Практические работы по дисциплине «Технологии обработки транзакций клиент-серверных приложений» направления подготовки бакалавриата 09.03.04 «Программная инженерия»

Практическая работа №8

Транзакции. Блокировки. Блокировки строк

Теория для понимания практики:

Темы

- Исключительные и разделяемые блокировки строк
- Мультитранзакции и заморозка
  - Проблема переполнения счетчика транзакций
  - Заморозка версий строк и правила видимости
  - Настройка автоочистки для выполнения заморозки
- Реализация очереди ожидания
- Взаимоблокировки

Блокировки на уровне строк

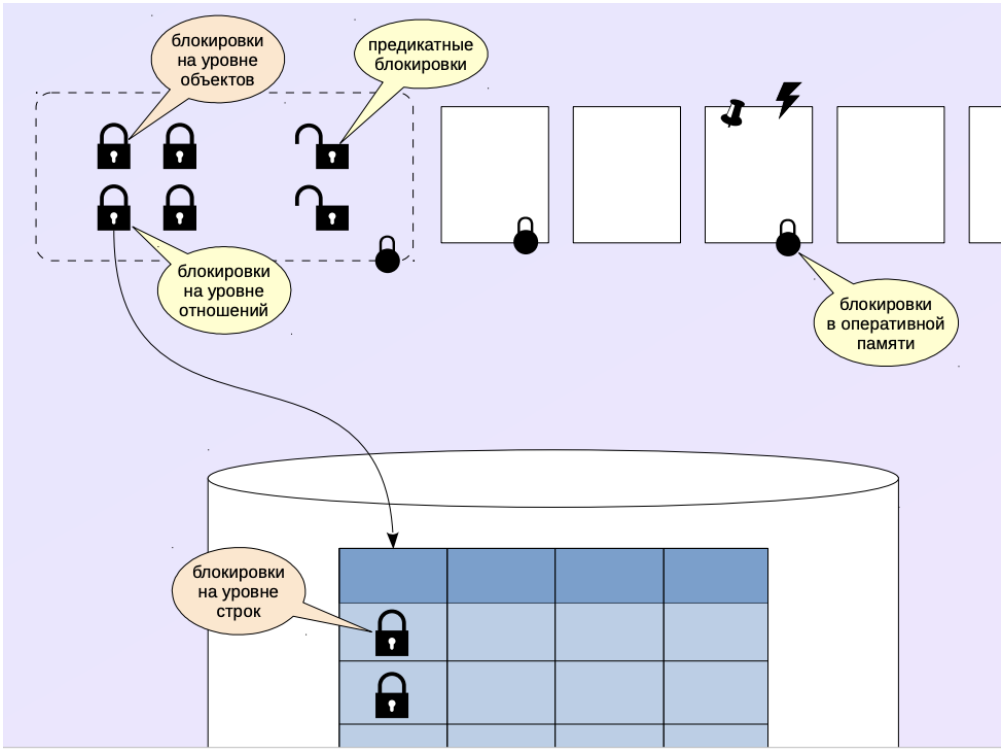

Рисунок 1 – Блокировки на уровне строк

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, то 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 блокировок уровня строки может быть неограниченное количество. 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 так же не снижает производительность системы.

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

Исключительный режим

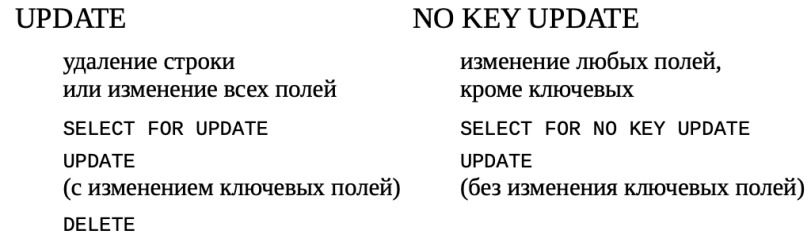

Рисунок 2 – Исключительные режимы

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.

FOR UPDATE

В режиме FOR UPDATE строки, выданные оператором SELECT, блокируются как для изменения. При этом они защищаются от блокировки, изменения и удаления другими транзакциями до завершения текущей. То есть другие транзакции, пытающиеся выполнить UPDATE, DELETE, SELECT FOR UPDATE, SELECT FOR NO KEY UPDATE, SELECT FOR SHARE или SELECT FOR KEY SHARE с этими строками, будут заблокированы до завершения текущей транзакции; и наоборот, команда SELECT FOR UPDATE будет ожидать окончания параллельной транзакции, в которой выполнилась одна из этих команд с той же строкой, а затем установит блокировку и вернёт изменённую строку (или не вернёт, если она была удалена). Однако в транзакции REPEATABLE READ или SERIALIZABLE возникнет ошибка, если блокируемая строка изменилась с момента начала транзакции.

Режим блокировки FOR UPDATE также запрашивается на уровне строки любой командой DELETE и командой UPDATE, изменяющей значения определённых столбцов. В настоящее время блокировка с UPDATE касается столбцов, по которым создан уникальный индекс, применимый в качестве

внешнего ключа (так что на частичные индексы и индексы выражений это не распространяется), но в будущем это может поменяться.

FOR NO KEY UPDATE

Действует подобно FOR UPDATE, но запрашиваемая в этом режиме блокировка слабее: она не будет блокировать команды SELECT FOR KEY SHARE, пытающиеся получить блокировку тех же строк. Этот режим блокировки также запрашивается любой командой UPDATE, которая не требует блокировки FOR UPDATE.

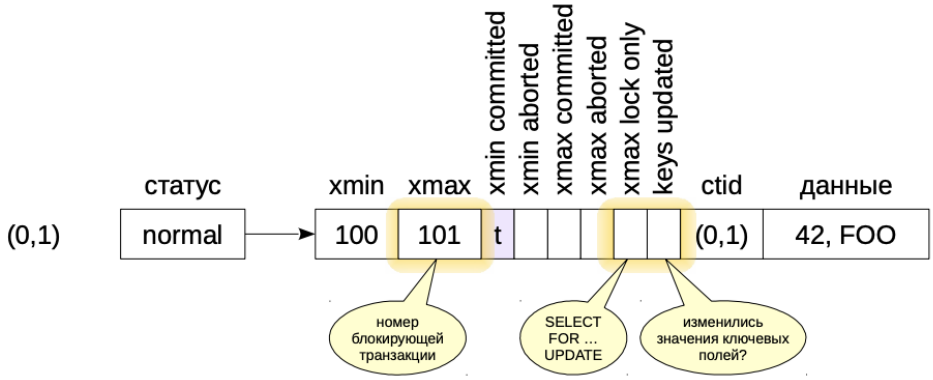

Рисунок 3 – поля актуальной версии

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.

Разделяемый режим

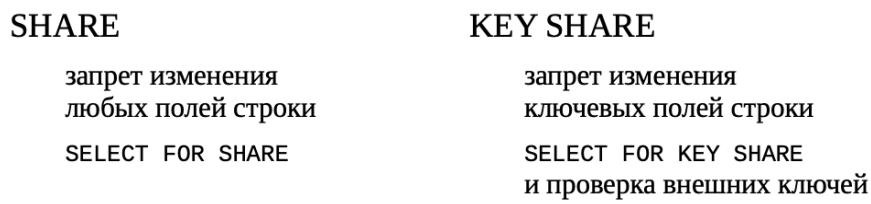

Рисунок 4 – Разделяемые режимы

Запрашиваемый режим блокировки	Текущий режим блокировки			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

Рисунок 5 – Конфликтующие блокировки на уровне строк

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 на рисунке 5. 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).

FOR SHARE

Действует подобно FOR NO KEY UPDATE, за исключением того, что для каждой из полученных строк запрашивается разделяемая, а не исключительная блокировка. Разделяемая блокировка не позволяет другим

транзакциям выполнять с этими строками UPDATE, DELETE, SELECT FOR UPDATE или SELECT FOR NO KEY UPDATE, но допускает SELECT FOR SHARE и SELECT FOR KEY SHARE.

FOR KEY SHARE

Действует подобно FOR SHARE, но устанавливает более слабую блокировку: блокируется SELECT FOR UPDATE, но не SELECT FOR NO KEY UPDATE. Блокировка разделяемого ключа не позволяет другим транзакциям выполнять команды DELETE и UPDATE, только если они меняют значение ключа (но не другие UPDATE), и при этом допускает выполнение команд SELECT FOR NO KEY UPDATE, SELECT FOR SHARE и SELECT FOR KEY SHARE.

PostgreSQL не держит информацию об изменённых строках в памяти, так что никаких ограничений на число блокируемых строк нет. Однако блокировка строки может повлечь запись на диск, например, если SELECT FOR UPDATE изменяет выбранные строки, чтобы заблокировать их, при этом происходит запись на диск.

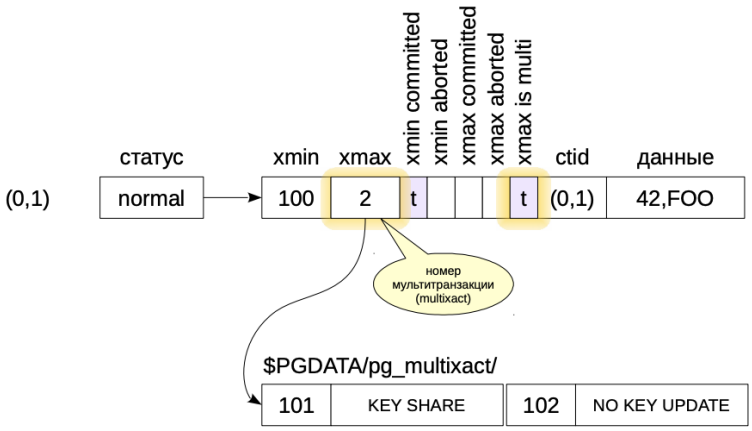

Рисунок 6 – поля актуальной версии

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, то 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 рассматривается ниже для обычных транзакций.

Переполнение счетчика

меньшие номера — прошлое, бóльшие — будущее

разрядность счетчика — 32 бита, что делать при переполнении?

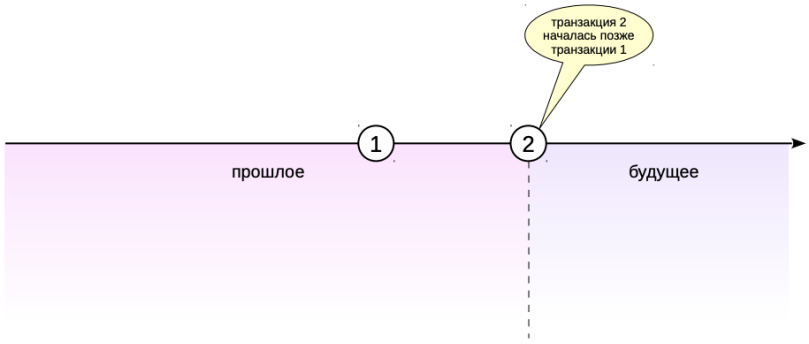

Рисунок 7 – Переполнение счетчика

Кроме освобождения места в страницах, очистка выполняет также задачу по предотвращению проблем, связанных с переполнением счетчика транзакций.

Под номер транзакции в PostgreSQL выделено 32 бита. Это довольно большое число (около 4 млрд), но при активной работе сервера оно вполне может быть исчерпано. Например, при нагрузке 1000 транзакций в секунду это произойдет всего через полтора месяца непрерывной работы.

Механизм многоверсионности полагается на последовательную нумерацию транзакций — из двух транзакций транзакция с меньшим номером считается начавшейся раньше. Понятно, что нельзя просто обнулить счетчик и продолжить нумерацию заново.

Почему под номер транзакции не выделено 64 бита — ведь это полностью исключило бы проблему? Дело в том, что в заголовке каждой версии строки хранятся два номера транзакций — xmin и xmax. Заголовок и так достаточно большой, а увеличение разрядности привело бы к его увеличению еще на 8 байт.

Нумерация по кругу

- пространство номеров транзакций закольцовано
- половина номеров — прошлое, половина — будущее

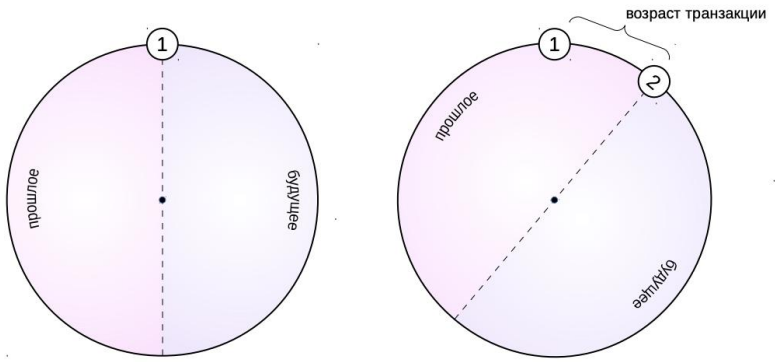

Рисунок 8 – Нумерация по кругу

Поэтому вместо линейной схемы все номера транзакций закольцованы. Для любой транзакции половина номеров «против часовой стрелки» считается принадлежащей прошлому, а половина «по часовой стрелке» — будущему.

Возрастом транзакции называется число транзакций, прошедших с момента ее появления в системе (независимо от того, переходил ли счетчик через ноль или нет).

Проблема видимости

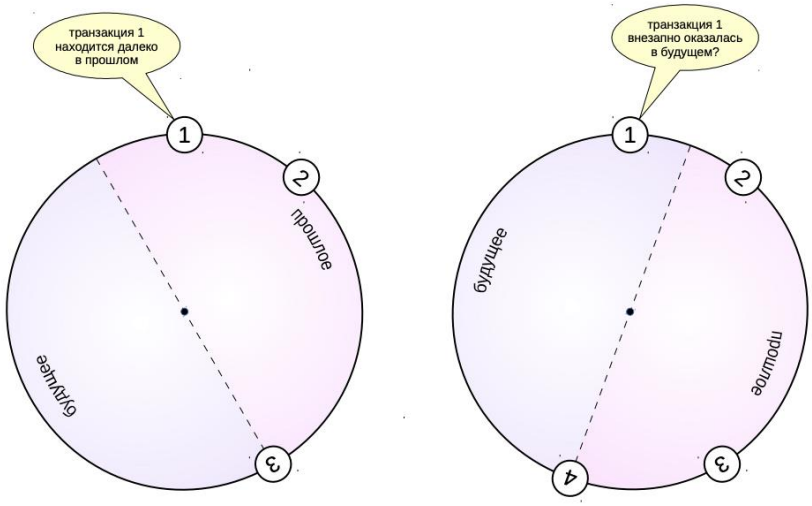

Рисунок 9 – Проблема видимости

В такой закольцованной схеме возникает неприятная ситуация. Транзакция, находившаяся в далеком прошлом (транзакция 1 на слайде), через некоторое время окажется в той половине круга, которая относится к будущему. Это, конечно, нарушает правила видимости и привело бы к проблемам.

Заморозка версий строк

замороженные версии строк считаются «бесконечно старыми» номер транзакции xmin может быть использован заново

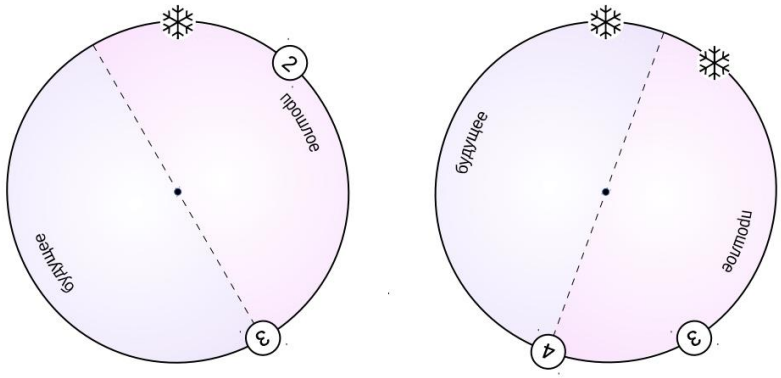

Рисунок 10 – Заморозка версий строк

Чтобы не допустить путешествий из прошлого в будущее, процесс очистки выполняет еще одну задачу. Он находит достаточно старые и «холодные» версии строк (которые видны во всех снимках и изменение которых уже маловероятно) и специальным образом помечает — «замораживает» — их. Замороженная версия строки считается старше любых обычных данных и всегда видна во всех снимках данных. При этом уже не требуется смотреть на номер транзакции xmin, и этот номер может быть безопасно использован заново. Таким образом, замороженные версии строк всегда остаются в прошлом.

Еще одна задача процесса очистки если вовремя не заморозить версии строк, они окажутся в будущем и сервер остановится для предотвращения ошибки

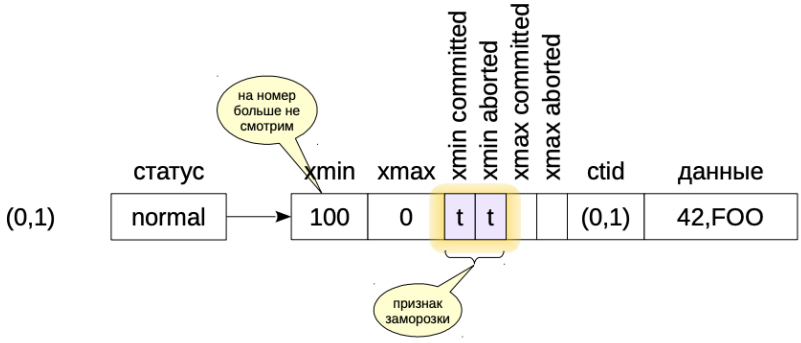

Рисунок 11 – поля актуальной версии

Для того, чтобы пометить номер транзакции xmin как замороженный, выставляются одновременно оба бита-подсказки — бит фиксации и бит отмены.

Заметим, что транзакцию xmax замораживать не нужно. Ее наличие означает, что данная версия строки больше не актуальна. После того, как она перестанет быть видимой в снимках данных, такая версия строки будет очищена.

Многие источники (включая документацию) упоминают специальный номер FrozenTransactionId = 2, которым помечаются замороженные транзакции. Такая система действовала до версии 9.4, но сейчас заменена на биты-подсказки — это позволяет сохранить в версии строки исходный номер транзакции, что удобно для целей поддержки и отладки. Однако транзакции с номером 2 еще могут встретиться в старых системах, даже обновленных до последних версий.

Важно, чтобы версии строк замораживались вовремя. Если возникнет ситуация, при которой еще не замороженная транзакция рискует попасть в будущее, PostgreSQL аварийно остановится. Это возможно в двух случаях: либо транзакция не завершена и, следовательно, не может быть заморожена, либо не сработала очистка.

При запуске сервера транзакция будет автоматически отменена; дальше администратор должен вручную выполнить очистку и после этого система сможет продолжить работать дальше.

Настройка заморозки

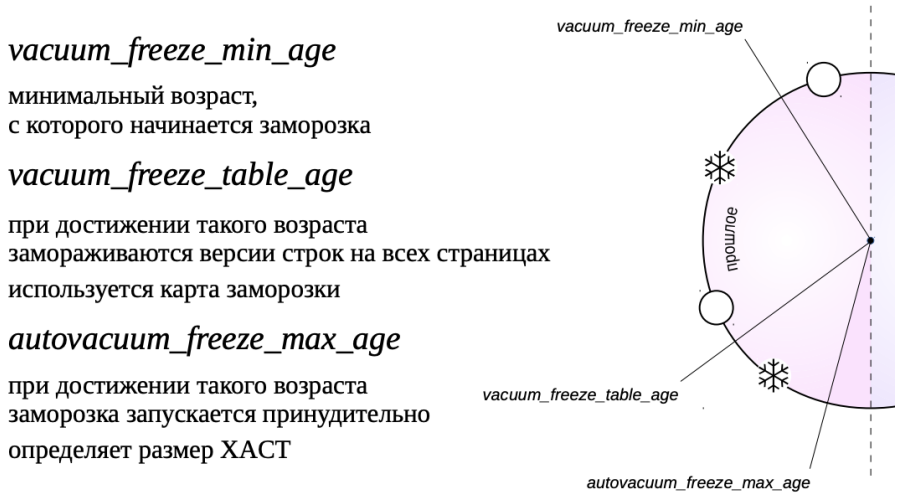

Рисунок 12 – Настройка заморозки

Заморозкой управляют три основных параметра.

vacuum_freeze_min_age определяет минимальный возраст транзакции xmin, при котором версию строки можно замораживать. Чем меньше это значение, тем больше накладных расходов: версии строк могут быть «горячими» и активно меняться — их заморозка пропадет без пользы, а вместо этого придется снова замораживать уже новые версии. Заметим, что очистка просматривает только страницы, не отмеченные в карте видимости. Если на странице остались только актуальные версии, то очистка не придет в такую страницу и не заморозит их.

vacuum_freeze_table_age определяет возраст транзакции, при котором пора выполнять заморозку на всех страницах. Для этого каждая таблица хранит номер транзакции, который был заморожен в прошлый раз (pg_class.relfrozenxid). До версии 9.6 для этого выполнялось полное сканирование таблицы. Начиная с 9.6, в карту видимости была встроена карта заморозки, в которой отмечены страницы с замороженными версиями строк. Так или иначе, заморозка всех страниц выполняется раз в (vacuum_freeze_table_age - vacuum_freeze_min_age) транзакций. Таким образом, большое значение vacuum_freeze_min_age увеличивает накладные расходы.

autovacuum_freeze_max_age определяет возраст транзакции, при котором заморозка будет выполняться принудительно — автоочистка запустится, даже если она отключена. Этот параметр также определяет размер XACT.

Конфигурационные параметры

- vacuum_freeze_min_age = 50 000 000
- vacuum_freeze_table_age = 150 000 000
- autovacuum_freeze_max_age = 200 000 000

Параметры хранения таблиц

- autovacuum_freeze_min_age
- toast.autovacuum_freeze_min_age
- autovacuum_freeze_table_age
- toast.autovacuum_freeze_table_age
- autovacuum_freeze_max_age
- toast.autovacuum_freeze_max_age

Значения по умолчанию довольно консервативны. Предел для autovacuum_freeze_max_age составляет порядка 2 млрд транзакций, а используется значение, в 10 раз меньшее. Это сделано для ограничения размера XACT (два бита на транзакцию, т. е. примерно 50 МБ при значении по умолчанию). Скорее всего, значение можно увеличивать в разы для уменьшения избыточных накладных расходов.

Обратите внимание, что изменение параметра autovacuum_freeze_max_age требует перезапуска сервера.

Параметры также можно устанавливать на уровне отдельных таблиц с помощью параметров хранения. Это имеет смысл делать только в особенных случаях, когда таблица действительно требует особого обхождения. Обратите внимание, что имена параметров на уровне таблиц немного отличаются от имен конфигурационных параметров.

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

Заметим, что заморозка «обычных» номеров транзакций 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.

«Очередь» ожидания

Блокировки строк не представлены в памяти, взамен используется блокировка номера транзакции.

Примечание: здесь и далее на рисунках сплошной линией показаны захваченные блокировки, а пунктирной — блокировки, которые еще не удалось захватить.

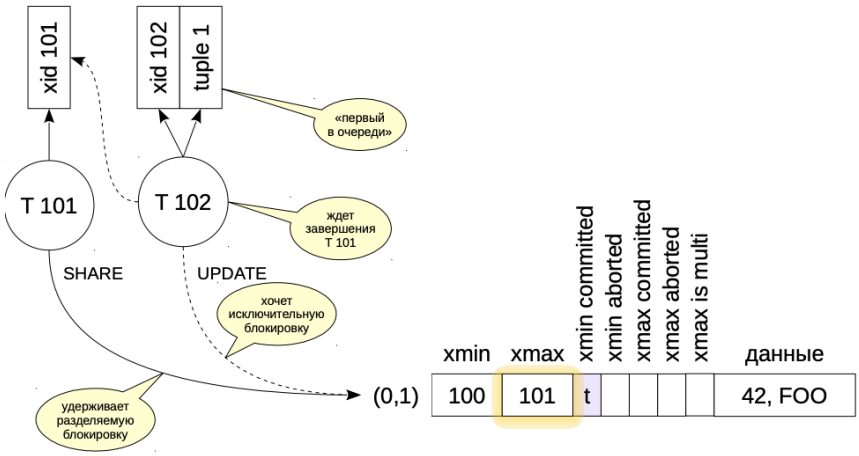

Рисунок 13 – «Очередь» ожидания

Как мы рассмотрели, транзакция блокирует строку, проставляя в поле xmax свой номер (101 в примере на рисунке 13). 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.

Поэтому используется своего рода трюк. 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 (фактически — просто исчезнет), транзакция 102 будет разбужена и сможет заблокировать строку (установить xmax = 102 в соответствующей версии строки).

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

Транзакции, конфликтующие с текущей блокировкой, встают за транзакцией, которая удерживает блокировку типа tuple

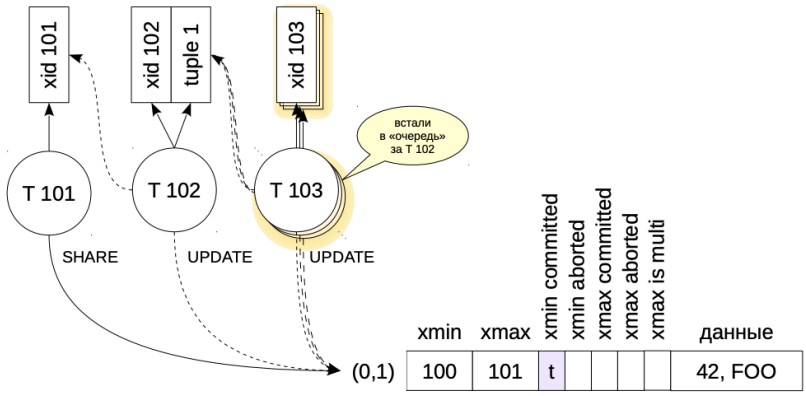

Рисунок 14 – «Очередь» ожидания

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

В нашем примере, поскольку блокировка уже удерживается транзакцией 102, другие транзакции ждут освобождения этой блокировки. Получается своеобразная «очередь», в которой есть первый и все остальные.

Когда транзакция 101 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.

Транзакции, не конфликтующие с текущей блокировкой, проходят без очереди

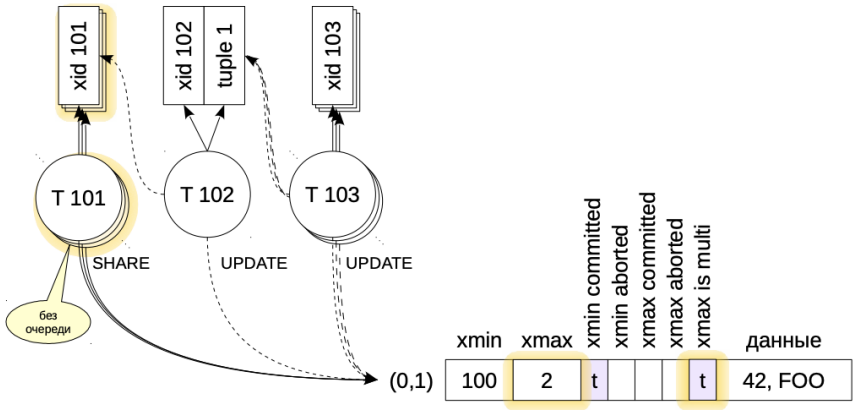

Рисунок 15 – «Очередь» ожидания

Изначальная идея такой двухуровневой схемы блокирования состояла в том, чтобы избежать ситуации вечного ожидания «невезучей» транзакции. Тем не менее, такая ситуация вполне возможна.

В нашем примере транзакция 101 заблокировала строку в разделяемом (SHARE) режиме. Пока транзакция 102 ожидает завершения транзакции 101, может появиться еще одна транзакция, желающая получить блокировку строки в разделяемом режиме, совместимом с блокировкой транзакции 101. Ничто не мешает ей работать со строкой. Поэтому она формирует мультитранзакцию, состоящую из нее самой и транзакции 101, и записывает номер мультитранзакции в xmax.

Когда транзакция 101 завершится, транзакция 102 будет разбужена, но не сможет заблокировать строку, поскольку теперь строка заблокирована другой транзакцией. 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, и все ждущие процессы выстраиваются в «честную» очередь.

Взаимоблокировки

Обнаруживаются поиском контуров в графе ожиданий

- проверка выполняется после ожидания deadlock_timeout

Одна из транзакций обрывается, остальные продолжают

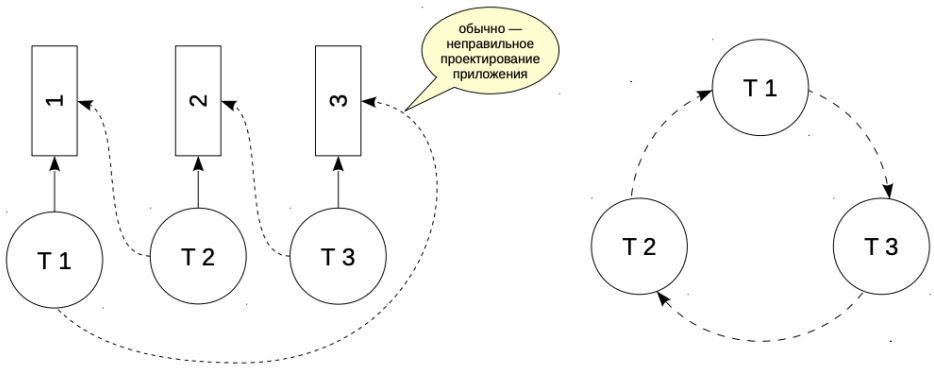

Рисунок 16 – Взаимоблокировки

Возможна ситуация взаимоблокировки, когда одна транзакция пытается захватить ресурс, уже захваченные 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.

Частое применение явных блокировок может увеличить вероятность *взаимоблокировок*, то есть ситуаций, когда две (или более) транзакций держат блокировки так, что взаимно блокируют друг друга. Например, если транзакция 1 получает исключительную блокировку таблицы A, а затем пытается получить исключительную блокировку таблицы B, которую до этого получила транзакция 2, в данный момент требующая исключительную блокировку таблицы A, ни одна из транзакций не сможет продолжить работу. PostgreSQL автоматически выявляет такие ситуации и разрешает их, прерывая одну из сцепившихся транзакций и тем самым позволяя другой (другим) продолжить работу. (Какая именно транзакция будет прервана, обычно сложно предсказать, так что рассчитывать на определённое поведение не следует.)

Заметьте, что взаимоблокировки могут вызываться и блокировками на уровне строк (таким образом, они возможны, даже если не применяются явные блокировки). Рассмотрим случай, когда две параллельных транзакции изменяют таблицу. Первая транзакция выполняет:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum =
11111;
```

При этом она получает блокировку строки с указанным номером счёта. Затем вторая транзакция выполняет:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum =
22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum =
11111;
```

Первый оператор UPDATE успешно получает блокировку указанной строки и изменяет данные в ней. Однако второй оператор UPDATE обнаруживает, что строка, которую он пытается изменить,

уже заблокирована, так что он ждёт завершения транзакции, получившей блокировку. Таким образом, вторая транзакция сможет продолжиться только после завершения первой. Теперь первая транзакция выполняет:

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

Первая транзакция пытается получить блокировку заданной строки, но ей это не удаётся: эта блокировка уже принадлежит второй транзакции. Поэтому первой транзакции остаётся только ждать завершения второй. В результате первая транзакция блокируется второй, а вторая — первой: происходит взаимоблокировка. PostgreSQL выявляет эту ситуацию и прерывает одну из транзакций.

Обычно лучший способ предотвращения взаимоблокировок — добиться, чтобы все приложения, обращающиеся к базе данных, запрашивали блокировки нескольких объектов единообразно. В данном примере, если бы обе транзакции изменяли строки в одном порядке, взаимоблокировка бы не произошла. Блокировки в транзакции следует упорядочивать так, чтобы первой для какого-либо объекта запрашивалась наиболее ограничивающая из тех, которые для него потребуются. Если заранее обеспечить такой порядок нельзя, взаимоблокировки можно обработать по факту, повторяя прерванные транзакции.

Если ситуация взаимоблокировки не будет выявлена, транзакция, ожидающая блокировки на уровне таблицы или строки, будет ждать её освобождения неограниченное время. Это означает, что приложения не должны оставлять транзакции открытыми долгое время (например, ожидая ввода пользователя).

Практика 1

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
"postgres".
=> 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
commited,
       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
"postgres".
=> 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  | commited | aborted | lock_only | is_multi |
keys_upd
-------+-------+----------+---------+-----------+----------+----
------
 (0,1) | 26993 |          |         |           |          |
 (0,2) | 26993 |          |         |           |          | 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
"postgres".
=> 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  | commited | aborted | lock_only | is_multi |
keys_upd
-------+-------+----------+---------+-----------+----------+----
------
 (0,1) |     1 |          |         |           | t        |
 (0,2) | 26993 |          |         |           |          | t
 (0,3) | 26994 |          |         | 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     | 1
multi      | t
xids       | {26993,26994}
modes      | {"No Key Update","Key Share"}
pids       | {11973,12082}
-[ RECORD 2 ]--------------------------
locked_row | (0,2)
locker     | 26993
multi      | f
xids       | {26993}
modes      | {Update}
pids       | {11973}
-[ RECORD 3 ]--------------------------
locked_row | (0,3)
locker     | 26994
multi      | f
xids       | {26994}
modes      | {"For Share"}
pids       | {12082}
```

Расширение дает информацию о режимах всех блокировок.

```
=> ROLLBACK;
ROLLBACK

=> ROLLBACK;
ROLLBACK
```

"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||':'||tuple::text
       END AS lockid,
       mode,
       granted
FROM pg_locks;
CREATE VIEW
```

Начнем с двух транзакций.

Пусть одна транзакция заблокирует строку в разделяемом режиме...

```
=> BEGIN;
BEGIN
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
        26997 |          11882
(1 row)

=> SELECT * FROM accounts WHERE acc_no = 1 FOR SHARE;
 acc_no | amount
--------+---------
      1 | 1000.00
(1 row)
```

...а вторая попробует выполнить обновление:

```
=> BEGIN;
BEGIN
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
        26998 |          11973
(1 row)
```

```
=> 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):

```
=> SELECT * FROM locks WHERE pid = 11973;
  pid  |   locktype    |    lockid     |       mode       |
granted
-------+---------------+---------------+------------------+-----
----
 11973 | relation      | accounts_pkey | RowExclusiveLock | t
 11973 | relation      | accounts      | RowExclusiveLock | t
 11973 | virtualxid    | 3/26          | ExclusiveLock    | t
 11973 | transactionid | 26997         | ShareLock        | f
 11973 | transactionid | 26998         | ExclusiveLock    | t
 11973 | tuple         | accounts:1    | ExclusiveLock    | t
(6 rows)
```

При этом она удерживает блокировку версии строки (locktype = tuple).

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(11973);
 pg_blocking_pids
------------------
 {11882}
(1 row)
```

Если теперь появляется третья транзакция, желающая заблокировать ту же строку в разделяемом режиме, она проходит без очереди.

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

```
 txid_current | pg_backend_pid
--------------+----------------
        26999 |          12082
(1 row)

=> SELECT * FROM accounts WHERE acc_no = 1 FOR SHARE;
 acc_no | amount
--------+---------
      1 | 1000.00
(1 row)

=> SELECT * FROM pgrowlocks('accounts') \gx
-[ RECORD 1 ]-------------
locked_row | (0,1)
locker     | 2
multi      | t
xids       | {26997,26999}
modes      | {Share,Share}
pids       | {11882,12082}
```

После того, как первая транзакция завершится, вторая не сможет обновить строку потому, что остается третья транзакция.

```
=> COMMIT;
COMMIT
```

Теперь вторая транзакция ожидает завершения третьей:

```
=> SELECT * FROM locks WHERE pid = 11973;
  pid  |   locktype    |    lockid     |       mode       |
granted
-------+---------------+---------------+------------------+-----
----
 11973 | relation      | accounts_pkey | RowExclusiveLock | t
 11973 | relation      | accounts      | RowExclusiveLock | t
 11973 | virtualxid    | 3/26          | ExclusiveLock    | t
 11973 | transactionid | 26998         | ExclusiveLock    | t
 11973 | transactionid | 26999         | ShareLock        | f
```

```
 11973 | tuple          | accounts:1    | ExclusiveLock    | t
(6 rows)

=> COMMIT;
COMMIT

UPDATE 1
=> ROLLBACK;
ROLLBACK
```

Как избежать ожидания

Обычно команды SQL ожидают освобождения блокировки. Но иногда хочется отказаться от выполнения, если блокировку не удалось получить сразу же. Для этого есть два варианта команды SELECT FOR.

Первый вариант - использовать фразу NOWAIT.

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

=> SELECT * FROM accounts FOR UPDATE NOWAIT;
ERROR:  could not obtain lock on row in relation "accounts"
Команда немедленно завершается с ошибкой.
```

Второй вариант - использовать фразу SKIP LOCKED. При этом команда будет пропускать заблокированные строки, но обрабатывать оставшиеся.

```
=> BEGIN;
BEGIN
=> DECLARE c CURSOR FOR
SELECT * FROM accounts ORDER BY acc_no FOR UPDATE SKIP LOCKED;
DECLARE CURSOR
=> FETCH c;
 acc_no | amount
--------+---------
```

```
      2 | 2000.00
(1 row)
```

Практическая применимость такого подхода состоит в многопоточной обработке очередей.

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

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 11973 waits for ShareLock on transaction 27003;
blocked by process 12082.
Process 12082 waits for ShareLock on transaction 27002; blocked
by process 11973.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,2) in relation "accounts"
=> COMMIT;
ROLLBACK

UPDATE 1
=> 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

Задание на практическую работу:

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.