiveLock    | t
(6 rows)
```

Отменим изменения.

```
U1=> ROLLBACK;
```

```
ROLLBACK
```

```
   UPDATE 1
```

```
   U2=> 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
```

Для рекомендательных блокировок также есть функции, позволяющие либо сразу захватить блокировку, либо получить ошибку:

```
=> \df pg_try_advisory*
```

```
                                  List of functions
   Schema   |               Name               | Result data type | Argument data types | Type
------------+----------------------------------+------------------+---------------------+------
 pg_catalog | pg_try_advisory_lock             | boolean          | bigint              | func
 pg_catalog | pg_try_advisory_lock             | boolean          | integer, integer    | func
 pg_catalog | pg_try_advisory_lock_shared      | boolean          | bigint              | func
 pg_catalog | pg_try_advisory_lock_shared      | boolean          | integer, integer    | func
 pg_catalog | pg_try_advisory_xact_lock        | boolean          | bigint              | func
 pg_catalog | pg_try_advisory_xact_lock        | boolean          | integer, integer    | func
 pg_catalog | pg_try_advisory_xact_lock_shared | boolean          | bigint              | func
 pg_catalog | pg_try_advisory_xact_lock_shared | boolean          | integer, integer    | func
(8 rows)
```

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 | 2000.00
(1 row)
```

В этом случае команда пропускает уже заблокированную первую строку и мы немедленно получаем блокировку второй строки.

```
=> 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
```

А при выполнении очистки можно указать, что команда должна пропускать обработку таблицы, если ее блокировку не удалось получить немедленно. Это может оказаться особенно актуальным, когда выполняется очистка всей базы:

```
=> VACUUM (skip_locked);
```

```
VACUUM
```

```
=> ROLLBACK;
```

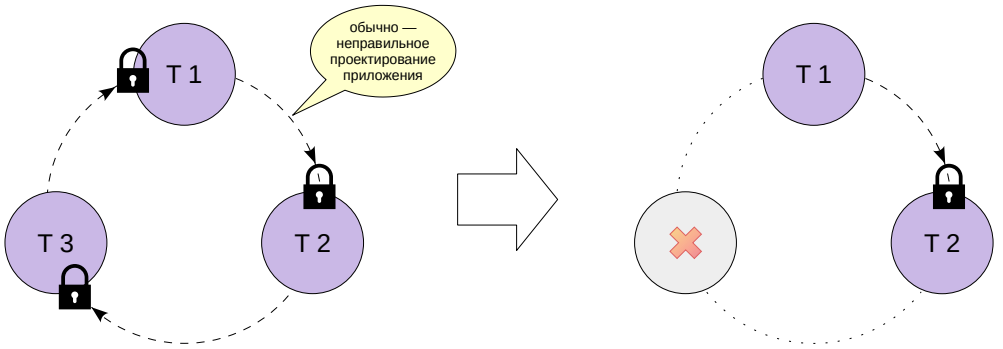
```
ROLLBACK
```

Взаимоблокировки

Обнаруживаются поиском контуров в графе ожиданий

проверка выполняется после ожидания *deadlock_timeout*

Одна из транзакций обрывается, остальные продолжают



Возможна ситуация взаимоблокировки, когда одна транзакция пытается захватить ресурс, уже захваченный другой транзакцией, в то время как другая транзакция пытается захватить ресурс, захваченный первой. Взаимоблокировка возможна и при нескольких транзакциях: на слайде показан пример такой ситуации для трех транзакций.

Визуально взаимоблокировку удобно представлять, построив граф ожиданий. Для этого мы убираем конкретные ресурсы и оставляем только транзакции, отмечая, какая транзакция какую ожидает. Если в графе есть контур (из некоторой вершины можно по стрелкам добраться до нее же самой) — это взаимоблокировка.

Если взаимоблокировка возникла, участвующие транзакции уже не могут ничего с этим сделать — они будут ждать бесконечно. Поэтому PostgreSQL автоматически отслеживает взаимоблокировки. Проверка выполняется, если какая-либо транзакция ожидает освобождения ресурса дольше, чем указано в параметре *deadlock_timeout*. Если выявлена взаимоблокировка, одна из транзакций принудительно прерывается, чтобы остальные могли продолжить работу.

Взаимоблокировки обычно означают, что приложение спроектировано неправильно. Сообщения в журнале сервера или увеличивающееся значение pg_stat_database.deadlocks — повод задуматься о причинах.

https://postgrespro.ru/docs/postgresql/13/explicit-locking#LOCKING-DEADLOCKS

Взаимоблокировка

Обычная причина возникновения взаимоблокировок — разный порядок блокирования строк таблиц.

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, но обнаруживает, что строка заблокирована.

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

Затем вторая транзакция пытается увеличить первый счет, но тоже блокируется.

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

Возникает циклическое ожидание, который никогда не завершится само по себе. Поэтому сервер, обнаружив такой цикл, прерывает одну из транзакций.

```
ERROR:  deadlock detected
DETAIL:  Process 99909 waits for ShareLock on transaction 515379; blocked by process 100070.
Process 100070 waits for ShareLock on transaction 515378; blocked by process 99909.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,2) in relation "accounts"
```

```
    UPDATE 1
```

```
=> COMMIT;
```

```
ROLLBACK
```

```
    => COMMIT;
```

```
    COMMIT
```

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.

Итоги

Блокировки строк хранятся в страницах данных

- из-за потенциально большого количества

Очереди и обнаружение взаимоблокировок обеспечиваются блокировками объектов

- приходится прибегать к сложным схемам блокирования

Практика

1. Смоделируйте ситуацию обновления одной и той же строки тремя командами UPDATE в разных сеансах.
   Изучите возникшие блокировки в представлении pg_locks и убедитесь, что все они понятны.
2. Воспроизведите взаимоблокировку трех транзакций.
   Можно ли разобраться в ситуации постфактум, изучая журнал сообщений?
3. Могут ли две транзакции, выполняющие единственную команду UPDATE одной и той же таблицы, заблокировать друг друга? Попробуйте воспроизвести такую ситуацию.

1. Блокировки при нескольких обновлениях строки

Для простоты создадим таблицу без первичного ключа.

```
=> CREATE DATABASE locks_rows;

CREATE DATABASE

=> \c locks_rows

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

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

CREATE TABLE

=> INSERT INTO accounts VALUES (1,1000.00),(2,2000.00),(3,3000.00);

INSERT 0 3
```

Создадим представление над pg_locks как в демонстрации:

```
=> CREATE VIEW locks AS
SELECT pid,
       locktype,
       CASE locktype
         WHEN 'relation' THEN relation::REGCLASS::text
         WHEN 'virtualxid' THEN virtualxid::text
         WHEN 'transactionid' THEN transactionid::text
         WHEN 'tuple' THEN relation::REGCLASS::text||':'||page::text||','||tuple::text
       END AS lockid,
       mode,
       granted
FROM pg_locks;

CREATE VIEW
```

Первая транзакция обновляет и, соответственно, блокирует строку:

```
=> BEGIN;

BEGIN

=> SELECT txid_current(), pg_backend_pid();

 txid_current | pg_backend_pid
--------------+----------------
       695923 |         125866
(1 row)


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

UPDATE 1
```

Вторая транзакция делает то же самое:

```
=> \c locks_rows

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

=> BEGIN;

BEGIN

=> SELECT txid_current(), pg_backend_pid();

 txid_current | pg_backend_pid
--------------+----------------
       695924 |         126040
(1 row)


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

И третья:

```
=> \c locks_rows

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

=> BEGIN;

BEGIN

=> SELECT txid_current(), pg_backend_pid();

 txid_current | pg_backend_pid
--------------+----------------
       695925 |         126145
(1 row)


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

Блокировки для первой транзакции:

```
=> SELECT * FROM locks WHERE pid = 125866;

  pid   |   locktype    |  lockid  |       mode       | granted
--------+---------------+----------+------------------+---------
 125866 | relation      | pg_locks | AccessShareLock  | t
 125866 | relation      | locks    | AccessShareLock  | t
 125866 | relation      | accounts | RowExclusiveLock | t
 125866 | virtualxid    | 6/6      | ExclusiveLock    | t
 125866 | transactionid | 695923   | ExclusiveLock    | t
(5 rows)
```

- Тип relation для pg_locks и locks в режиме AccessShareLock — устанавливаются на читаемые отношения.
- Тип relation для accounts в режиме RowExclusiveLock — устанавливается на изменяемое отношение.
- Типы virtualxid и transactionid в режиме ExclusiveLock — удерживаются каждой транзакцией для самой себя.

Блокировки для второй транзакции:

```
=> SELECT * FROM locks WHERE pid = 126040;
```

```
  pid   |   locktype    |    lockid    |       mode       | granted
--------+---------------+--------------+------------------+---------
 126040 | relation      | accounts     | RowExclusiveLock | t
 126040 | virtualxid    | 3/15         | ExclusiveLock    | t
 126040 | tuple         | accounts:0,1 | ExclusiveLock    | t
 126040 | transactionid | 695924       | ExclusiveLock    | t
 126040 | transactionid | 695923       | ShareLock        | f
(5 rows)
```

По сравнению с первой транзакцией:

- Блокировки для pg_locks и locks отсутствуют, так как вторая транзакция не обращалась к этим отношениям.
- Транзакция ожидает получение блокировки типа transactionid в режиме ShareLock для первой транзакции.
- Удерживается блокировка типа tuple для обновляемой строки.

Блокировки для третьей транзакции:

```
=> SELECT * FROM locks WHERE pid = 126145;
```

```
  pid   |   locktype    |    lockid    |       mode       | granted
--------+---------------+--------------+------------------+---------
 126145 | relation      | accounts     | RowExclusiveLock | t
 126145 | virtualxid    | 4/5          | ExclusiveLock    | t
 126145 | tuple         | accounts:0,1 | ExclusiveLock    | f
 126145 | transactionid | 695925       | ExclusiveLock    | t
(4 rows)
```

Транзакция ожидает получение блокировки типа tuple для обновляемой строки.

Общую картину текущих ожиданий можно увидеть в представлении pg_stat_activity. Для удобства можно добавить и информацию о блокирующих процессах:

```
=> SELECT pid, wait_event_type, wait_event, pg_blocking_pids(pid)
FROM pg_stat_activity
WHERE backend_type = 'client backend';
```

```
  pid   | wait_event_type |  wait_event   | pg_blocking_pids
--------+-----------------+---------------+------------------
 126040 | Lock            | transactionid | {125866}
 126145 | Lock            | tuple         | {126040}
 125866 |                 |               | {}
(3 rows)
```

```
=> ROLLBACK;
```

```
ROLLBACK
```

```
=> ROLLBACK;
```

```
UPDATE 1
ROLLBACK
```

```
   => ROLLBACK;
```

```
   UPDATE 1
   ROLLBACK
```

2. Взаимоблокировка трех транзакций

Воспроизведем взаимоблокировку трех транзакций.

```
=> BEGIN;
```

```
BEGIN
```

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

```
UPDATE 1
```

```
=> BEGIN;
```

```
BEGIN
```

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

```
UPDATE 1
```

```
   => BEGIN;
```

```
   BEGIN
```

```
   => UPDATE accounts SET amount = amount - 100.00 WHERE acc_no = 3;
```

```
   UPDATE 1
```

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

```
=> UPDATE accounts SET amount = amount + 100.00 WHERE acc_no = 3;
```

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

```
=> COMMIT;
```

```
=> COMMIT;
```

```
   => COMMIT;
```

```
ERROR:  deadlock detected
DETAIL:  Process 125866 waits for ShareLock on transaction 695927; blocked by process 126040.
Process 126040 waits for ShareLock on transaction 695928; blocked by process 126145.
Process 126145 waits for ShareLock on transaction 695926; blocked by process 125866.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,2) in relation "accounts"
ROLLBACK
```

```
UPDATE 1
COMMIT
```

```
   UPDATE 1
   COMMIT
```

Вот какую информацию о взаимоблокировке можно получить из журнала:

```
postgres$ tail -n 10 /var/log/postgresql/postgresql-13-main.log
```

```
2022-12-16 19:58:39.407 MSK [125866] student@locks_rows ERROR:  deadlock detected
2022-12-16 19:58:39.407 MSK [125866] student@locks_rows DETAIL:  Process 125866 waits for ShareLock on transaction 695927; blocked by process 126040.
        Process 126040 waits for ShareLock on transaction 695928; blocked by process 126145.
        Process 126145 waits for ShareLock on transaction 695926; blocked by process 125866.
        Process 125866: UPDATE accounts SET amount = amount + 100.00 WHERE acc_no = 2;
        Process 126040: UPDATE accounts SET amount = amount + 100.00 WHERE acc_no = 3;
        Process 126145: UPDATE accounts SET amount = amount + 100.00 WHERE acc_no = 1;
2022-12-16 19:58:39.407 MSK [125866] student@locks_rows HINT:  See server log for query details.
2022-12-16 19:58:39.407 MSK [125866] student@locks_rows CONTEXT:  while updating tuple (0,2) in relation "accounts"
2022-12-16 19:58:39.407 MSK [125866] student@locks_rows STATEMENT:  UPDATE accounts SET amount = amount + 100.00 WHERE acc_no = 2;
```

3. Взаимоблокировка двух операций UPDATE

Команда UPDATE блокирует строки по мере их обновления. Это происходит не одномоментно.

Поэтому если одна команда будет обновлять строки в одном порядке, а другая — в другом, они могут взаимозаблокироваться. Это может произойти, если для команд будут построены разные планы выполнения, например, одна будет читать таблицу последовательно, а другая — по индексу.

Получить такую ситуацию непросто даже специально, в реальной работе она маловероятна. Проиллюстрировать ее проще всего с помощью курсоров, поскольку это дает возможность управлять порядком чтения.

```
=> BEGIN;
```

```
BEGIN
```

```
=> DECLARE c1 CURSOR FOR
SELECT * FROM accounts ORDER BY acc_no
FOR UPDATE;
```

```
DECLARE CURSOR
```

```
   => BEGIN;
```

```
   BEGIN
```

```
   => DECLARE c2 CURSOR FOR
   SELECT * FROM accounts ORDER BY acc_no DESC -- в обратную сторону
   FOR UPDATE;
```

```
   DECLARE CURSOR
```

```
=> FETCH c1;
```

```
 acc_no | amount
--------+---------
      1 | 1100.00
(1 row)
```

```
   => FETCH c2;
```

```
    acc_no | amount
   --------+---------
         3 | 3000.00
   (1 row)
```

```
=> FETCH c1;
```

```
 acc_no | amount
--------+---------
      2 | 1900.00
(1 row)
```

```
   => FETCH c2;
```

Вторая команда ожидает блокировку...

```
=> FETCH c1;
```

Произошла взаимоблокировка. И через некоторое время:

```
 acc_no | amount
--------+---------
      3 | 3000.00
(1 row)
```

```
   ERROR:  deadlock detected
   DETAIL:  Process 126145 waits for ShareLock on transaction 695929; blocked by process 126040.
   Process 126040 waits for ShareLock on transaction 695930; blocked by process 126145.
   HINT:  See server log for query details.
   CONTEXT:  while locking tuple (0,8) in relation "accounts"
```

```
=> COMMIT;
```

```
   => COMMIT;
```