

Архитектура Блокировки

Postgres
PROFESSIONAL

16

Авторские права

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

edu@postgrespro.ru

Отказ от ответственности

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Общая информация о блокировках

Блокировки отношений и других объектов

Блокировки на уровне строк

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. Когда процесс перестает нуждаться в ресурсе, 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, как работает механизм блокировок.

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 этого же типа — рекомендательные блокировки — упоминался в теме «Обработка ошибок» курса DEV1.

Типы ресурсов в pg_locks

virtualxid — виртуальный номер транзакции

transactionid — настоящий номер транзакции

Режимы

исключительный

разделяемый

Транзакция удерживает исключительную блокировку собственного номера

способ дождаться завершения транзакции

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

Блокировки объектов хранятся в общей памяти сервера, поэтому их количество ограничено. Все такие блокировки можно посмотреть в представлении pg_locks: они устроены одинаково и отличаются только типами ресурсов и режимами блокирования. Номерам транзакций соответствуют типы ресурсов **Transactionid** и **Virtualxid**.

Каждой транзакции при старте назначается сначала виртуальный номер, который используется, пока транзакция только читает данные. Это сделано для экономии настоящих номеров — виртуальный номер никак не учитывается в правилах видимости. Его можно выдать быстро, без обращения к общей памяти (требуется лишь уникальность номеров всех активных транзакций).

Уникальный настоящий номер выдается счетчиком XID в тот момент, когда транзакция изменяет какие-либо данные. Он нужен, чтобы отслеживать статус пишущей транзакции (активна, зафиксирована или оборвана); такой номер можно записывать в поля xmin и xmax заголовка версий строк.

Каждая транзакция удерживает исключительную блокировку своих номеров: и виртуального, и настоящего, если он есть. Это дает простой способ дождаться окончания какой-либо транзакции: надо запросить блокировку ее номера. Если блокировку сразу получить не удастся, то запрашивающий процесс заснет и будет разбужен только по завершении транзакции — когда блокировка освободится.

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
| -----
|          137556
| (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 = 137556;
```

```
locktype | virtxid | xid | mode | granted
-----+-----+----+-----+-----
virtualxid | 3/12 | | ExclusiveLock | t
(1 row)
```

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

Каждой транзакции сразу 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 pg_current_xact_id();
```

```
| pg_current_xact_id
| -----
|          805
| (1 row)
```

```
=> SELECT locktype, virtualxid AS virtxid, transactionid AS xid,
       mode, granted
FROM pg_locks
WHERE pid = 137556;
```

```
locktype | virtxid | xid | mode | granted
-----+-----+----+-----+-----
virtualxid | 3/12 | | ExclusiveLock | t
transactionid | | 805 | ExclusiveLock | t
(2 rows)
```

Теперь транзакция удерживает исключительную блокировку обоих номеров.

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, MERGE	
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, REINDEX, VACUUM FULL, LOCK TABLE, ALTER TABLE, REFRESH MAT. VIEW	

7

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

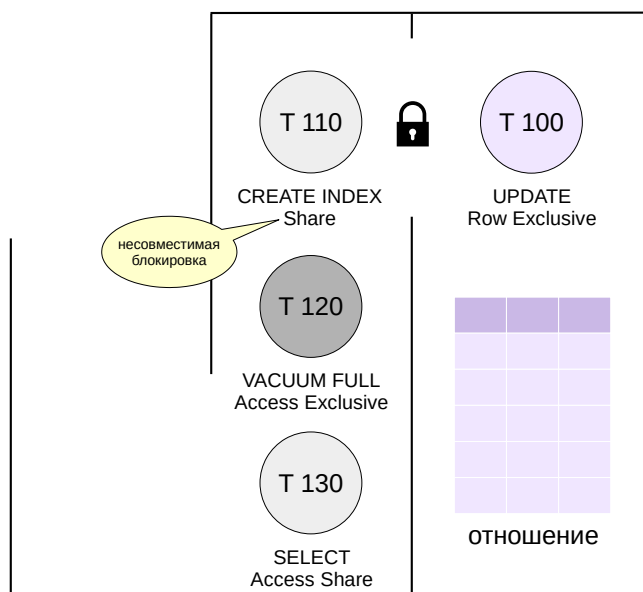
Для них определено целых 8 различных режимов, которые показаны на слайде вместе с примерами команд SQL, использующих эти режимы. Матрица совместимости здесь не показана из-за ее большого размера, но она приведена в документации:

<https://postgrespro.ru/docs/postgresql/16/explicit-locking#LOCKING-TABLES>

Такое количество режимов существует 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 совместим с изменением данных в таблице. Например, команда CREATE INDEX заблокирует команды INSERT, UPDATE и DELETE (и наоборот). Поэтому существует команда CREATE INDEX CONCURRENTLY, использующая режим Share Update Exclusive, который совместим с такими изменениями (за счет этого команда выполняется дольше).

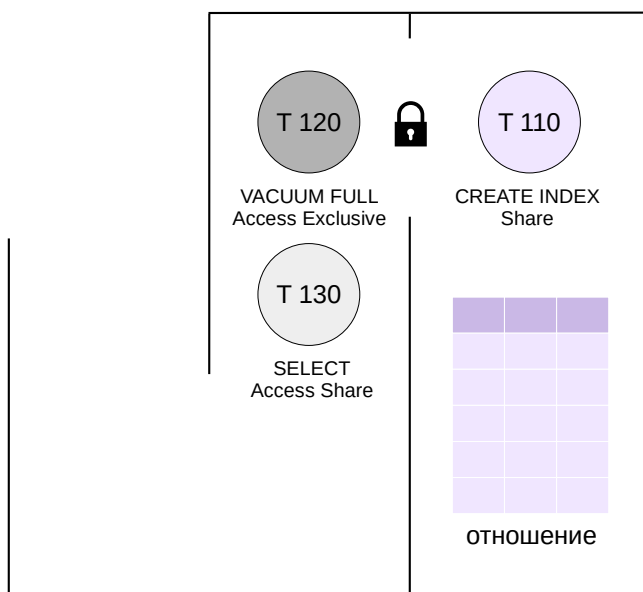


Блокировки объектов используют «честную» очередь ожидания. Это означает, что запросы блокировок обслуживаются один за другим в порядке их поступления.

Пусть, как на рисунке, транзакция T100 выполняет команду UPDATE, при этом на отношение будет установлена блокировка с режимом Row Exclusive.

Транзакция T110, выполняющая CREATE INDEX, встает в очередь за T100, поскольку запрашивает режим Share, не совместимый с Row Exclusive. Если следом придет транзакция T120 с командой VACUUM FULL (ей нужен режим Access Exclusive, не совместимый ни с каким другим), она встанет в очередь за CREATE INDEX.

А еще одна транзакция с обычным запросом SELECT (требуется Access Share, совместимый с выполняющейся сейчас командой UPDATE) тоже честно встанет в очередь за VACUUM FULL.



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

Блокировка отношений

Создадим таблицу банковских счетов с тремя строками:

```
=> 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 = 137556;
```

locktype	relation	virtualxid	xid	mode	granted
relation	accounts			AccessShareLock	t
virtualxid		3/12		ExclusiveLock	t
transactionid			805	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 
-----
        137944 
(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 = 137944;
```

locktype	relation	virtualxid	xid	mode	granted
virtualxid		4/5		ExclusiveLock	t
relation	accounts			AccessExclusiveLock	f
transactionid			808	ExclusiveLock	t

(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 = 137944;
```

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 случае — несколько номеров)...

```
=> SELECT pg_blocking_pids(137944);
```

pg_blocking_pids
{137556}

(1 row)

...и посмотреть информацию о сеансах, к которым они относятся:

```
=> SELECT * FROM pg_stat_activity
WHERE pid = ANY(pg_blocking_pids(137944)) \gx
```

-[RECORD 1]-----	
datid	16560
datname	arch_locks
pid	137556
leader_pid	
usesysid	16384
username	student
application_name	psql
client_addr	
client_hostname	
client_port	-1
backend_start	2024-08-14 11:03:24.086527+03
xact_start	2024-08-14 11:03:24.138557+03
query_start	2024-08-14 11:03:24.497056+03
state_change	2024-08-14 11:03:24.501821+03
wait_event_type	Client
wait_event	ClientRead
state	idle in transaction
backend_xid	805
backend_xmin	
query_id	
query	SELECT * FROM accounts;
backend_type	client backend

После завершения транзакции все блокировки снимаются и таблица удаляется:

```
=> COMMIT;

COMMIT

|| DROP TABLE

|| => COMMIT;

|| COMMIT
```

Разделяемые и исключительные блокировки
«Нечестная» очередь ожидания

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) и 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 из вариантов решения — повышение уровня (эскалация): 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 исключительно в заголовке версии строки внутри страницы данных. Там она представлена номером блокирующей транзакции (xmax) и дополнительными информационными битами.

За счет этого можно установить неограниченное количество блокировок уровня строки, и это не требует дополнительных ресурсов и не снижает производительность системы.

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

Существует 4 режима, в которых можно заблокировать строку (в версии строки режим указывается 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 прочесть 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 самим PostgreSQL при проверке внешних ключей.

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

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/16/explicit-locking#LOCKING-ROWS>

Блокировка строк

Снова создадим таблицу счетов, но теперь 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 pg_current_xact_id();
|
| pg_current_xact_id
| -----
|                811
| (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
-----+-----+-----
 811 |      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. В таком случае номера транзакций хранятся в отдельной структуре, называемой мультитранзакцией, а в xmax помещается ссылка на нее. Но мы не будем вдаваться в детали реализации блокировок строк, а просто воспользуемся расширением pgrowlocks:

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

```
CREATE EXTENSION
```

```
=> SELECT locked_row, xids, modes, pids FROM pgrowlocks('accounts');
```

```
locked_row | xids  | modes          | pids
-----+-----+-----+-----
(0,1)      | {811} | {"No Key Update"} | {137556}
(1 row)
```

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 locked_row, xids, modes, pids FROM pgrowlocks('accounts');
|
| locked_row | xids  | modes          | pids
| -----+-----+-----+-----
| (0,1)      | {811} | {"No Key Update"} | {137556}
| (0,2)      | {811} | {Update}         | {137556}
| (2 rows)
```

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 pg_current_xact_id();
||
||   pg_current_xact_id
||   -----
||                   813
|| (1 row)
||
|| => SELECT locked_row, xids, modes, pids FROM pgrowlocks('accounts');
||
|| locked_row | xids      | modes                                | pids
|| -----+-----+-----+-----
|| (0,1)      | {811,813} | {"No Key Update", "Key Share"}      | {137556,137944}
|| (0,2)      | {811}      | {Update}                            | {137556}
|| (0,3)      | {813}      | {"For Share"}                       | {137944}
|| (3 rows)
||
||
|| => 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 - строка уже заблокирована. Но при успехе `SELECT ... FOR UPDATE` в этой транзакции можно было бы далее изменять заблокированную строку.

```
| => ROLLBACK;
```

```
| ROLLBACK
```

Другой подход к блокировке строк предоставляет предложение `SKIP LOCKED`. Запросим блокировку одной строки, не указывая конкретный номер 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 таймаут ожидания (по умолчанию он не задан и ожидание будет бесконечным):

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

```
| SET
```

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

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

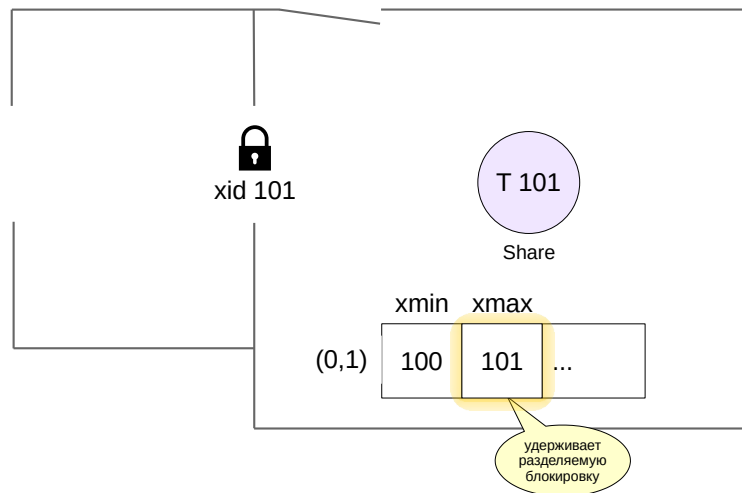
Получаем ошибку без длительного ожидания освобождения ресурса.

```
| => RESET lock_timeout;
```

```
| RESET
```

```
| => ROLLBACK;
```

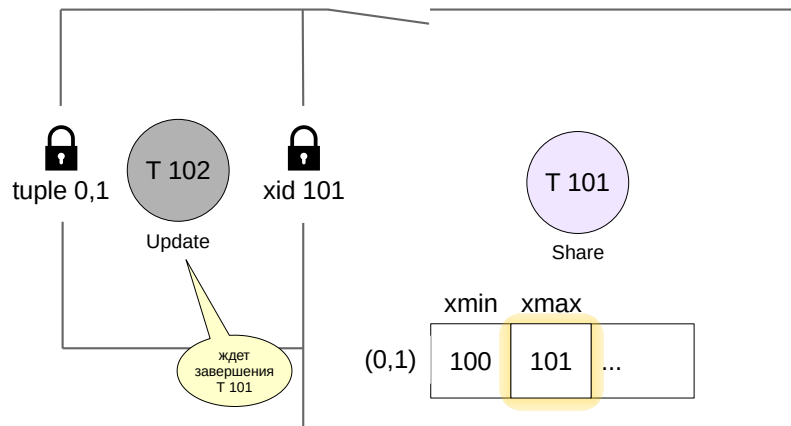
```
| ROLLBACK
```



Для эффективной организации очереди ожидания используется механизм блокировок объектов, при этом информация хранится в оперативной памяти и видна в `pg_locks`.

Допустим, транзакции с номером 101 удалось заблокировать строку в разделяемом режиме Share. Информация о блокировке записывается в заголовок версии строки — в поле `xmax` появляется номер этой транзакции.

«Очередь» ожидания



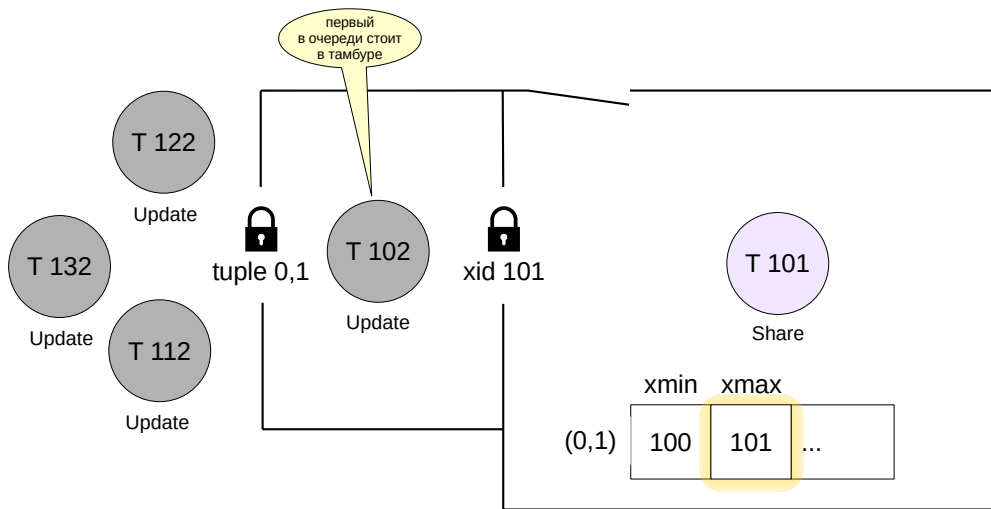
16

Другая транзакция 102, желая получить блокировку, обращается к строке и видит, что строка уже заблокирована. Если режимы блокировок конфликтуют, транзакция 102 должна встать в очередь, чтобы система «разбудила» ее, когда блокировка освободится.

Поскольку каждая транзакция удерживает исключительную блокировку своего номера, транзакция 102 запрашивает блокировку номера транзакции 101, чтобы дождаться ее завершения.

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

«Очередь» ожидания



17

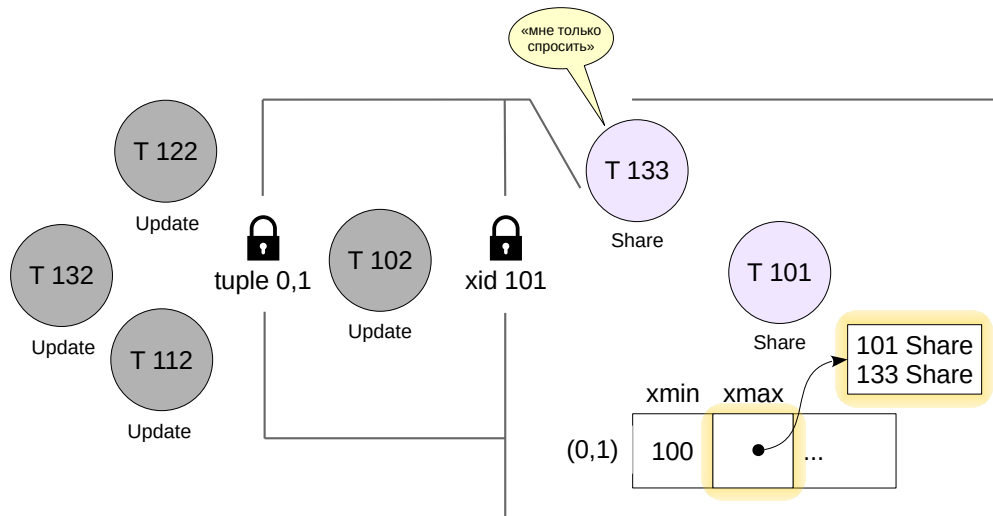
Если появляются другие транзакции, конфликтующие с текущей блокировкой версии строки (в нашем примере — 112, 122, 132), первым делом они пытаются захватить блокировку типа tuple для этой версии.

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

Если бы все прибывающие транзакции занимали очередь непосредственно за транзакцией 101, при освобождении блокировки возникала бы ситуация гонки. При неудачном стечении обстоятельств транзакция 102 могла бы ждать блокировку вечно. Двухуровневая схема блокирования немного упорядочивает подобную конкуренцию.

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.

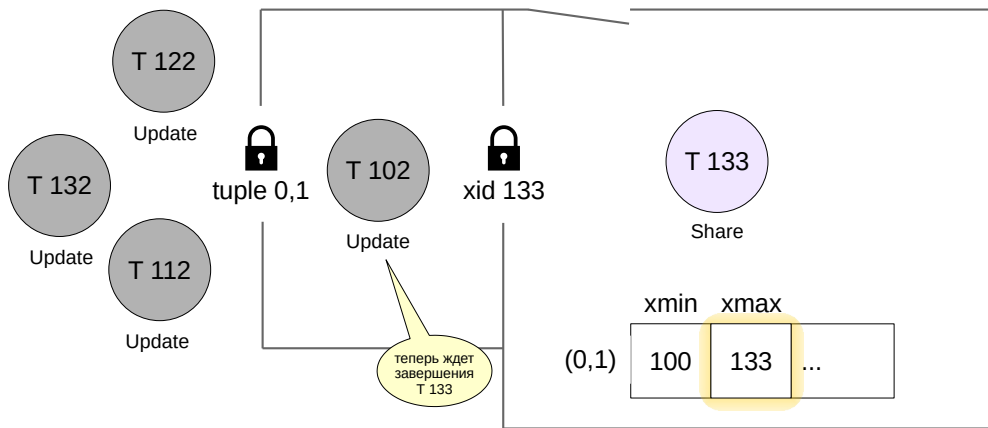
«Очередь» ожидания



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. В этом случае список номеров блокирующих транзакций будет вынесен в *мультитранзакцию*, а в поле xmax строки запишется номер этой мультитранзакции.

«Очередь» ожидания



19

Когда транзакция 101 завершится, транзакция 102 будет разбужена, но не сможет захватить блокировку строки, поскольку строка все еще заблокирована транзакцией 133. Транзакция 102 будет вынуждена снова «заснуть».

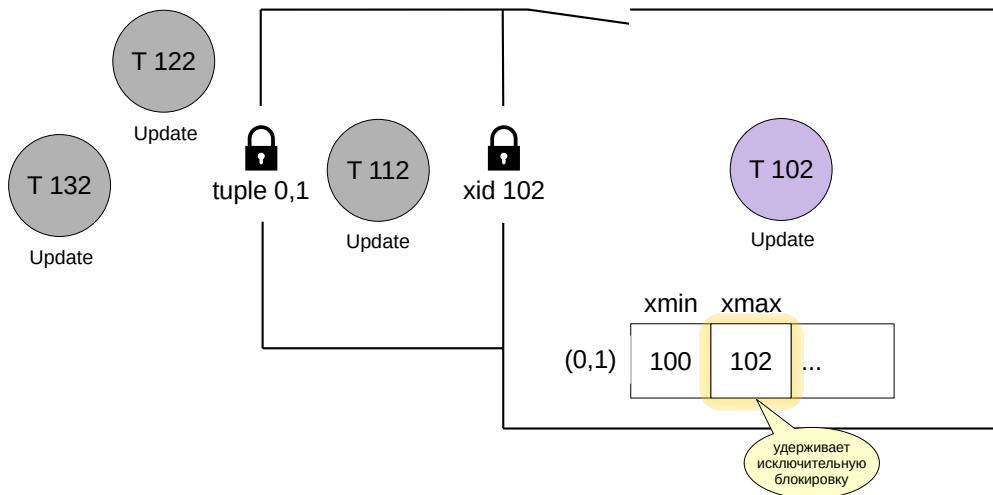
PostgreSQL использует разделяемые блокировки для внутренних нужд, в частности, при проверке внешних ключей. В прикладном коде можно явно запросить разделяемую блокировку с помощью команды `SELECT ... FOR SHARE` или `SELECT ... FOR KEY SHARE`.

Однако следует помнить, что при постоянном потоке разделяемых блокировок пишущая транзакция в худшем случае может ждать своей очереди бесконечно. Эта 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



20

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* в «очереди».

«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
137512	transactionid			818	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
137512	transactionid			818	ExclusiveLock	t
137556	transactionid			819	ExclusiveLock	t
137556	tuple	0	1		ExclusiveLock	t
137556	transactionid			818	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 (137512,137556) ORDER BY pid;
```

pid	pg_blocking_pids
137512	{}
137556	{137512}

(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
137512	transactionid			818	ExclusiveLock	t
137556	transactionid			819	ExclusiveLock	t
137556	tuple	0	1		ExclusiveLock	t
137556	transactionid			818	ShareLock	f
137944	transactionid			820	ExclusiveLock	t
137944	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 (137512,137556,137944) ORDER BY pid;
```

pid	pg_blocking_pids
137512	{}
137556	{137512}
137944	{137556}

(3 rows)

Если теперь первая транзакция успешно завершается...

```
=> COMMIT;
```

```
COMMIT
```

```
| UPDATE 1
```

...выполняется UPDATE и первая версия становится неактуальной.

```
=> 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
137556	transactionid			819	ExclusiveLock	t
137944	transactionid			820	ExclusiveLock	t
137944	transactionid			819	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 (137556,137944) ORDER BY pid;
```

pid	pg_blocking_pids
137556	{}
137944	{137556}

(2 rows)

```
| => COMMIT;
```

```
| COMMIT
```

```
|| UPDATE 1
```

```
|| => COMMIT;
```

```
|| COMMIT
```

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

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 упоминались в теме «Обработка ошибок» курса DEV1.

<https://postgrespro.ru/docs/postgresql/16/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,
- вторая транзакция удерживает блокировку объектов, необходимых первой транзакции для продолжения работы.

В общем случае может произойти взаимоблокировка более чем двух транзакций.

1. Блокировки при чтении строки по первичному ключу

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

```
CREATE DATABASE
```

```
=> \c arch_locks
```

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

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

```
pg_backend_pid
-----
          177659
(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 arch_locks
```

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

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

```
| pg_backend_pid
| -----
|          177792
| (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 = 177792;
```

locktype	relation	virtualxid	xid	mode	granted
relation	accounts_pkey			AccessShareLock	t
relation	accounts			AccessShareLock	t
virtualxid		3/15		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 = 177792;

```

```

locktype | virtualxid | objid | mode | granted
-----+-----+-----+-----+-----
virtualxid | 3/16 | | 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     | 811
multi      | f
xids       | {811}
modes      | {"For Key Share"}
pids       | {177792}

```

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.

```
| ERROR: deadlock detected  
| DETAIL: Process 177792 waits for ShareLock on transaction 813; blocked by process 177659.  
| Process 177659 waits for ShareLock on transaction 814; blocked by process 177792.  
| HINT: See server log for query details.  
| CONTEXT: while updating tuple (0,1) in relation "accounts"  
  
UPDATE 1
```

```
=> COMMIT;  
  
COMMIT  
  
| => 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.