ChatGPT

Блокировки в Postgres Pro 16: виды и особенности

Postgres Pro 16 (основанный на PostgreSQL 16) применяет механизм **MVCC** (Multiversion Concurrency Control, многоверсионное управление параллелизмом) в сочетании с различными типами блокировок для обеспечения целостности данных при одновременном доступе. Главное преимущество модели MVCC – читатели не блокируют писателей, а писатели не блокируют читателей [1]. Благодаря этому обычные SELECT-запросы не мешают параллельным изменениям данных, и наоборот, что повышает производительность в многопользовательской среде. В то же время при выполнении операций записи и некоторых видов чтения СУБД задействует различные **блокировки**, чтобы синхронизировать транзакции. Ниже подробно рассмотрены **все типы блокировок в Postgres Pro 16**, ситуации их использования, влияние на параллелизм и производительность, а также устройство механизма блокировок и примеры типичных взаимоблокировок.

Блокировки на уровне строк (row-level locks)

Блокировки строк защищают отдельные строки таблицы от одновременных конфликтующих операций. Когда транзакция изменяет или явно блокирует строку, Postgres помечает соответствующую версию строки специальными флагами – по сути, фиксирует в ее заголовке идентификатор текущей транзакции (в поле *xmax*), иногда с дополнительными битами [2] [3]. Такая пометка выполняет роль блокировки: пока транзакция не завершится (COMMIT или ROLLBACK), другие транзакции видят, что эта версия строки «занята». **Важно:** информация о блокировке строки хранится *только* в самой строке на диске, а не в общей таблице блокировок в памяти [2]. Это означает, что для каждой заблокированной строки не расходуются ресурсы shared memory – можно заблокировать сколь угодно много строк без роста накладных расходов [2].

Однако и **очереди ожидания** для таких блокировок напрямую не существует, ведь другие процессы не могут «увидеть» отметку о блокировке в памяти [4]. Решение PostgreSQL – использовать обычные блокировки транзакций для организации ожидания [5]. Каждая активная транзакция удерживает **исключительную блокировку своего идентификатора** (виртуального и при присвоении – постоянного) на время выполнения; если другой процесс хочет изменить уже занятую строку, он будет ожидать завершения блокирующей транзакции, запросив *разделяемую блокировку* на идентификатор этой транзакции [5] [6]. Таким образом, при конкуренции за строку фактически одна транзакция **ждет разблокировки другой**, а число задействованных в памяти блокировок пропорционально числу активных транзакций, а не количеству заблокированных строк [5]. В системном представлении `pg_locks` такие ожидания обычно проявляются как блокировки типа `transactionid` с режимом `ShareLock` на XID блокирующей транзакции [6].

Режимы блокировки строк. PostgreSQL предоставляет **четыре режима блокировки отдельных строк** [7]. Среди них два – эксклюзивные, которые допускают владение строкой

только одной транзакцией, и два – разделяемые (шаринг), позволяющие совместный доступ для некоторых операций только чтения:

- **FOR UPDATE:** эксклюзивный режим, означающий, что транзакция намерена полностью изменить или удалить данную строку. Блокирует эту строку от любых изменений или чтения с блокировкой другими транзакциями [8] .
- **FOR NO KEY UPDATE:** также эксклюзивный режим, но «мягче» – транзакция изменяет строку без затрагивания ключевых столбцов, участвующих в уникальных индексах (т.е. без изменения значений, на которые могут ссылаться внешние ключи) [9] . Такой режим используется по умолчанию обычной командой UPDATE для большинства изменений [10] . Он блокирует конкурирующие изменения и запросы `SELECT FOR UPDATE/NO KEY UPDATE` на эту же строку, но допускает более слабые блокировки чтения (см. ниже).
- **FOR SHARE:** разделяемый (неэксклюзивный) режим блокировки строки. Указывает, что транзакция **читает** строку с намерением использовать данные (например, для проверки существования строки). Несколько транзакций могут одновременно удерживать `FOR SHARE` на разных или даже тех же строках. Однако эта блокировка предотвращает эксклюзивные изменения той же строки другими транзакциями до окончания текущей (блокирует операции UPDATE/DELETE на эту строку в других транзакциях) [11] .
- **FOR KEY SHARE:** еще более «слабый» разделяемый режим. Транзакция читает строку без намерения менять *какие-либо* данные, в том числе не планирует менять внешние ключи, ссылающиеся на эту строку. Блокировка `FOR KEY SHARE` позволяет даже другим транзакциям накладывать `FOR NO KEY UPDATE` на эту же строку (т.е. изменять неключевые поля) [12] . Обычно такой режим применяется автоматически в подчиненных таблицах при проверке внешних ключей – чтобы гарантировать наличие связанной строки в основной таблице, но при этом не мешать обновлениям неключевых полей в ней.

Таким образом, **эксклюзивные блокировки строк** (`FOR UPDATE` / `FOR NO KEY UPDATE`) предотвращают любый параллельные изменения *и даже чтение с блокировкой* данной строки другими, а **разделяемые блокировки строк** (`FOR SHARE` / `FOR KEY SHARE`) позволяют совместное чтение и некоторую степень совместимости с обновлениями. Все четыре режима блокировок строк взаимосвязаны по иерархии конфликтов: например, `FOR UPDATE` конфликтует со всеми остальными режимами на ту же строку, `FOR NO KEY UPDATE` конфликтует со `FOR UPDATE` и `FOR SHARE`, но совместим с `FOR KEY SHARE` [13] , и т.д. (подробнее см. матрицу конфликтов режимов строк [14]).

Неявные и явные блокировки строк. В большинстве случаев разработчику не нужно вручную ставить блокировку на строку – PostgreSQL делает это сам при выполнении операций: - Любой `UPDATE` или `DELETE` автоматически блокирует затронутые строки в режиме `FOR UPDATE` или `FOR NO KEY UPDATE` (выбирается минимально достаточный режим) [15] . В частности, если обновляются столбцы, от которых зависят внешние ключи, будет применён более строгий `FOR UPDATE` [16] . - `SELECT ... FOR UPDATE/SHARE` – явный запрос блокировки на найденные строки. Он используется, когда нужно **прочитать и заблокировать** данные, чтобы затем безопасно их изменить или убедиться, что они не изменятся до конца текущей транзакции. - Внутренние механизмы также используют блокировки строк. Например, при вставке новой строки в таблицу с уникальным индексом, PostgreSQL может временно заблокировать конфликтующую старую версию строки (если происходит **спекулятивная вставка** для реализации upsert-логики).

Влияние на параллелизм и взаимоблокировки. Блокировки на уровне строк обладают *высокой гранулярностью*, что обычно способствует параллелизму: транзакции, обновляющие

разные строки одной таблицы, работают независимо без конфликтов. Благодаря MVCC, немодифицирующие SELECT-запросы вообще не требуют блокировать строки – они читают снимок данных, поэтому «чтение никогда не мешает записи, а запись – чтению» [1]. Тем не менее, при конкурентном изменении *одних и тех же строк* может возникать ожидание и даже **deadlock (взаимоблокировка)**. Классический пример: две транзакции пытаются обновить две строки в разном порядке. Транзакция A сначала блокирует строку X, затем пытается блокировать Y; транзакция B в обратном порядке — сначала Y, потом X. Каждая получит эксклюзивный доступ к первой строке и будет ждать другую при попытке доступа ко второй – возникает циклическое ожидание. PostgreSQL через `deadlock_timeout` (по умолчанию 1 с) обнаружит такую ситуацию и принудительно прервет одну из транзакций [17] [18] (см. раздел о взаимоблокировках далее). Во избежание подобных проблем рекомендуется всегда брать блокировки (например, выполнять обновления или SELECT FOR UPDATE) в согласованном порядке – допустим, всегда по возрастанию первичного ключа, если логика приложения это позволяет [19].

В целом, **строчные блокировки** – наименее «тяжелые» для системы: они не хранятся в памяти и блокируют минимальный возможный объект. Это обеспечивает максимальный параллелизм при правильном использовании, хотя и усложняет выявление проблем (такие блокировки не видны напрямую через `pg_locks`, пока не заставят ждать транзакцию [20]). Разработчикам важно понимать, что длительные транзакции, изменяющие много строк, будут удерживать блокировки этих строк до завершения – тем самым потенциально долго блокируя чужие изменения тех же записей. В таких случаях стоит либо сокращать время транзакции, либо разделять её логику, либо использовать более крупные блокировки (но краткосрочные), если приемлемо.

Блокировки на уровне таблиц (table-level locks)

Табличные блокировки распространяются на всю таблицу (или другое отношение в целом) и контролируются через **менеджер блокировок** в общей памяти. PostgreSQL поддерживает *восемь* режимов блокировок таблицы [21] [22] – от наиболее «мягкого» `ACCESS SHARE` до самого строгого `ACCESS EXCLUSIVE`. Эти режимы определяют, какие действия с таблицей может параллельно выполнять другая транзакция: у каждого режима есть набор *конфликтующих* режимов, с которыми он несовместим [23]. Если две транзакции запросили несовместимые блокировки одной таблицы, вторая будет ждать освобождения ресурса первой. Ниже перечислены режимы табличных блокировок и их использование (конфликты см. в документации, табл. 13.2 [24] [25]):

- **ACCESS SHARE** (`AccessShareLock`): самый либеральный режим. **Не конфликтует** ни с каким другим, кроме `ACCESS EXCLUSIVE` [26]. Любой обычный SELECT (чтение без блокаировки строк) получает на все задействованные таблицы блокировку `ACCESS SHARE` автоматически [27]. Таким образом, множество транзакций могут одновременно читать одну таблицу, пока никто не выполняет с ней операцию, требующую эксклюзивной блокировки.
- **ROW SHARE** (`RowShareLock`): конфликтует с `EXCLUSIVE` и `ACCESS EXCLUSIVE` [28]. Этот режим используют SELECT-запросы с `FOR UPDATE/SHARE` для базовых таблиц – фактически он сигнализирует, что транзакция удерживает строчные блокировки в данной таблице и, следовательно, намерена потенциально её изменять или закрепить данные [29]. `ROW SHARE` не мешает другим транзакциям читать или даже брать `ROW SHARE` на ту же таблицу, но блокирует более строгие блокировки (например, ни одна транзакция не сможет эксклюзивно захватить таблицу, пока другая держит на ней хотя бы `ROW SHARE`).
- **ROW EXCLUSIVE** (`RowExclusiveLock`): конфликтует с режимами `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` и `ACCESS EXCLUSIVE` [30]. Этот режим автоматически берётся

при любом изменении данных – команды `INSERT`, `UPDATE`, `DELETE`, а также `MERGE` накладывают `ROW EXCLUSIVE` на изменяемую таблицу [31]. Блокировка означает, что таблица сейчас изменяется, и предотвращает параллельные операции, несовместимые с изменением (например, одновременный `CREATE INDEX` без CONCURRENTLY или другие эксклюзивные действия). Однако несколько транзакций *могут* одновременно держать `ROW EXCLUSIVE` на одной таблице – то есть параллельные изменения данных возможны, если они не мешают друг другу на уровне строк.

- **SHARE UPDATE EXCLUSIVE** (`ShareUpdateExclusiveLock`): конфликтует с режимами `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`, `ACCESS EXCLUSIVE` [32]. Эта блокировка защищает таблицу от изменений её схемы и от запуска **VACUUM**. Она ставится автоматически командами, которые читают или слегка модифицируют таблицу, требуя стабильной структуры: например, `VACUUM` (без FULL), `ANALYZE`, создание индекса **CONCURRENTLY**, сбор статистики (`CREATE STATISTICS`), некоторые DDL-операции вроде `ALTER TABLE ... VALIDATE CONSTRAINT` и т.п. [33]. `SHARE UPDATE EXCLUSIVE` позволяет параллельно изменять данные (не конфликтует с `ROW EXCLUSIVE`), но не даёт выполнять другие DDL, создающие аналогичную или большую блокировку.
- **SHARE** (`ShareLock`): конфликтует с `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`, `ACCESS EXCLUSIVE` [34]. Блокирует параллельные изменения данных в таблице. Используется в основном для операций, которым нужно «заморозить» таблицу на время чтения данных. Например, `CREATE INDEX` **без CONCURRENTLY** накладывает `SHARE` на индексируемую таблицу [35], чтобы предотвратить модификации данных во время построения индекса. В отличие от `ACCESS SHARE`, режим `SHARE` не совместим с `ROW EXCLUSIVE` – т.е. параллельные INSERT/UPDATE/DELETE будут ждать.
- **SHARE ROW EXCLUSIVE** (`ShareRowExclusiveLock`): конфликтует с `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`, `ACCESS EXCLUSIVE` [36]. По сути, это также блокировка против параллельных изменений данных, но к тому же *самоисключающая* – только одна транзакция может удерживать `SHARE ROW EXCLUSIVE` на данной таблице [37]. Применяется в более узких случаях, например при создании **триггера** (`CREATE TRIGGER`) или некоторых вариантах `ALTER TABLE` (например, изменение типа колонки с переписью таблицы) [38]. То есть операции, которым нежелательно даже параллельное чтение аналогичных данных другой такой же операцией.
- **EXCLUSIVE** (`ExclusiveLock`): конфликтует со всеми режимами, *кроме* `ACCESS SHARE` [39]. Это означает, что пока транзакция держит `EXCLUSIVE`, другим разрешается только обычное чтение таблицы (без блокирующего чтения и без изменений) [40]. Пример использования – `REFRESH MATERIALIZED VIEW CONCURRENTLY`: при обновлении материализованного представления в конкурентном режиме исходная таблица данных берётся в `EXCLUSIVE` [39], чтобы никто не менял ее, пока идёт вычисление нового содержимого (но простые SELECT всё ещё возможны параллельно).
- **ACCESS EXCLUSIVE** (`AccessExclusiveLock`): самый строгий режим, **конфликтует со всеми без исключения** [41]. Гарантирует, что транзакция – единственная, кто может обращаться к таблице: никакого параллельного чтения или записи не допускается [42]. Этот режим ставится при **DDL-операциях**, которые изменяют или могут пересоздавать всю таблицу: `DROP TABLE`, `TRUNCATE`, `VACUUM FULL`, `CLUSTER`, неконкурентный `REINDEX`, `REFRESH MATERIALIZED VIEW` (без CONCURRENTLY) и при большинстве форм `ALTER TABLE` / `ALTER INDEX` [43]. Фактически, `ACCESS EXCLUSIVE` на время полностью «закрывает» таблицу от других транзакций.

Как видно, **табличные блокировки** покрывают широкий спектр ситуаций. PostgreSQL старается автоматически брать *минимально необходимый* режим блокировки при выполнении команды [22]. Например, простой `SELECT` использует лишь `ACCESS SHARE` (не мешает почти ничему), `UPDATE` – `ROW EXCLUSIVE`, `VACUUM` – `SHARE UPDATE EXCLUSIVE` и т.д. Разработчику же явная команда `LOCK TABLE ... IN <mode>` позволяет вручную взять нужную блокировку, если логика приложения требует на время транзакции гарантировать отсутствие определённых конкурирующих действий [22]. Явный `LOCK TABLE` обычно применяется редко – например, чтобы зафиксировать моментальный снимок данных в режиме Read Committed или предотвратить потенциальный дедлок при сложных обновлениях, запросив сразу более строгую блокировку [44] [45].

Влияние на параллелизм. Табличные блокировки более *грубозернистые*, чем строчные, поэтому сильнее ограничивают параллельную работу. Пока транзакция удерживает, скажем, `ACCESS EXCLUSIVE` на таблице, все другие операции с этой таблицей вынужденно ждут – это негативно сказывается на производительности, особенно если такая транзакция длительная. Даже менее строгие режимы, как `SHARE` или `EXCLUSIVE`, могут существенно тормозить работу, если часто применяются, так как блокируют целые категории операций. Поэтому важно минимизировать время удержания крупных блокировок: например, DDL-операции (INDEX, ALTER) делать в наименее загруженное время или с опцией CONCURRENTLY, если доступно.

Взаимоблокировки. Deadlock с участием табличных блокировок возможен, если несколько транзакций запрашивают блокировки на *разные таблицы в разном порядке*. Простой сценарий: транзакция 1 берет `ACCESS SHARE` на таблицу A (читает её), затем хочет `ACCESS SHARE` на таблицу B; параллельно транзакция 2 уже держит `ACCESS SHARE` на B и запрашивает на A. Казалось бы, чтение не мешает чтению – но если обе транзакции выполняют SELECT FOR UPDATE (т.е. реально режим `ROW SHARE` или строчные блокировки внутри), либо пытаются повысить режим (например, затем выполнить `UPDATE`), может возникнуть конфликт и циклическое ожидание. Однако на практике такие ситуации редки, так как чистые чтения совместимы, а при смешанных операциях обычно задействуются еще и строчные блокировки. PostgreSQL, в свою очередь, никогда не сталкивается с дедлоком сам с собой – транзакция может повысить свою блокировку (например, с `ACCESS SHARE` до `ROW EXCLUSIVE`) последовательно, не конфликтуя сама с собой [46]. Тем не менее, общий совет: если ваше приложение в одной транзакции работает с несколькими таблицами, всегда берите блокировки/обновляйте таблицы в фиксированном порядке, чтобы разные транзакции не «пересекались» в разных последовательностях.

Ресурсы и мониторинг. В отличие от строчных, блокировки на уровне таблиц (и другие «тяжелые» блокировки) хранятся в общей памяти и их число ограничено настройками. Параметр `max_locks_per_transaction` определяет размер таблицы блокировок: примерно столько объектов (таблиц) может удерживать *каждый* процесс или подготовленная транзакция, а общее число различных заблокированных объектов = `max_locks_per_transaction * max_connections` (по умолчанию 64 * 100 = 6400) [47] [48]. Эти блокировки видны в системном представлении `pg_locks`: там они имеют тип `relation` (отношение) в столбце `locktype` [49] [50]. Также `pg_locks` показывает режим (`mode`) и флаг `granted` (получена или ожидается) [51]. Если запрос ожидает таблицу, можно увидеть запись с `granted = false` и нужным режимом, а также увидеть, какая другая транзакция удерживает конфликтующую блокировку (у нее будет `granted = true` на тот же объект). Для отладки взаимоблокировок полезно строить *граф ожидания* или использовать функции типа `pg_blocking_pids()`, но часто достаточно сообщений в логе (Postgres пишет в журнал подробности обнаруженного deadlock'a, включая участвующие транзакции и блокируемые ресурсы) [52].

Блокировки объектов (object locks) и другие специальные блокировки

Под **блокировками объектов** обычно понимаются блокировки *непосредственно не относящиеся к конкретной таблице или строке*. В PostgreSQL для этого используется тип блокировки с именем `object` [53]. Такой объект – это практически *любая запись системного каталога*, не являющаяся обычным отношением. Примеры: *табличные пространства, базы данных, схемы, роли, типы данных, индексы, ограничения* и пр. – то есть все, что имеет OID в системе и может требовать синхронизации при изменениях [54].

Когда используются блокировки типа object? Как правило, при выполнении операций, создающих, изменяющих или удаляющих различные объекты базы данных, Postgres автоматически блокирует связанные с ними записи каталогов, чтобы предотвратить конфликтующие изменения. Например, при создании новой таблицы внутри транзакции СУБД наложит `AccessShareLock` на объект роли-владельца и на объект схемы, в которой создается таблица [55]. Благодаря этому никто не сможет одновременно удалить эту роль или схему, пока транзакция не завершена, что сохраняет целостность каталога [55]. В `pg_locks` такие блокировки будут видны с `locktype = 'object'`, а идентифицируются они тройкой полей: `classid` (OID системного каталога, например `pg_authid` для ролей или `pg_namespace` для схем), `objid` (OID конкретного объекта в этом каталоге) и `database` (OID базы, либо 0 для глобальных объектов) [56] [57]. В приведенном Е.Роговым примере при создании таблицы блокируется роль *student* (глобальный объект, `database=0`, `classid=pg_authid`, `objid=16384`) и схема *public* текущей БД (`classid=pg_namespace`, `objid=2200`) [58] [57].

Кроме схем и ролей, блокировки объектов ставятся на множество других сущностей: **базу данных** (например, на время `DROP DATABASE` или при эксклюзивных операций, затрагивающих всю базу), **табличное пространство** (на время его удаления, перемещения объектов), **конкретные функции, виды, типы** – при изменении их определения, **индексы** – при изменении индекса, и т.д. Многие DDL-команды используют как блокировку уровня отношения (если оперируют таблицей), так и дополнительные блокировки объектов. Например, `ALTER TABLE` при переименовании колонки возьмет блокировку `AccessExclusive` на таблицу *и* `AccessShareLock` на сам каталог `pg_attribute` или другие связанные объекты, чтобы никто не менял их параллельно. Эти детали чаще внутренние, но упоминание об *object lock* иногда всплывает при анализе `pg_locks`.

Отдельно к «объектным» можно отнести два особых типа блокировок, которые в выводе `pg_locks` выделены в собственные категории: **блокировка расширения отношения** (`extend`) и блокировка замороженного XID (`frozenid`).

- **Блокировка расширения отношения** – это внутренняя блокировка, предохраняющая от одновременного расширения (увеличения файла) таблицы или индекса разными процессами [59]. Когда одной транзакции нужно добавить новую страницу к концу отношения (например, при вставке данных, если нет свободного места), она берет на долю секунды эксклюзивную блокировку типа `extend`. Этот же механизм используется при чистке GIN-индексов, чтобы два процесса не добавляли страницы одновременно во время реорганизации индекса [60]. Такая блокировка удерживается *минимальное время* и сразу снимается – она не дожидается конца транзакции [60]. Поэтому в нормальной работе разработчик почти не видит влияния `extend`, разве что при очень высоконагрузочных конкурентных вставках он может заметить паузы на расширении. С 9.6 PostgreSQL

оптимизировал расширение: если много процессов ожидают `extend`, система добавит сразу несколько страниц за раз (до 512), чтобы раздать их всем [61].

- **«Замораживание XID» (`frozenid`)** – спецблокировка, связанная с поддержанием `datfrozenxid` у базы данных. Когда выполняется продвинутое замораживание транзакций (автовакуум обновляет глобальный минимальный XID), может браться блокировка `frozenid` на базу данных, не позволяющая параллельно выполнять какие-то конфликтующие действия. Этот тип упоминается в документации [49], но в повседневной работе он редко проявляется явно для пользователей.

Блокировка страниц. PostgreSQL в целом *не применяет явных блокировок на уровне страницы* (то есть отдельного блока данных) при обычной работе транзакций – в отличие от некоторых СУБД, тут нет автоматического повышения блокировки со строки до страницы. **Исключение** – случаи, связанные с индексами и сериализацией. Один из примеров – GIN-индексы. Чтобы ускорить вставку в GIN, используется буферизация изменений (список pending list). Когда накопившиеся ключи перемещаются из этого списка в основной индекс, **метастраница индекса** блокируется в режиме `Exclusive` на время слияния [62]. Это блокировка типа `page` (страница) – единственный штатный случай применения page-level lock, не считая предикатных (о них ниже) [63]. Эксклюзивная блокировка метастраницы не мешает остальным операциям поиска/вставки в индекс, потому что они работают с другими страницами; то есть для пользователей это прозрачно [62]. Другие методы индексов могут также использовать кратковременные блокировки страниц *внутри себя* (например, B-деревья применяют свои механизмы latching для страниц, но это уже не отражается в `pg_locks` как «page»).

Взаимосвязь с другими блокировками. Объектные блокировки зачастую идут *вместе* с блокировками таблиц. Например, создание таблицы: помимо описанных блокировок схемы и роли (object-locks), сама новая таблица тоже блокируется на уровне отношения (обычно `ACCESS EXCLUSIVE`, так как она только что создана и до конца транзакции недоступна другим). При создании индекса мы увидим `ShareLock` на таблицу и `AccessShareLock` на индекс как на объект. Такая взаимосвязь гарантирует целостность: пока держится блокировка таблицы, никто ее не удалит; пока держится блокировка связанного объекта, он тоже не изменится или не исчезнет из-под ног операции.

Блокировки транзакций (transaction locks)

При конкуренции транзакций за данные PostgreSQL часто использует блокировки не только *данных*, но и *идентификаторов транзакций*. Как упоминалось, каждая активная транзакция удерживает **исключительную блокировку своего виртуального XID** на время работы, а при получении реального идентификатора (XID) – блокирует и его до завершения [6]. Эти блокировки служат, по сути, для организации очереди ожидания между транзакциями. Когда одна транзакция должна дождаться завершения другой, она запрашивает **разделяемую блокировку** на XID той транзакции [6]. Например, ситуация с внешними ключами: транзакция A вставила строку в таблицу «родителя», транзакция B пытается вставить строку в дочернюю таблицу, ссылающуюся на родительскую. Если транзакция A еще не зафиксирована, B должна убедиться, что ссылка валидна – для этого B будет ждать окончания A. Этот механизм реализован через блокировку идентификатора транзакции A: процесс B ставит `ShareLock` на XID транзакции A и засыпает до ее завершения [6]. После коммита A блокировка снимается, и B продолжает работу.

Блокировки транзакций – это внутренний механизм синхронизации, прозрачный для пользователя. Обычно они короткоживущие и существуют только пока одна транзакция ждёт другую. Видны они в `pg_locks` как `locktype = 'transactionid'` (для постоянных XID) или `virtualxid` (для виртуальных, реже нужно) [64] [65]. Режим обычно `ShareLock` (ждущая транзакция) или `ExclusiveLock` (сама транзакция на свой XID). Поскольку **собственная транзакция никогда не конфликтует сама с собой** [46], несколько разных блокировок на один XID не накладываются – в этом нет смысла.

В редких случаях можно столкнуться с взаимоблокировкой и на уровне XID. Например, две транзакции с взаимоотношениями через внешние ключи: транзакция 1 вставила строку в таблицу A и ждет (через AFTER trigger) коммита вставки в таблицу B из транзакции 2, а транзакция 2, в свою очередь, зависит от коммита транзакции 1. Такой цикл обычно невозможен, если внешние ключи не обозначены как deferrable, но гипотетически может возникнуть, если приложение вручную организует подобные ожидания. PostgreSQL обнаружит и разорвет такой deadlock на уровне XID так же, как любой другой (через `deadlock_timeout`).

В целом, **блокировки транзакций** – это вспомогательный тип, который позволяет реализовать принцип: «кто-то держит строку – жди завершения его транзакции». Они влияют на параллелизм только в конфликтных ситуациях (например, при конкурирующих апдейтах или пересекающихся ограничениях), введя упорядоченное ожидание вместо хаотичного конфликта. Накладные расходы от таких блокировок минимальны (каждая транзакция всегда держит *одну* эксклюзивную блокировку на себя, плюс разделяемые – только если ждет других).

Мониторинг: Долгое ожидание на блокировке XID может свидетельствовать о проблеме (например, забытый COMMIT в другой сессии). В `pg_stat_activity` такая сессия появится как «waiting», а в `pg_locks` – запись с `transactionid` (granted=false). Поле `pg_locks.waitstart` покажет, когда ожидание началось, что помогает оценить длительность блокировки [66]. Параметр `lock_timeout` может прерывать слишком долгие ожидания блокировок (любых, в том числе XID) [67].

Advisory Locks (рекомендательные/советующие блокировки)

Рекомендательные блокировки – это отдельный механизм, позволяющий приложениям самостоятельно устанавливать логические блокировки, *не привязанные прямо к объектам базы данных*. PostgreSQL называет их *advisory locks* (в русской документации также «рекомендательные»), потому что система сама **никогда не берёт их автоматически** [68] – ими управляет только прикладной код. Они полезны, когда нужна синхронизация на уровне приложения: например, блокировка какого-то ресурса или раздела данных, который не соответствует одной конкретной таблице или строке.

Виды advisory-блокировок: эти блокировки бывают *двух уровней* – сесссионные и транзакционные – и в *двух режимах* (разделяемые и эксклюзивные). - Сессионные (session-level) держатся до тех пор, пока соединение не разорвётся (или пока явно не отпущены). Они игнорируют границы транзакций. - Транзакционные (transaction-level) привязаны к текущей транзакции и автоматически освобождаются при её завершении (COMMIT/ROLLBACK).

Разработчик выбирает нужный вариант вызовом соответствующей функции SQL: для установки блокировки – `pg_advisory_lock(...)` (эксклюзивная, транзакционная), `pg_advisory_xact_lock(...)` (эксклюзивная, но ограниченная транзакцией), или их

разделяемые версии `pg_advisory_share_lock(...)` и т.д. Для сессионных блокировок используются варианты без «xact» в названии.

Каждая advisory-блокировка идентифицируется либо *двумя 32-бит числами*, либо одним 64-битным (Postgres предоставляет два интерфейса). Чаще всего программисты придумывают некий **числовой идентификатор ресурса**. Например, можно взять хэш от имени логического ресурса: `SELECT pg_advisory_lock(hashtext('ресурс1'));` – захватит эксклюзивную блокировку с определенным числом [69] [70]. Если другой процесс попытается взять такую же блокировку (например, по тому же идентификатору), он будет ждать освобождения.

Advisory-locks могут быть **разделяемыми** (shared) – тогда несколько сессий могут совместно «держать» один ресурс, пока ни одна не запросит эксклюзивную блокировку. Это удобно для случаев «многим читать – одному писать», по аналогии с read/write-lock.

Использование и влияние. Поскольку эти блокировки не связаны напрямую с состоянием БД, их применение полностью зависит от правильности логики приложения. Они могут охватывать произвольные критические секции кода. Например, можно таким образом **гарантировать уникальность действия**, распределяя работу между узлами кластера (кто захватил advisory-lock – тот выполняет задачу, остальные ждут). При разумном использовании advisory-блокировки облегчают координацию без дополнительной таблицы-флагов.

Важно понимать, что advisory-блокировка – такой же объект в менеджере блокировок, как и остальные. Если разные транзакции/сессии будут дожидаться друг друга по советующим блокировкам, **возможны взаимоблокировки** (deadlock). PostgreSQL **умеет обнаруживать** дедлоки и с участием advisory-locks, поскольку они участвуют в общем графе блокировок. Поэтому правила те же: если приложение берёт несколько advisory-lock, лучше всегда делать это в одном и том же порядке, иначе велика вероятность тупика. Впрочем, явные взаимоблокировки на уровне приложенческих локов легче контролировать, так как логика их расставления находится в коде приложения.

Мониторинг: advisory-блокировки видны в `pg_locks` с типом `advisory` [71]. Например, после выполнения `pg_advisory_lock(243773337)` мы увидим строку с `locktype = 'advisory'`, `objid = 243773337` (идентификатор), и режимом `ExclusiveLock`, granted = true [72]. Если кто-то ждет advisory-ресурс, будет аналогичная запись с granted = false. При необходимости можно с помощью SQL-функций освобождать блокировки (`pg_advisory_unlock(...)`), проверять их наличие (`pg_try_advisory_lock` не блокирует, а возвращает признак успеха). Память под рекомендательные блокировки ограничена общей таблицей блокировок (они занимают слоты наряду с остальными heavy-locks).

Подводя итог, **советующие блокировки** – мощный инструмент для синхронизации, не затрагивающей непосредственно данные в таблицах. При правильном использовании они *не влияют на транзакционную целостность* (Postgres не знает о семантике этих локов) и позволяют достичь высокой параллельности, когда стандартных средств недостаточно. Однако неосторожное применение (например, забыли снять лок) может привести к зависанию логики приложения, поэтому обращаться с ними нужно осмотрительно.

MVCC и предикатные блокировки (Serializable & SSI)

MVCC (Multi-Version Concurrency Control) – основа конкурентности в PostgreSQL. Как уже отмечалось, MVCC обеспечивает изоляцию транзакций *без традиционных блокировок на чтение*:

каждое чтение работает со **снимком** данных. В результате блокировки, получаемые при чтении, не конфликтуют с блокировками для записи [1] – поэтому SELECT не останавливает параллельный UPDATE, а UPDATE не мешает SELECT. Вместо этого при модификации данных СУБД создает новую версию строки, а все читающие транзакции продолжают видеть старую версию до своего завершения. Такой подход радикально снижает конкуренцию за блокировки и обычно повышает производительность [73]. Даже для самого строгого уровня изоляции (`SERIALIZABLE`) Postgres сохраняет это свойство: параллельные чтения и записи не блокируют друг друга, а полная изоляция достигается другим способом – при помощи **SSI (Serializable Snapshot Isolation)** [74].

Предикатные блокировки (predicate locks) – ключевой элемент SSI. Несмотря на название, это не совсем «блокировки» в обычном смысле. Их цель – *отслеживание потенциальных конфликтов* между сериализуемыми транзакциями, а не непосредственная блокировка ресурсов. Термин «предикатная блокировка» исторически означает блокировку результата некоторого условного выражения (предиката) вместо конкретных строк [75]. В старых реализациях Serializable пытались блокировать *условия запросов*, чтобы предотвратить появление «фантомов» (новых строк, удовлетворяющих условию где ранее таких не было) [76]. Однако вычислительно точно блокировать произвольный предикат сложно, поэтому PostgreSQL пошел другим путем: он реализует уровень `SERIALIZABLE` **поверх snapshot-изоляции**, анализируя зависимости транзакций на этапе фиксации [77].

Когда транзакции работают в уровне изоляции Serializable, система отслеживает два типа потенциально опасных зависимостей [78]: - RW-зависимость: транзакция A прочла строку, которую потом изменила транзакция B. - WR-зависимость: транзакция A изменила строку, которую потом прочла транзакция B.

Если такие зависимости образуют цикл между тремя и более транзакциями, может проявиться аномалия, нарушающая сериализуемость. **Предикатные блокировки фиксируют факты чтения определенных данных**, чтобы обнаруживать RW-зависимости, которых не видно через обычные блокировки записей [79]. Конкретно, при выполнении запросов в Serializable Postgres помечает: - все *прочитанные строки* – как бы ставит на них пометку (SIReadLock); - все *прочитанные диапазоны страниц* индекса или даже всю таблицу – чтобы учесть диапазон, из которого выполнялась выборка (это нужно для ловли «фантомов»).

Важно, что такие «блокировки» **не предотвращают действия**. Например, если транзакция A прочитала «предикат» (условие) *price > 100* в таблице товаров, она получит предикатную блокировку на этот диапазон значений. Транзакция B, пытающаяся добавить новый товар с price=150, *не будет заблокирована* сразу – она успешно вставит строку. Однако при попытке зафиксировать обе транзакции механизм SSI проанализирует их операции: транзакция B добавила строку, которая подпадает под условие, читанное транзакцией A, значит возникла опасная ситуация (фантом). В этот момент **одна из транзакций (обычно та, что фиксируется позже)** будет принудительно прервана с ошибкой сериализации [80]. Таким образом, *предикатные блокировки служат для детектирования конфликтов*, а не для их профилактики путём остановки исполнения.

Технически, каждая предикатная блокировка – это запись в специальной структуре в shared memory (отдельная таблица предикатных блокировок). Она характеризуется типом объекта: **отношение, страница или конкретная строка (версия)**, и специальным режимом `SIReadLock` [81] [82]. Других режимов нет – все предикатные блокировки однотипны. Например, если сериализуемая транзакция выполняет seq scan таблицы без индекса, то она получит одну запись о блокировке всего отношения (типа `relation`, режим SIReadLock) [82]. Если же

используется индексный доступ по диапазону, то помечаются конкретные строки, которые были прочитаны, *и* страницы индекса, которые были просмотрены [83] [84]. Это логически означает «транзакция читал такие-то существующие строки и такой-то диапазон, ничего не пропустил».

Чтобы предикатных блокировок не накапливалось слишком много, в PostgreSQL предусмотрен **механизм повышения уровня (lock escalation)** *только для них*. Параметры конфигурации `max_pred_locks_per_transaction` (по умолчанию 64) и `max_pred_locks_per_page` (по умолчанию 2) определяют, сколько версий строк можно пометить на одной странице прежде чем пометка «повышается» до уровня страницы, и сколько страниц – прежде чем до уровня отношения [85] [86] [87]. Например, если транзакция в Serializable прочитала много строк на одной странице, то вместо десятков записей о каждой строке в таблице предикатных блокировок останется одна запись о блокировке всей страницы [85]. А если очень много страниц – то одна запись на всю таблицу. Это экономит память, но делает гранулярность грубее (что повышает шанс ложного конфликта). Впрочем, такие ситуации редки, а параметры можно настроить под нагрузку. Максимальное общее число предикатных блокировок в системе ограничено произведением `max_pred_locks_per_transaction * max_connections` (по умолчанию 64*100=6400) [88], память под них выделяется при старте сервера [86]. Если лимит исчерпан – новые сериализуемые транзакции получать ошибку, поэтому в высоконагруженной системе стоит увеличивать эти значения.

Мониторинг и вывод. Предикатные блокировки тоже видны в `pg_locks`, но косвенно: они отображаются как блокировки типа `relation`, `page` или `tuple` с режимом `SIReadLock` [89] [81]. Например, выше упомянутый seq scan дал `relation | pred | SIReadLock`, а индексный пример – две записи `tuple | pred | SIReadLock` (две строки) и `page | pred_n_idx | SIReadLock` (номер страницы индекса) [84]. Обратите внимание: `granted` для них всегда TRUE (они ставятся сразу при чтении), ожидания по ним не бывает, так как они никого не блокируют. Вместо ожидания возможен только откат транзакции при фиксации. После COMMIT/ROLLBACK предикатные блокировки не сразу удаляются – они могут сохраниться, пока не завершатся все пересекающиеся по времени транзакции, чтобы корректно проанализировать всю историю зависимостей [90]. Но вручную ими управлять нельзя – это полностью автоматический механизм.

Выводы по MVCC и блокировкам. MVCC позволяет большинству операций выполняться без блокировок, удерживаемых до конца транзакции. Для обеспечения же строгой сериализуемости MVCC дополнен предикатными блокировками, которые не мешают транзакциям работать параллельно, но фиксируют потенциальные конфликты. Таким образом, *на уровне READ COMMITTED и REPEATABLE READ* PostgreSQL полагается на строчные и табличные блокировки для предотвращения конфликтов (при этом чтения не блокируют, что даёт отличную параллельность). *На уровне SERIALIZABLE* СУБД добавляет контроль через предикатные блокировки (SSI) – это снижает параллелизм (потому что иногда транзакции откатываются при конфликте), но гарантирует отсутсвие аномалий. В любом случае, MVCC при правильном использовании обеспечивает более высокую пропускную способность, чем грубое блокирование всего и вся [91]. Разработчикам рекомендуется избегать неоправданно высокого уровня изоляции и явных блокировок, если можно обойтись снапшот-изоляцией – тогда система покажет наилучшие показатели параллелизма.

Взаимоблокировки и типичные ситуации блокировок

Чтобы завершить обзор, рассмотрим еще раз **возможность взаимоблокировок** и примеры типичных сценариев, приводящих к блокированию.

Deadlock (взаимная блокировка) возникает, когда две или более транзакции образуют цикл ожидания ресурсов друг друга. Мы уже упоминали сценарии: для **строчных блокировок** – разные порядки обновления строк (пример с переводом денег между счетами). Для **табличных** – разные порядки захвата таблиц или смешанные режимы. Также возможны паттерны с участием **advisory-locks** (например, если два процесса берут две разные советующие блокировки в разном порядке – получить дедлок очень просто). PostgreSQL оснащен детектором дедлоков: по истечении *deadlock_timeout* (по умолчанию 1 сек) любая продолжающаяся блокировка инициирует построение **графа ожидания** и поиск циклов [17]. Найдя цикл, сервер немедленно прерывает одну из транзакций (обычно ту, что последней начала ждать) с ошибкой *ERROR: deadlock detected* [18]. Все ее изменения откатываются, тем самым другие транзакции получают шанс продолжить работу (их блокировки освобождаются). В логе при этом фиксируется подробный расклад, какие процессы и что не поделили [52]. **Важно:** собственно обнаружение дедлока – довольно редкое событие, указывающее на логическую проблему в приложении [92]. Постоянные deadlock-ошибки лучше решать корректировкой порядка операций или добавлением явных блокировок в нужных местах (чтобы конфликт выявлялся раньше, в удобном для вас месте, и обработался, например, повтором транзакции).

Пример 1: взаимоблокировка двух UPDATE. Первый транзакция обновляет строку A, затем строку B; второй – в обратном порядке. В результате каждая удерживает одну строку, ожидая другую. Через секунду (по умолчанию) первая обнаруживает тупик и прекращается сервером [93] [52]. Вторая получает освобождение блокировок и может завершиться (но обычно приложение тоже откатывает её, узнав о потерянном сопернике). Решение – всегда обновлять строки в одном порядке, например по возрастанию первичного ключа, тогда взаимоблокировка исключена [94].

Пример 2: блокировка из-за долгой транзакции. Транзакция 1 выполнила `UPDATE` (заблокировала несколько строк) и «зависла» в ожидании ввода пользователя. Транзакция 2 пытается обновить одну из тех же строк – и просто останавливается в ожидании. Это не дедлок (цикла нет: T2 ждет T1, а T1 ни от кого не ждет), но ситуация, потенциально ухудшающая производительность. Если T1 не нужна, ее стоит завершить как можно скорее. В противном случае T2 можно прервать по тайм-ауту (`lock_timeout`), если настроен. Выявить такого «виновника» можно посмотрев на `pg_stat_activity` (там будет T2 waiting, а T1 idle in transaction) или через запрос блокирующих пидов.

Пример 3: взаимоблокировка при вставке с внешними ключами. Транзакция A вставляет новую строку в таблицу *clients* и строку в связанную таблицу *orders* (которая ссылается на clients), но делает это двумя отдельными транзакциями; транзакция B делает аналогично для другого клиента. Если A вставила запись клиента1 и ждёт (в другой транзакции) вставки заказа, а B вставила клиента2 и тоже ждёт – они могут пересечься при проверке FK: каждый будет ждать коммита чужой транзакции вставки клиента. Получится дедлок на уровне XID. Лечение – либо вставлять парные записи в рамках *одной* транзакции, либо помечать внешние ключи как DEFERRABLE и выполнять проверку в конце (что меняет характер блокировок). Этот случай достаточно редок, но демонстрирует, что **блокировки транзакций** тоже могут участвовать во взаимоблокировке.

Пример 4: некорректное использование advisory-lock. Допустим, два процесса решили использовать `pg_advisory_lock(1)` и `pg_advisory_lock(2)` для двух ресурсов, но в разном порядке. Процесс P1: берет lock(1), потом lock(2); процесс P2: наоборот. Если P1 успел взять 1, а P2 – 2, дальше каждый будет ждать на втором вызове бесконечно. Deadlock-detector это через секунду обнаружит и убьет один из процессов с *deadlock detected*. Решение – договориться о порядке (напр. всегда брать ресурсы по возрастанию ID) или объединить два ресурса в один (что

не всегда возможно). В общем, с advisory-блокировками надо соблюдать ту же дисциплину, что и с обычными.

Заключение. В Postgres Pro 16 реализован гибкий и многоуровневый механизм блокировок. **Многоверсионность (MVCC)** позволяет выполнять большинство чтений без блокировок и тем самым обеспечивает высокую степень параллелизма. **Блокировки строк** гарантируют целостность при одновременных изменениях отдельных записей, минимально влияя на других пользователей. **Блокировки таблиц** защищают структуры и данные целиком там, где это необходимо (например, DDL), хотя и могут снижать параллелизм, поэтому применяются экономно или кратковременно. **Блокировки объектов** предохраняют от конфликтующих изменений каталога и других сущностей СУБД, обычно оставаясь незаметными приложению. **Блокировки транзакций** связывают транзакции в единый граф ожидания, позволяя отслеживать зависимость «транзакция ждет транзакцию». **Рекомендательные блокировки** дают разработчикам инструмент тонкой синхронизации вне стандартной модели данных. А **предикатные блокировки** дополняют MVCC, помогая выявлять трудноуловимые конфликты на уровне Serializable, не препятствуя при этом нормальной работе транзакций.

Понимание всех этих типов блокировок и их работы – ключ к написанию эффективных и корректных приложений под PostgreSQL. Правильно спроектированная система будет редко сталкиваться с блокировками и уж тем более дедлоками. Но если они и возникают, PostgreSQL предоставляет средства диагностики (`pg_locks`, `pg_stat_activity` и др.) и автоматические механизмы (детектор взаимоблокировок) [17], чтобы их обнаружить и разрешить. Пользователю же остается грамотно использовать транзакции и при необходимости – явные блокировки, всегда помня о принципах: как можно более **короткие транзакции**, **консистентный порядок** захвата ресурсов и **адекватный уровень изоляции**. Тогда Postgres Pro сможет обеспечить максимальную производительность без нарушения целостности данных.

Sources:

1. Официальная документация Postgres Pro (PostgreSQL 16), раздел *«Управление конкурентным доступом»*, подразделы о блокировках таблиц и строк [95] [96] [42], представление `pg_locks` [49] [51], введение по MVCC [1].
2. Блог Postgres Professional – серия статей Е. Рогова «Блокировки в PostgreSQL», ч.1–4. В частности, описание строчных блокировок и их реализации через xmax [2] [5], обзор блокировок объектов, расширения и страниц [54] [59], примеры предикатных блокировок и SSI [77] [82], а также разбор взаимоблокировок и задержки deadlock_timeout [17] [18].

---

[1] [73] [74] [91] Postgres Pro Standard : Документация: 17: 13.1. Введение : Компания Postgres Professional
https://postgrespro.ru/docs/postgrespro/current/mvcc-intro

[2] [3] [4] [5] [7] [9] [10] Блокировки в PostgreSQL: 2. Блокировки строк / Хабр
https://habr.com/ru/companies/postgrespro/articles/463819/

[6] [20] [49] [50] [51] [64] [65] [66] [71] PostgreSQL : Документация: 16: 54.12. pg_locks : Компания Postgres Professional
https://postgrespro.ru/docs/postgresql/16/view-pg-locks

[8] [11] [12] [13] [14] [15] [16] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [95] [96] PostgreSQL : Документация: 16: 13.3. Явные блокировки : Компания Postgres Professional
https://postgrespro.ru/docs/postgresql/16/explicit-locking

17 18 52 53 54 55 56 57 58 59 60 61 62 63 67 68 69 70 72 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 92 93 94 Блокировки в PostgreSQL: 3. Блокировки других объектов / Хабр
https://habr.com/ru/companies/postgrespro/articles/465263/

19 21 22 44 45 46 PostgreSQL : Документация: 16: LOCK : Компания Postgres Professional
https://postgrespro.ru/docs/postgresql/16/sql-lock

47 48 PostgreSQL : Документация: 16: 20.12. Управление блокировками : Компания Postgres Professional
https://postgrespro.ru/docs/postgresql/16/runtime-config-locks