

Блокировки Блокировки объектов



Авторские права

© Postgres Professional, 2016–2025

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

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

Использование материалов курса

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

Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

edu@postgrespro.ru

Отказ от ответственности

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

Общая информация о блокировках

Блокировки отношений и других объектов

Предикатные блокировки

Задача и механизм использования блокировок

Блокируемые ресурсы

Факторы, влияющие на эффективность

Время жизни блокировок

Задача: упорядочение конкурентного доступа к разделяемым ресурсам

Механизм

перед обращением к ресурсу процесс захватывает блокировку, после обращения — освобождает
блокировки приводят к очередям и ожиданиям

Альтернативы

многоверсионность — несколько версий данных
оптимистические блокировки — процессы не блокируются, но при неудачном стечении обстоятельств возникает ошибка

Блокировки используются, чтобы упорядочить конкурентный доступ к разделяемым ресурсам.

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

Блокировки не нужны, если нет конкуренции (одновременно к данным обращается только один процесс) или если нет разделяемого ресурса (например, общий буферный кеш нуждается в блокировках, а локальный — нет).

Перед тем как обратиться к ресурсу, защищенному блокировкой, процесс должен захватить эту блокировку. Когда процесс больше не нуждается в ресурсе, он освобождает блокировку, чтобы ресурсом могли воспользоваться другие процессы.

Захват блокировки возможен не всегда: ресурс может оказаться уже занятым кем-то другим. Тогда процесс либо встает в очередь ожидания, либо повторяет попытку захвата блокировки через определенное время. Так или иначе это приводит к тому, что процесс вынужден простаивать в ожидании освобождения блокировки.

Иногда удастся применить другие, неблокирующие, стратегии. Например, в одноименном модуле мы обсуждали механизм многоверсионности. Еще один пример — оптимистические блокировки, которые не блокируют процесс, но в случае неудачи приводят к ошибке.

Ресурс

все, что можно идентифицировать

Примеры ресурсов

реальные хранимые объекты: страницы, таблицы, строки и т. п.

структуры данных в общей памяти (хеш-таблицы, буферы...)

абстрактные ресурсы (число)

Ресурсом, защищаемым блокировкой, в принципе может быть все, что угодно, лишь бы ресурс можно было однозначно идентифицировать.

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), версия строки (страница и позиция 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 числом).

Гранулярность блокировки

степень детализации, уровень в иерархии ресурсов

например: таблица → страница → строки, хеш-таблица → корзины

выше гранулярность — больше возможностей для параллелизма

Режимы блокировок

совместимость режимов определяется матрицей

больше совместимых режимов — больше возможностей для параллелизма

На эффективность блокировок оказывает влияние много факторов, из которых мы выделим всего несколько.

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 процесс заинтересован всего в нескольких строках, а блокировка устанавливается на уровне таблицы, другие процессы не смогут работать с остальными строками.

Поэтому чем выше гранулярность — тем больше возможностей для распараллеливания, но это приводит к увеличению числа блокировок, информацию о которых надо где-то хранить.

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, принято называть *исключительным* (exclusive). Если режимы совместимы, блокировка может захватываться несколькими процессами одновременно; такие режимы называют *разделяемыми* (shared).

В целом, чем больше можно найти режимов, совместимых друг с другом, тем больше возможностей для параллелизма.

Долговременные блокировки

обычно захватываются до конца транзакции
и относятся к хранимым данным

большое число режимов

развитая «тяжеловесная» инфраструктура, мониторинг

Краткосрочные блокировки

обычно захватываются на доли секунды
и относятся к структурам в оперативной памяти

минимум режимов

«легковесная» инфраструктура, мониторинг может отсутствовать

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.

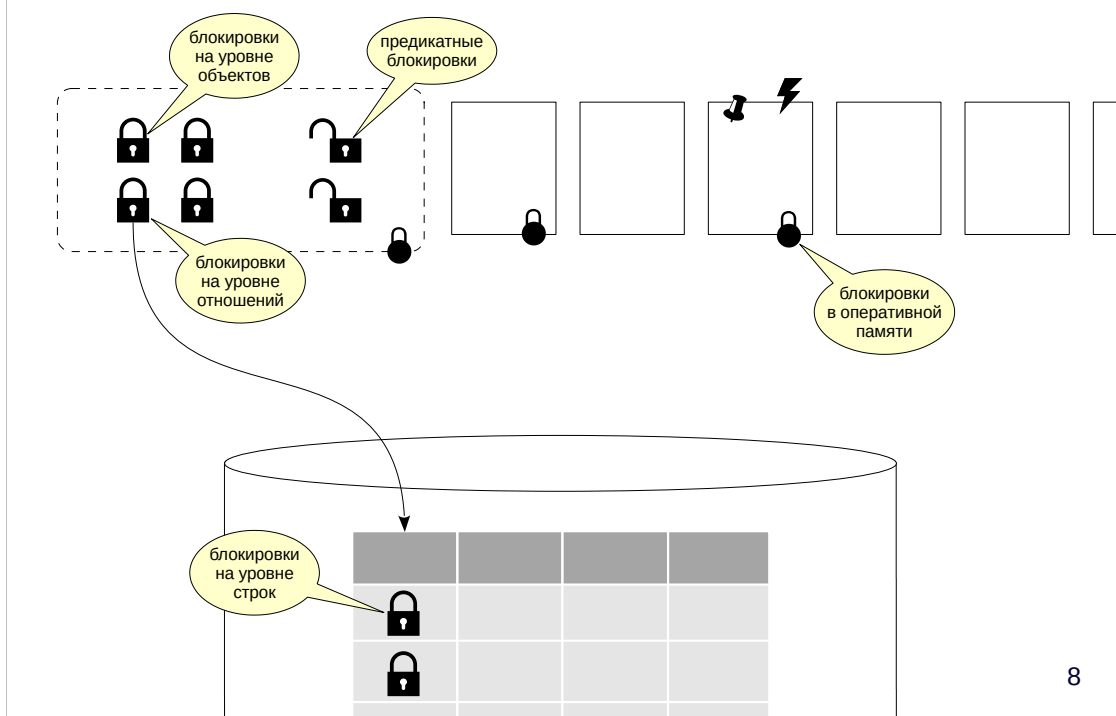
Для длительных блокировок характерно большое число режимов, чтобы допускать как можно больше одновременных действий над данными.

Обычно для долговременных блокировок предусмотрены развитая инфраструктура (например, поддержка очередей и обнаружение взаимоблокировок) и средства мониторинга.

Краткосрочные блокировки захватываются на небольшое время (от нескольких тактов 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 ограниченный набор простых режимов (часто всего лишь исключительный и разделяемый) и простая инфраструктура. Средства мониторинга могут отсутствовать.

Виды блокировок



В PostgreSQL используются разные виды блокировок.

Блокировки на уровне объектов относятся к долговременным, «тяжеловесным». В качестве ресурсов здесь выступают **отношения** и другие объекты (например, схемы).

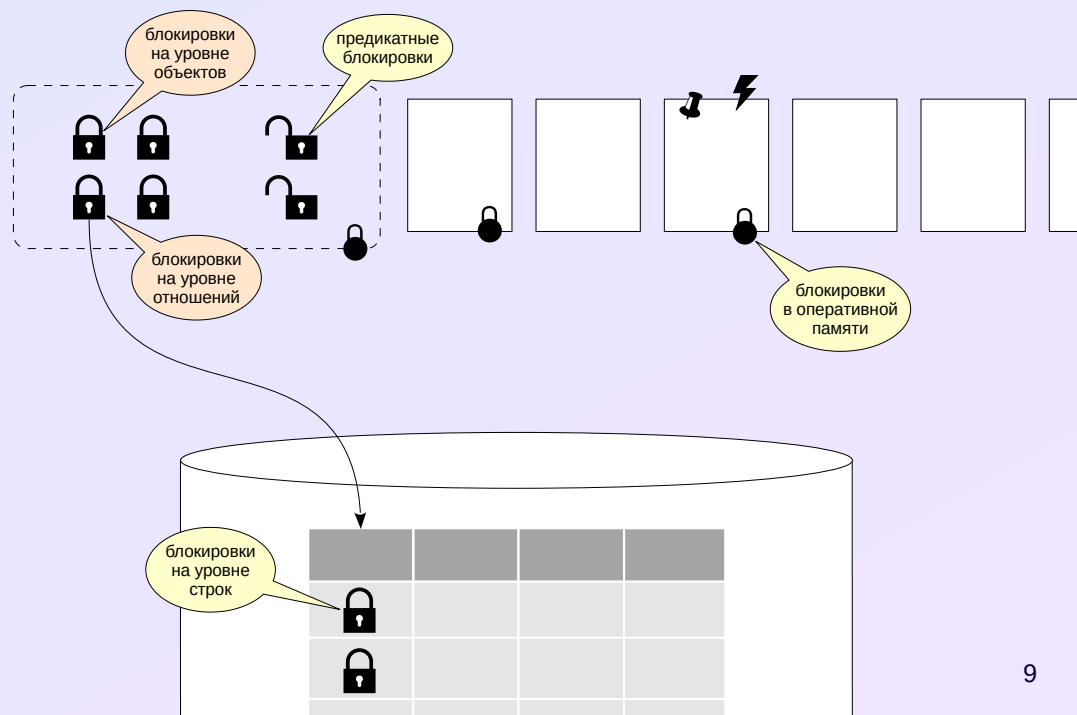
Еще один класс блокировок (оптимистических) — **предикатные**.

Информация обо всех этих блокировках хранится однотипным образом в оперативной памяти. Перечисленные виды блокировок подробно рассматриваются дальше в данной теме.

Среди долговременных блокировок особо выделяются **блокировки на уровне строк**. Их реализация отличается от остальных видов долговременных блокировок из-за потенциально огромного их количества (представьте обновление миллиона строк). Эти блокировки будут рассмотрены в теме «Блокировки строк».

К краткосрочным блокировкам относятся различные блокировки структур оперативной памяти. Они рассматриваются в теме «**Блокировки в оперативной памяти**».


Блокировки объектов



Информация в общей памяти сервера

представление `pg_locks`:
`locktype` — тип блокируемого ресурса,
`mode` — режим блокировки

Ограниченное количество

 $max_locks_per_transaction \times max_connections$

Инфраструктура

очередь ожидания: ждущие процессы не потребляют ресурсы
обнаружение взаимоблокировок

Начнем рассмотрение блокировок с блокировок уровня объектов, таких как таблицы, индексы, страницы, номера транзакций и др. Такие блокировки защищают объекты от одновременного изменения или использования в то время, когда объект изменяется, а также для ряда других нужд.

Информация о блокировках объектов располагается в общей памяти сервера. Их количество ограничено произведением значений двух параметров: `max_locks_per_transaction` и `max_connections` (по умолчанию 64×100). Этот пул блокировок — общий для всех транзакций, то есть одна транзакция вполне может захватить больше блокировок, чем `max_locks_per_transaction`; важно лишь, чтобы общее число блокировок в системе не превысило установленный предел.

Все блокировки можно увидеть в представлении `pg_locks`.

Если ресурс уже заблокирован, и транзакция пытается захватить блокировку в несовместимом режиме, то она становится в очередь и ожидает освобождения блокировки. Ожидающие транзакции не потребляют ресурсы процессора, они «засыпают» и пробуждаются только при освобождении ресурса. Ряд команд SQL позволяют указать ключевое слово `NOWAIT`: в этом случае попытка захватить занятый ресурс приводит не к ожиданию, а к ошибке.

Возможна ситуация взаимоблокировки (тупика), при которой две или более транзакций ждут друг друга. PostgreSQL автоматически определяет такие ситуации и аварийно прерывает бесконечное ожидание.

Тип ресурса: Relation

Режимы

Access Share	SELECT	} допускают параллельное изменение данных в таблице
Row Share	SELECT FOR UPDATE/SHARE	
Row Exclusive	UPDATE, DELETE, INSERT, MERGE	
Share Update Exclusive	VACUUM, ANALYZE, ALTER TABLE, CREATE INDEX CONCURRENTLY	
Share	CREATE INDEX	
Share Row Exclusive	CREATE TRIGGER, ALTER TABLE	
Exclusive	REFRESH MATERIALIZED VIEW CONCURRENTLY	
Access Exclusive	DROP, TRUNCATE, REINDEX, VACUUM FULL, LOCK TABLE, ALTER TABLE, REFRESH MATERIALIZED VIEW	

11

Важный частный случай блокировок — блокировки отношений (таблиц, индексов, последовательностей и т. п.). Такие блокировки имеют тип **relation** в представлении `pg_locks`.

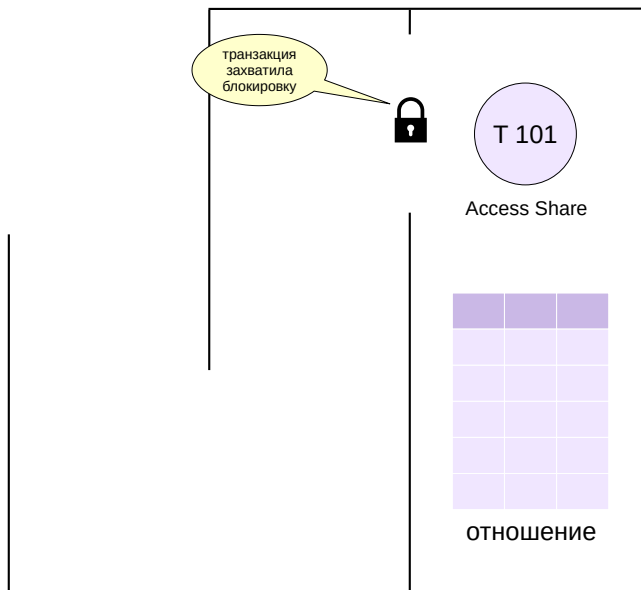
Для них определено целых 8 различных режимов, которые показаны на слайде вместе с примерами команд SQL, использующих эти режимы. Матрица совместимости, которая показывает, какие блокировки можно захватывать совместно, приведена в документации.

Такое количество режимов существует для того, чтобы позволить выполнять одновременно как можно большее количество команд, относящихся к одной таблице (индексу и т. п.).

Самый слабый режим — Access Share, он захватывается командой SELECT и совместим с любым режимом, кроме самого сильного — Access Exclusive. Это означает, что запрос не мешает ни другим запросам, ни изменению данных в таблице, ни чему-либо другому, но не дает, например, удалить таблицу в то время, когда из нее читаются данные.

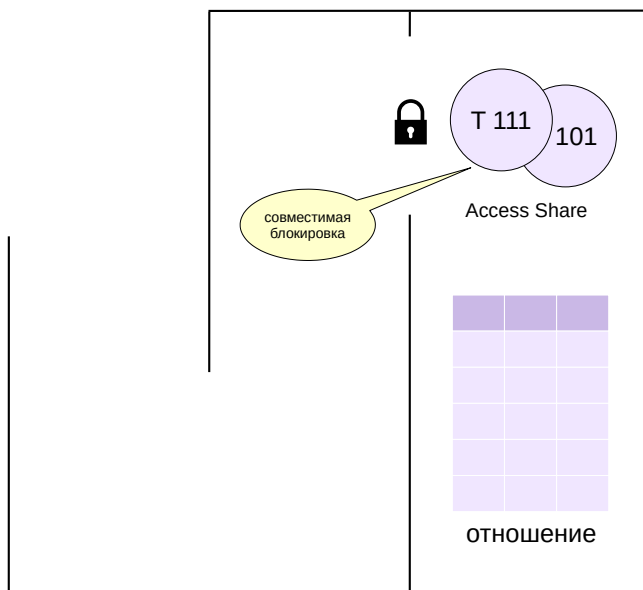
Другой пример: режим Share (как и другие более сильные режимы) не совместим с изменением данных в таблице. Например, команда CREATE INDEX заблокирует команды INSERT, UPDATE и DELETE (и наоборот). Поэтому существует команда CREATE INDEX CONCURRENTLY, использующая режим Share Update Exclusive, который совместим с такими изменениями (за счет этого команда выполняется дольше).

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

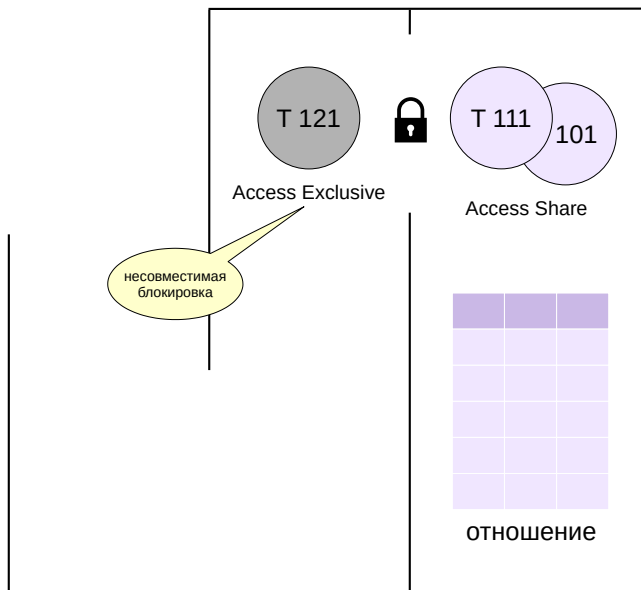


Чтобы лучше представить, к чему приводит появление несовместимой блокировки, можно посмотреть на приведенный пример.

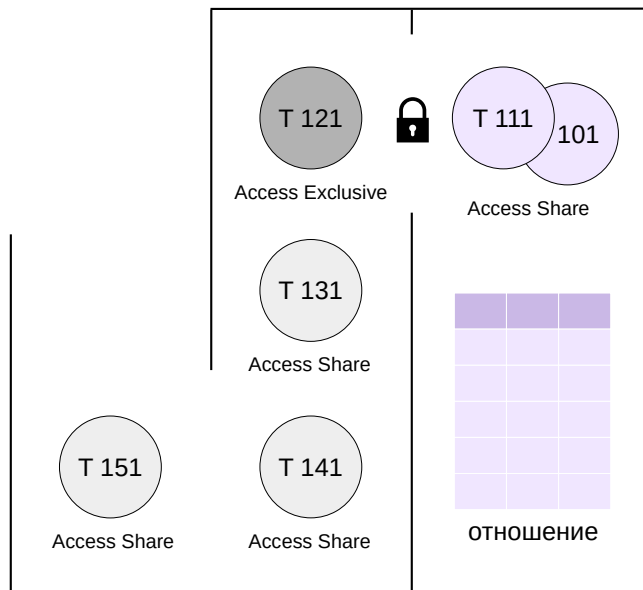
Вначале на таблице выполняется команда `SELECT`, которая запрашивает и получает блокировку уровня Access Share.



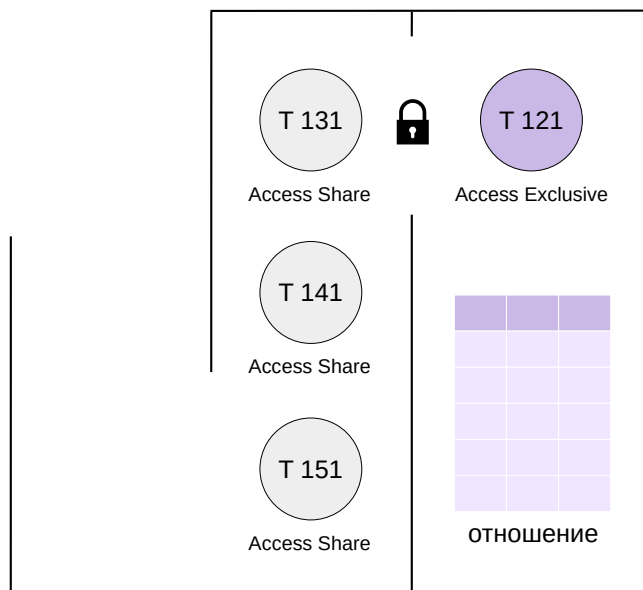
Затем еще одна команда `SELECT` запрашивает блокировку уровня Access Share и получает ее, так как запрошенная блокировка совместима с текущей.



Затем администратор выполняет команду `VACUUM FULL`, которой требуется блокировка уровня `Access Exclusive`, несовместимая с текущими `Access Share`. Транзакция встает в очередь.



Затем в системе появляются еще несколько команд SELECT. И, хотя какие-то из них теоретически могли бы «проскочить вперед», пока VACUUM FULL ждет своей очереди, все они честно занимают место за VACUUM FULL.



После того, как две первые транзакции с командами SELECT завершаются и освобождают блокировки, VACUUM FULL начинает выполняться.

Теперь только когда VACUUM FULL завершит свою работу и снимет (исключительную) блокировку, все накопившиеся в очереди команды SELECT смогут захватить блокировки уровня Access Share и начать выполняться.

Таким образом, не вовремя выполненная команда может парализовать работу системы на время, значительно превышающее время выполнения самой команды.

Обратите внимание: исключительная блокировка может потребоваться также обычной очистке (и автоочистке), чтобы в конце своей работы выполнить усечение таблицы, то есть «откусить» пустой хвост файла данных и вернуть место операционной системе. Очистка не должна допускать ситуаций долгого ожидания, поэтому в случае проблем этап усечения можно отключить параметром хранения `vacuum_truncate` или вызывая очистку с указанием `VACUUM (truncate off)`.

Типы ресурсов

Extend — добавление страниц к файлу отношения

Object — не-отношение: база данных, схема и т. п.

Page — страница (используется некоторыми типами индексов)

Tuple — версия строки

Advisory — рекомендательная блокировка

Transactionid — транзакция

Virtualxid — виртуальная транзакция

Режимы

исключительный

разделяемый

Кроме отношений есть еще несколько типов блокируемых ресурсов. Их блокировки захватываются либо только в исключительном режиме, либо в исключительном и разделяемом. К ним относятся:

- **Extend** при добавлении страниц к файлу какого-либо отношения;
- **Object** для блокирования объекта, который не является отношением (примеры таких объектов: база данных, схема, подписка и т. п.);
- **Page** для блокирования страницы (редкая блокировка, используется некоторыми типами индексов);
- **Tuple** используется в некоторых случаях для установки приоритета среди нескольких транзакций, ожидающих блокировку одной строки (подробнее см. тему «Блокировка строк» этого модуля);
- **Advisory** для рекомендательных блокировок (о них чуть позже).

Особый интерес представляют блокировки типа **Transactionid** и **Virtualxid**. Каждая транзакция удерживает исключительную блокировку своих номеров: и виртуального, и настоящего, если он есть. Это дает простой способ дождаться окончания какой-либо транзакции: надо запросить блокировку ее номера. Подробнее это рассматривается в теме «Блокировки строк».

Вывод сообщений в журнал сервера

параметр *log_lock_waits*:

выводит сообщение об ожидании дольше *deadlock_timeout*

Текущие блокировки

представление *pg_locks*

функция *pg_blocking_pids*

Возникающие в системе блокировки необходимы для обеспечения целостности и изоляции, однако могут приводить к нежелательным ожиданиям. Такие ожидания можно отслеживать, чтобы разобраться в их причине и по возможности устранить (например, изменив алгоритм работы приложения).

Один способ мониторинга 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, здесь речь идет об обычных ожиданиях).

Второй способ состоит в том, чтобы в момент возникновения долгой блокировки (или на периодической основе) выполнять запрос к представлению *pg_locks*, смотреть на блокируемые и блокирующие транзакции (функция *pg_blocking_pids*) и расшифровывать их при помощи *pg_stat_activity*.

Блокировки отношений и других объектов

Создадим таблицу «банковских» счетов. В ней будем хранить номер счета и сумму.

```
=> CREATE DATABASE locks_objects;
```

```
CREATE DATABASE
```

```
=> \c locks_objects
```

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

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

```
CREATE TABLE
```

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

```
INSERT 0 3
```

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

```
| => \c locks_objects
```

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

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

```
| pg_backend_pid  
| -----  
|          23987  
| (1 row)
```

```
| => BEGIN;
```

```
| BEGIN
```

Какие блокировки удерживает только что начавшаяся транзакция?

```
=> SELECT locktype, relation::regclass, virtualxid AS virtxid, transactionid AS xid, mode, granted  
FROM pg_locks WHERE pid = 23987;
```

```
locktype | relation | virtxid | xid | mode | granted  
-----+-----+-----+-----+-----+-----  
virtualxid |          | 3/13    |    | ExclusiveLock | t  
(1 row)
```

Только блокировку собственного виртуального номера.

Теперь обновим строку таблицы. Как изменится ситуация?

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

```
| UPDATE 1
```

```
=> SELECT locktype, relation::regclass, virtualxid AS virtxid, transactionid AS xid, mode, granted  
FROM pg_locks WHERE pid = 23987;
```

```
locktype | relation | virtxid | xid | mode | granted  
-----+-----+-----+-----+-----+-----  
relation | accounts |          |    | RowExclusiveLock | t  
virtualxid |          | 3/13    |    | ExclusiveLock | t  
transactionid |          |          | 746 | ExclusiveLock | t  
(3 rows)
```

Добавилась блокировка отношения в режиме RowExclusiveLock (что соответствует команде UPDATE) и исключительная блокировка собственного номера (который появился, как только транзакция начала изменять данные).

Теперь попробуем в еще одном сеансе создать индекс по таблице.

```
|| => \c locks_objects
```

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

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

```

pg_backend_pid
-----
24222
(1 row)

```

```

=> CREATE INDEX ON accounts(acc_no);

```

Команда не выполняется — ждет освобождения блокировки. Какой?

```

=> SELECT locktype, relation::regclass, virtualxid AS virtxid, transactionid AS xid, mode, granted,
to_char(waitstart, 'HH24:MI:SS') AS waitstart FROM pg_locks WHERE pid = 24222;

```

```

locktype | relation | virtxid | xid | mode | granted | waitstart
-----+-----+-----+-----+-----+-----+-----
virtualxid | accounts | 4/6 | | ExclusiveLock | t | 
relation | accounts | | | ShareLock | f | 23:46:56
(2 rows)

```

Видим, что транзакция пыталась получить блокировку таблицы в режиме ShareLock, но не смогла (granted = f). Столбец waitstart в этом случае показывает время, когда обслуживающий процесс начал ожидать блокировку.

Мы можем найти номер блокирующего процесса (в общем виде — несколько номеров)...

```

=> SELECT pg_blocking_pids(24222);

```

```

pg_blocking_pids
-----
{23987}
(1 row)

```

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

```

=> SELECT * FROM pg_stat_activity
WHERE pid = ANY(pg_blocking_pids(24222)) \gx

```

```

-[ RECORD 1 ]-----+-----
datid          | 16390
datname        | locks_objects
pid            | 23987
leader_pid     | 
usesysid       | 16384
username       | student
application_name | psql
client_addr    | 
client_hostname | 
client_port    | -1
backend_start  | 2025-06-23 23:46:55.67936+03
xact_start     | 2025-06-23 23:46:55.776661+03
query_start    | 2025-06-23 23:46:55.868022+03
state_change   | 2025-06-23 23:46:55.868728+03
wait_event_type | Client
wait_event     | ClientRead
state          | idle in transaction
backend_xid    | 746
backend_xmin   | 
query_id       | 
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

```

Устанавливаются приложением

Продолжительность

до конца сеанса

до конца транзакции

Режимы

исключительный

разделяемый

Рекомендательные блокировки можно использовать, если нужна логика блокирования, которую неудобно реализовывать с помощью других, «обычных» блокировок. Рекомендательные блокировки устанавливаются только приложением; PostgreSQL никогда не делает этого автоматически.

Имена функций, связанных с рекомендательными блокировками, начинаются с `pg_advisory` и могут содержать уточняющие ключевые слова.

Рекомендательная блокировка может быть установлена до конца сеанса (функция `pg_advisory_lock`) или до конца транзакции (`pg_advisory_xact_lock`).

Блокировки уровня сеанса, в отличие от блокировок уровня транзакции, не освобождаются автоматически по окончании транзакции, их нужно явно освобождать функцией `pg_advisory_unlock`. Таким образом, сеансовые рекомендательные блокировки нарушают обычную логику транзакций — блокировка, полученная в транзакции, даже если произойдет откат этой транзакции, будет сохраняться в сеансе; аналогично, освобождение блокировки остается в силе, даже если транзакция, в которой оно было выполнено, позже прерывается.

По умолчанию рекомендательные блокировки захватываются в исключительном режиме. Для использования разделяемого режима к имени функции добавляется слово `shared`, например, `pg_advisory_lock_shared` и `pg_advisory_unlock_shared`.

<https://postgrespro.ru/docs/postgresql/16/explicit-locking#ADVISORY-LOCKS>

Рекомендательные блокировки

Начнем транзакцию.

```
| => BEGIN;  
|  
| BEGIN
```

Получим блокировку некоего условного ресурса. В качестве идентификатора используется число; если ресурс имеет имя, удобно получить это число с помощью функции хеширования:

```
| => SELECT hashtext('песыпс1');  
|  
| hashtext  
| -----  
| 243773337  
| (1 row)  
|  
| => SELECT pg_advisory_lock(hashtext('песыпс1'));  
|  
| pg_advisory_lock  
| -----  
|  
| (1 row)
```

Информация о рекомендательных блокировках доступна в pg_locks:

```
=> SELECT locktype, objid, virtualxid AS virtxid, mode, granted  
FROM pg_locks WHERE pid = 23987;
```

locktype	objid	virtualxid	mode	granted
virtualxid		3/14	ExclusiveLock	t
advisory	243773337		ExclusiveLock	t

(2 rows)

Если другой сеанс попытается захватить ту же блокировку, он будет ждать ее освобождения:

```
|| => SELECT pg_advisory_lock(hashtext('песыпс1'));
```

В приведенном примере блокировка действует до конца сеанса, а не транзакции, как обычно.

```
| => COMMIT;  
|  
| COMMIT  
|  
|=> SELECT locktype, objid, virtualxid AS virtxid, mode, granted  
FROM pg_locks WHERE pid = 23987;  
|  
| locktype | objid | virtualxid | mode | granted  
| -----+-----+-----+-----+-----  
| advisory | 243773337 | | ExclusiveLock | t  
| (1 row)
```

Захвативший блокировку сеанс может получить ее повторно, даже если есть очередь ожидания.

```
| => SELECT pg_advisory_lock(hashtext('песыпс1'));  
|  
| pg_advisory_lock  
| -----  
|  
| (1 row)
```

Блокировку можно явно освободить:

```
| => SELECT pg_advisory_unlock(hashtext('песыпс1'));  
|  
| pg_advisory_unlock  
| -----  
| t  
| (1 row)
```

Но в нашем примере блокировка была получена сеансом дважды, поэтому придется освободить ее еще раз:

```
=> SELECT locktype, objid, virtualxid AS virtxid, mode, granted
FROM pg_locks WHERE pid = 23987;
```

locktype	objid	virtualxid	mode	granted
advisory	243773337		ExclusiveLock	t

(1 row)

```
=> SELECT pg_advisory_unlock(hashtext('pecypc1'));
```

pg_advisory_unlock
t

(1 row)

pg_advisory_lock

(1 row)

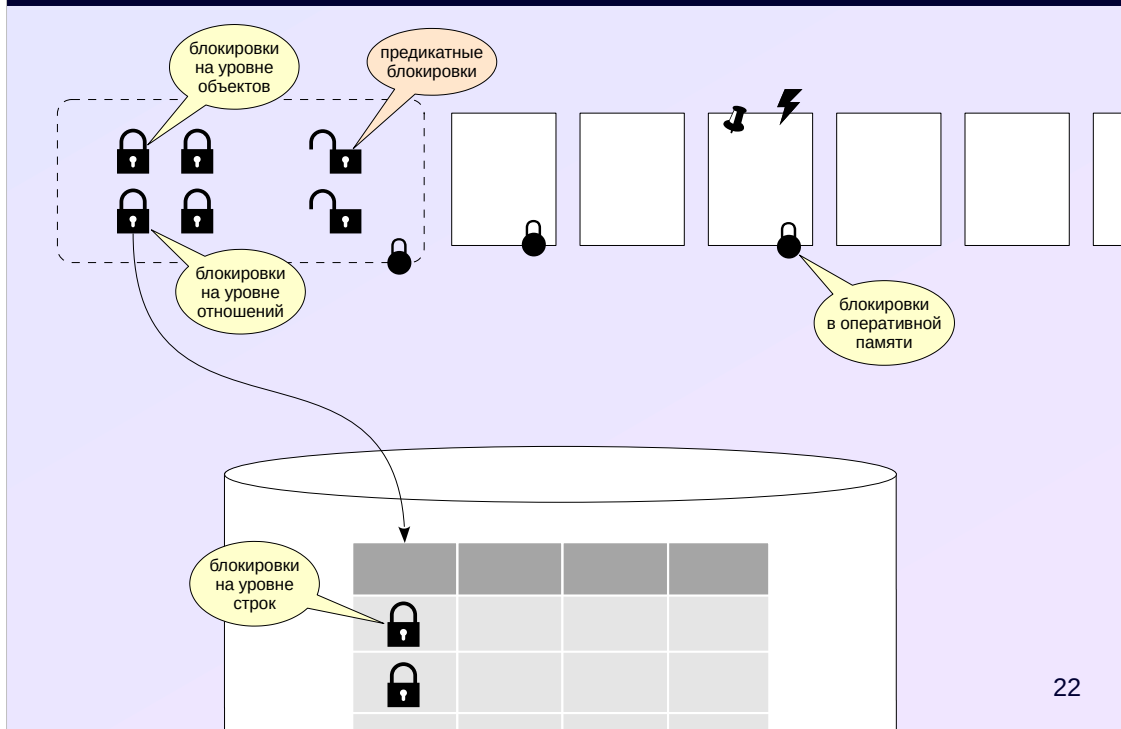
Существуют другие варианты функций для получения рекомендательных блокировок до конца транзакции, для получения разделяемых блокировок и т. п. Вот их полный список:

```
=> \df pg_advisory*
```

List of functions				
Schema	Name	Result data type	Argument data types	Type
pg_catalog	pg_advisory_lock	void	bigint	func
pg_catalog	pg_advisory_lock	void	integer, integer	func
pg_catalog	pg_advisory_lock_shared	void	bigint	func
pg_catalog	pg_advisory_lock_shared	void	integer, integer	func
pg_catalog	pg_advisory_unlock	boolean	bigint	func
pg_catalog	pg_advisory_unlock	boolean	integer, integer	func
pg_catalog	pg_advisory_unlock_all	void		func
pg_catalog	pg_advisory_unlock_shared	boolean	bigint	func
pg_catalog	pg_advisory_unlock_shared	boolean	integer, integer	func
pg_catalog	pg_advisory_xact_lock	void	bigint	func
pg_catalog	pg_advisory_xact_lock	void	integer, integer	func
pg_catalog	pg_advisory_xact_lock_shared	void	bigint	func
pg_catalog	pg_advisory_xact_lock_shared	void	integer, integer	func

(13 rows)

Предикатные блокировки



Задача: реализация уровня изоляции Serializable

используются в дополнение к обычной изоляции на снимках данных
оптимистические блокировки, название сложилось исторически

Информация в общей памяти сервера

представление pg_locks

Ограниченное количество

 $max_pred_locks_per_transaction \times max_connections$

Термин *предикатная блокировка* появился давно, еще при первых попытках реализовать полную изоляцию (Serializable) на основе блокировок в ранних СУБД. Идея состояла в том, что блокировать надо не только определенные строки, но и предикаты. Например, при выполнении запроса с условием $a > 10$ надо заблокировать диапазон $a > 10$, чтобы избежать появления фантомных строк и других аномалий.

В PostgreSQL уровень Serializable реализован поверх существующего механизма снимков данных, но термин остался. Фактически такие «блокировки» ничего не блокируют, а используются для отслеживания зависимостей транзакций по данным.

Как и для обычных блокировок, информация о предикатных блокировках отображается в представлении pg_locks, все они устанавливаются в одном специальном режиме **SIRead** (Serializable Isolation Read).

Число предикатных блокировок ограничено произведением параметров *max_pred_locks_per_transaction* и *max_connections* ($= 64 \times 100$).

Именно с использованием предикатных блокировок связано ограничение, что для достижения полной изоляции все транзакции должны работать на уровне Serializable. Отслеживание зависимостей будет работать только для транзакций, которые устанавливают предикатные блокировки.

<https://postgrespro.ru/docs/postgresql/16/transaction-iso#XACT-SERIALIZABLE>

<https://wiki.postgresql.org/wiki/SSI>

https://github.com/postgres/postgres/blob/REL_16_STABLE/src/backend/storage/lmgr/README-SSI

Типы ресурсов

Relation — отношение \rightarrow $max_pred_locks_per_relation$
Page — страница \rightarrow $max_pred_locks_per_page$
Tuple — версия строки

повышение
уровня

Режим

SIRead

Предикатные блокировки захватываются на трех уровнях.

При полном сканировании таблицы блокировка устанавливается **на уровне всей таблицы**.

При индексном сканировании устанавливаются блокировки тех **страниц индекса**, которые соответствуют условию доступа (предикату). Кроме того, устанавливаются блокировки на уровне **отдельных табличных версий строк** (это не то же самое, что блокировки строк, которые рассматриваются в одноименной теме).

При увеличении количества предикатных блокировок происходит автоматическое повышение уровня (эскалация): вместо нескольких мелких блокировок захватывается одна более высокого уровня.

Если число блокировок версий строк одной страницы превышает значение параметра $max_pred_locks_per_page$ (2 по умолчанию), вместо них захватывается одна блокировка уровня страницы.

Если число блокировок страниц или версий одной таблицы (индекса) превышает значение параметра $max_pred_locks_per_relation$, вместо них захватывается одна блокировка на все отношение. По умолчанию параметр равен -2; для отрицательных чисел значение вычисляется как $max_pred_locks_per_transaction / abs(max_pred_locks_per_relation)$.

Повышение уровня блокировок может приводить к ложным ошибкам сериализации, из-за чего снижается пропускная способность системы.

Предикатные блокировки

Начнем транзакцию с уровнем Serializable и прочитаем одну строку таблицы последовательным сканированием.

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE;

BEGIN

=> EXPLAIN (analyze, costs off, timing off) SELECT * FROM accounts LIMIT 1;

               QUERY PLAN
-----
Limit (actual rows=1 loops=1)
  -> Seq Scan on accounts (actual rows=1 loops=1)
    Planning Time: 0.145 ms
    Execution Time: 0.023 ms
(4 rows)
```

Посмотрим на блокировки:

```
=> SELECT locktype, relation::regclass, page, tuple, virtualxid AS vxid, transactionid AS xid, mode, granted
FROM pg_locks WHERE pid = 23987;
```

locktype	relation	page	tuple	vxid	xid	mode	granted
relation	accounts_acc_no_idx					AccessShareLock	t
relation	accounts					AccessShareLock	t
virtualxid				3/18		ExclusiveLock	t
relation	accounts					SIReadLock	t

(4 rows)

Появилась предикатная блокировка всей таблицы accounts (несмотря на то что читается одна строка).

```
=> COMMIT;

COMMIT
```

Теперь прочитаем одну строку таблицы, используя индекс:

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE;

BEGIN

=> SET enable_seqscan = off;

SET

=> EXPLAIN (analyze, costs off, timing off) SELECT * FROM accounts WHERE acc_no = 1;

               QUERY PLAN
-----
Index Scan using accounts_acc_no_idx on accounts (actual rows=1 loops=1)
  Index Cond: (acc_no = 1)
    Planning Time: 0.062 ms
    Execution Time: 0.061 ms
(4 rows)
```

Блокировки:

```
=> SELECT locktype, relation::regclass, page, tuple, virtualxid AS vxid, transactionid AS xid, mode, granted
FROM pg_locks WHERE pid = 23987;
```

locktype	relation	page	tuple	vxid	xid	mode	granted
relation	accounts_acc_no_idx					AccessShareLock	t
relation	accounts					AccessShareLock	t
virtualxid				3/19		ExclusiveLock	t
tuple	accounts	0	4			SIReadLock	t
page	accounts_acc_no_idx	1				SIReadLock	t

(5 rows)

При индексном сканировании устанавливаются мелкогранулярные предикатные блокировки:

- блокировки прочитанных страниц индекса;
- блокировки прочитанных версий строк.

```
=> COMMIT;
```

```
COMMIT
```

Блокировки отношений и других объектов БД используются для организации конкурентного доступа к общим ресурсам

- поддерживаются очереди и обнаружение взаимоблокировок
- рекомендательные блокировки для ресурсов, не связанных с хранимыми объектами

Предикатные блокировки используются для реализации уровня изоляции Serializable

- ничего не блокируют, отслеживают зависимости транзакций по данным

1. Какие блокировки на уровне изоляции Read Committed удерживает транзакция, прочитавшая одну строку таблицы по первичному ключу? Проверьте на практике.
2. Воспроизведите автоматическое повышение уровня предикатных блокировок при чтении строк таблицы по индексу. Покажите, что при этом возможна ложная ошибка сериализации.
3. Настройте сервер так, чтобы в журнал сообщений сбрасывалась информация о долгих (более 100 миллисекунд) ожиданиях блокировок. Воспроизведите ситуацию, при которой в журнале появятся такие сообщения.

2. В каждой из двух транзакций прочитайте две строки таблицы, используя индекс, и измените одну из прочитанных строк. Такие транзакции сериализуются. Задайте `max_pred_locks_per_relation = 2` и повторите опыт.

0. Подготовка

Создадим таблицу как в демонстрации; номер счета будет первичным ключом.

```
=> CREATE DATABASE locks_objects;
```

```
CREATE DATABASE
```

```
=> \c locks_objects
```

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

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

```
CREATE TABLE
```

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

```
INSERT 0 3
```

Нам понадобятся два дополнительных сеанса.

```
| => \c locks_objects
```

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

```
|| => \c locks_objects
```

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

1. Блокировки читающей транзакции, Read Committed

Начинаем транзакцию и читаем одну строку.

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

```
| pg_backend_pid  
|-----  
|          56019  
| (1 row)
```

```
| => BEGIN;
```

```
| BEGIN
```

```
| => SELECT * FROM accounts WHERE acc_no = 1;
```

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

Блокировки:

```
=> SELECT locktype, relation::regclass, virtualxid AS virtxid, transactionid AS xid, mode, granted  
FROM pg_locks WHERE pid = 56019;
```

locktype	relation	virtualxid	xid	mode	granted
relation	accounts_pkey			AccessShareLock	t
relation	accounts			AccessShareLock	t
virtualxid		3/14		ExclusiveLock	t

(3 rows)

Здесь мы видим:

- Блокировку таблицы accounts в режиме AccessShareLock;
- Блокировку индекса accounts_pkey, созданного для первичного ключа, в том же режиме;
- Исключительную блокировку собственного номера виртуальной транзакции.

Если смотреть блокировки в самой транзакции, к ним добавится блокировка на представление pg_locks:

```
| => SELECT locktype, relation::regclass, virtualxid AS virtxid, transactionid AS xid, mode, granted  
FROM pg_locks WHERE pid = 56019;
```

locktype	relation	virtxid	xid	mode	granted
relation	pg_locks			AccessShareLock	t
relation	accounts_pkey			AccessShareLock	t
relation	accounts			AccessShareLock	t
virtualxid		3/14		ExclusiveLock	t

(4 rows)

```
=> COMMIT;
```

COMMIT

2. Повышение уровня предикатных блокировок

В двух сеансах начинаем транзакции с уровнем Serializable.

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE;
```

BEGIN

```
|| => BEGIN ISOLATION LEVEL SERIALIZABLE;
```

|| BEGIN

В первой читаем строки счетов 1 и 3, во второй — строки счетов 2 и 3.

```
=> SELECT * FROM accounts WHERE acc_no IN (1,3);
```

acc_no	amount
1	1000.00
3	3000.00

(2 rows)

```
|| => SELECT * FROM accounts WHERE acc_no IN (2,3);
```

acc_no	amount
2	2000.00
3	3000.00

(2 rows)

Блокируются страница индекса и отдельные версии строк:

```
=> SELECT pid, locktype, relation::regclass, page, tuple, mode, granted
FROM pg_locks
WHERE mode = 'SIReadLock'
ORDER BY 1,2,3,4,5;
```

pid	locktype	relation	page	tuple	mode	granted
56019	page	accounts_pkey	1		SIReadLock	t
56019	tuple	accounts	0	1	SIReadLock	t
56019	tuple	accounts	0	3	SIReadLock	t
56059	page	accounts_pkey	1		SIReadLock	t
56059	tuple	accounts	0	2	SIReadLock	t
56059	tuple	accounts	0	3	SIReadLock	t

(6 rows)

Изменим в первом сеансе остаток первого счета, во втором — второго.

```
=> UPDATE accounts SET amount = amount + 10 WHERE acc_no = 1;
```

UPDATE 1

```
|| => UPDATE accounts SET amount = amount + 10 WHERE acc_no = 2;
```

|| UPDATE 1

Транзакции сериализуются:

```
=> COMMIT;
```

COMMIT

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

|| COMMIT

Теперь настроим повышение уровня так, чтобы при блокировке двух версий строк блокировалась вся таблица.


```
=> ALTER SYSTEM SET max_pred_locks_per_relation = 1;
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 main reload
```

Повторяем опыт.

```
| => BEGIN ISOLATION LEVEL SERIALIZABLE;
```

```
| BEGIN
```

```
|| => BEGIN ISOLATION LEVEL SERIALIZABLE;
```

```
|| BEGIN
```

```
| => SELECT * FROM accounts WHERE acc_no IN (1,3);
```

```
| acc_no | amount  
|-----+-----  
|      1 | 1010.00  
|      3 | 3000.00  
(2 rows)
```

```
|| => SELECT * FROM accounts WHERE acc_no IN (2,3);
```

```
|| acc_no | amount  
||-----+-----  
||      2 | 2010.00  
||      3 | 3000.00  
|| (2 rows)
```

Теперь каждая транзакция блокирует всю таблицу:

```
=> SELECT pid, locktype, relation::regclass, page, tuple, mode, granted  
FROM pg_locks  
WHERE mode = 'SIReadLock'  
ORDER BY 1,2,3,4,5;
```

pid	locktype	relation	page	tuple	mode	granted
56019	page	accounts_pkey	1		SIReadLock	t
56019	relation	accounts			SIReadLock	t
56059	page	accounts_pkey	1		SIReadLock	t
56059	relation	accounts			SIReadLock	t

(4 rows)

Изменяем остатки...

```
| => UPDATE accounts SET amount = amount + 10 WHERE acc_no = 1;
```

```
| UPDATE 1
```

```
|| => UPDATE accounts SET amount = amount + 10 WHERE acc_no = 2;
```

```
|| UPDATE 1
```

...и фиксируем транзакции.

```
| => COMMIT;
```

```
| COMMIT
```

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

```
|| ERROR: could not serialize access due to read/write dependencies among transactions  
|| DETAIL: Reason code: Canceled on identification as a pivot, during commit attempt.  
|| HINT: The transaction might succeed if retried.
```

Из-за повышения уровня блокировок сериализация невозможна.

3. Вывод в журнал информации о блокировках

Требуется изменить параметры:

```
=> ALTER SYSTEM SET log_lock_waits = on;
```

```
ALTER SYSTEM
```

```
=> ALTER SYSTEM SET deadlock_timeout = '100ms';
```

```
ALTER SYSTEM
```

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf
-----
t
(1 row)
```

Воспроизведем блокировку.

```
=> BEGIN;
```

```
BEGIN
```

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

```
UPDATE 1
```

```
| => BEGIN;
```

```
| BEGIN
```

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

В первом сеансе выполним задержку и после этого завершим транзакцию.

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

```
pg_sleep
-----
(1 row)
```

```
=> COMMIT;
```

```
COMMIT
```

```
| UPDATE 1
```

```
| => COMMIT;
```

```
| COMMIT
```

Вот что попало в журнал:

```
student$ tail -n 7 /var/log/postgresql/postgresql-16-main.log
```

```
2025-06-24 00:14:11.400 MSK [56019] student@locks_objects LOG: process 56019 still
waiting for ShareLock on transaction 750 after 100.115 ms
2025-06-24 00:14:11.400 MSK [56019] student@locks_objects DETAIL: Process holding the
lock: 55933. Wait queue: 56019.
2025-06-24 00:14:11.400 MSK [56019] student@locks_objects CONTEXT: while updating tuple
(0,6) in relation "accounts"
2025-06-24 00:14:11.400 MSK [56019] student@locks_objects STATEMENT: UPDATE accounts SET
amount = 100.00 WHERE acc_no = 1;
2025-06-24 00:14:12.421 MSK [56019] student@locks_objects LOG: process 56019 acquired
ShareLock on transaction 750 after 1121.229 ms
2025-06-24 00:14:12.421 MSK [56019] student@locks_objects CONTEXT: while updating tuple
(0,6) in relation "accounts"
2025-06-24 00:14:12.421 MSK [56019] student@locks_objects STATEMENT: UPDATE accounts SET
amount = 100.00 WHERE acc_no = 1;
```