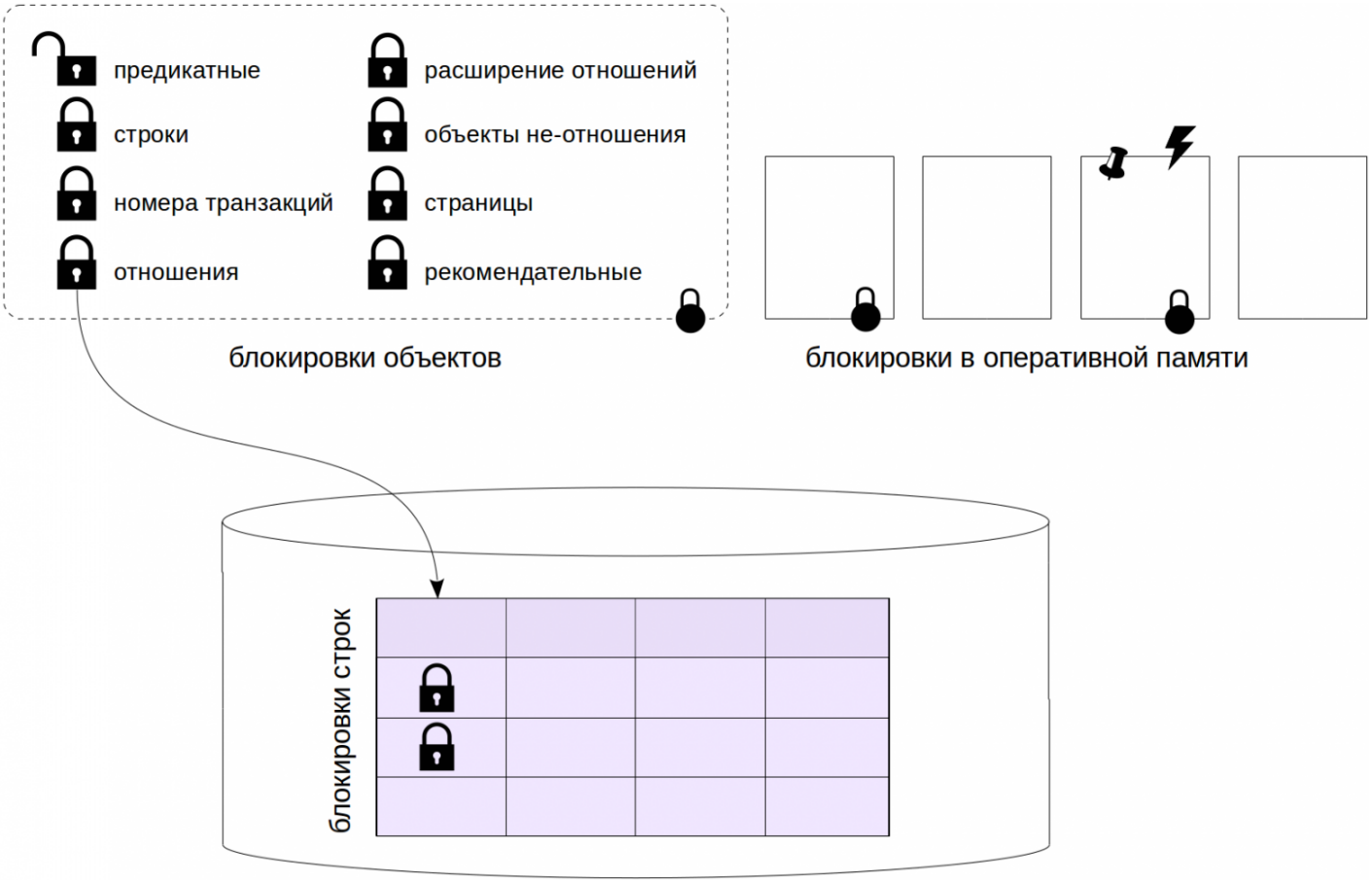
?????????? ?????????



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

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

Блокировки используются, чтобы упорядочить конкурентный доступ к разделяемым ресурсам.

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

Если нет конкуренции, то нет нужды и в блокировках (например, общий буферный кеш

требует блокировок, а локальный – нет).

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

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

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

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

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

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

Защищаемым ресурсом в принципе может быть все, что угодно, лишь бы этот ресурс можно было однозначно идентифицировать и сопоставить ему адрес блокировки.

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

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

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

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

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

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

По времени использования блокировки можно разделить на длительные и короткие.

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

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

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

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

В PostgreSQL используются разные виды блокировок.

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

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

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

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

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

Итак, мы начинаем с блокировок уровня объектов. Под объектом здесь понимаются в первую очередь *отношения* (relations), то есть таблицы, индексы, последовательности, материализованные представления, но также и некоторые другие сущности. Эти блокировки обычно защищают объекты от одновременного изменения или от использования в то время, когда объект изменяется, но также и для других нужд.

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

??????????

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

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

Все блокировки можно посмотреть в представлении pg_locks.

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

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

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

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

- **relation**

  Блокировки отношений.
- **transactionid** и **virtualxid**

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

- **tuple**

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

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

- **extend**

  Используется при добавлении страниц к файлу какого-либо отношения.

- **object**

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

- **page**

  Блокировка страницы, используется нечасто и только некоторыми типами индексов.

- **advisory**

  Рекомендательная блокировка, устанавливается пользователем вручную.

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

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

odwcnenfvjv_ikwmqw8x-vaexqi.png

??????

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

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

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

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

Некоторые комментарии:

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

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

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

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

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

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

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

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

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

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

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

```
=> SELECT locktype, relation::REGCLASS, virtualxid AS virtxid, transactionid AS xid, mode,
granted
FROM pg_locks WHERE pid = 4746;
```

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

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

Теперь обновим строку таблицы. Как изменится ситуация?

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

```
=> \g
```

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

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

Теперь попробуем в еще одном сеансе создать индекс по таблице.

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

```
||      pg_backend_pid
||     ----------------
||                4782
||     (1 row)
```

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

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

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

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

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

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

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

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

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

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

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

После завершения транзакции блокировки снимаются и индекс создается.

```
|  => COMMIT;
```

```
|  COMMIT
```

```
||     CREATE INDEX
```

? ???????!..

Чтобы лучше представить, к чему приводит появление несовместимой блокировки,

посмотрим, что произойдет, если в процессе работы системы выполнить команду VACUUM FULL.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
=> COMMIT;
```

```
COMMIT
```

```
|  LOCK TABLE
```

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

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

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

```
|  => COMMIT;
```

```
|  COMMIT
```

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

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

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

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

С одним способом мы уже познакомились: в момент возникновения долгой блокировки мы можем выполнить запрос к представлению pg_locks, посмотреть на блокируемые и

блокирующие транзакции (функция pg_blocking_pids) и расшифровывать их при помощи pg_stat_activity.

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

Попробуем.

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

Значение параметра *deadlock_timeout* по умолчанию равно одной секунде:

```
=> SHOW deadlock_timeout;
```

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

Воспроизведем блокировку.

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

```
UPDATE 1
```

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

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

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

```
COMMIT
```

Теперь и вторая транзакция может завершиться.

```
|  UPDATE 1
```

```
|  => COMMIT;
```

```
|  COMMIT
```

И вся важная информация попала в журнал:

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

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

```
2019-08-07 15:26:30.836 MSK [5898] student@test LOG:  process 5898 acquired ShareLock on
transaction 529427 after 1009.536 ms
2019-08-07 15:26:30.836 MSK [5898] student@test CONTEXT:  while updating tuple (0,4) in
relation "accounts"
2019-08-07 15:26:30.836 MSK [5898] student@test STATEMENT:  UPDATE accounts SET amount =
amount + 100.00 WHERE acc_no = 1;
```

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

- Блокировки в PostgreSQL: 1. Блокировки отношений

Revision #2
Created 2024-06-25 11:43:18 UTC by Антон Сергеевич Абраменко
Updated 2024-06-25 11:49:36 UTC by Антон Сергеевич Абраменко