**Лабораторная работа № 12\_1 Транзакции, блокировки**

Транзакции в PotsgreSQL

Транзакции — это механизм, который инкапсулирует несколько операторов в одну операцию для обработки базой данных. Вместо подачи отдельных операторов база данных может интерпретировать группу команд и воздействовать на нее как на единое целое. Это помогает обеспечить согласованность набора данных для многих тесно связанных утверждений.

Одной из причин, по которой люди используют транзакции, является получение определенных гарантий согласованности своих данных и среды, в которой они обрабатываются. Согласованность может быть нарушена разными способами, что влияет на то, как базы данных пытаются это предотвратить.

Существует четыре основных класса несогласованности в зависимости от реализации транзакции

* «Грязное» чтение - чтение данных, добавленных или изменённых транзакцией, которая впоследствии не подтвердится (откатится)
* Неповторяющееся чтение - ситуация, когда при повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными.
* Чтение строк-«фантомов» - ситуация, когда при повторном чтении в рамках одной транзакции одна и та же выборка дает разные множества строк.
* Аномалии сериализации - происходит, когда результаты нескольких транзакций, совершенных одновременно, приведут к другим результатам, чем если бы они были совершены одна за другой.

Уровни изоляции транзакций по стандарту SQL

1. Read uncommited - это уровень изоляции, который предлагает наименьшие гарантии обеспечения согласованности и изоляции данных. Хотя транзакции, использующие этот уровень изоляции, имеют определенные функции, часто связанные с транзакциями, такие как возможность одновременной фиксации нескольких операторов или откат операторов в случае возникновения ошибки, они допускают множество ситуаций, в которых может быть нарушена согласованность, а именно – на этом уровне изоляции допустимы все вышеуказанные несоглаованности. Этот уровень изоляции фактически не реализован в PostgreSQL. *Хотя PostgreSQL распознает имя уровня изоляции, на самом деле оно не поддерживается, и вместо него будет использоваться “read commited” (описано ниже).*
2. Read commited - это уровень изоляции, который специально защищает от «грязного» чтения. Когда транзакции используют этот уровень изоляции, незафиксированные данные никогда не могут повлиять на внутренний контекст транзакции. Это обеспечивает базовый уровень согласованности, гарантируя, что незафиксированные данные никогда не повлияют на транзакцию, но допускают остальные классы несоглаованности. *PostgreSQL будет использовать уровень изоляции чтения по умолчанию, если не указан другой уровень изоляции.*
3. Repeatable read - основан на гарантии, обеспечиваемой read commited. Как и раньше, он позволяет избежать грязного чтения, но также предотвращает и неповторяющееся чтение. Это означает, что никакие изменения, совершенные вне транзакции, никогда не повлияют на данные, считанные внутри транзакции. Запрос, выполняемый в начале транзакции, никогда не будет иметь другой результат в конце транзакции, если только это не вызвано непосредственно операторами внутри транзакции. Хотя стандартное определение уровня изоляции повторяемого чтения требует только предотвращения грязных и неповторяющихся операций чтения, PostgreSQL также предотвращает фантомное чтение на этом уровне. Это означает, что фиксации вне транзакции не могут изменить количество строк, удовлетворяющих запросу.
4. Serializable - Сериализуемый уровень изоляции обеспечивает высочайший уровень изоляции и согласованности. Это предотвращает все сценарии, которые происходят на уровне «repeatable read», а также устраняет возможность аномалий сериализации.

Теперь, когда мы рассмотрели различные уровни изоляции, которые PostgreSQL может использовать в транзакциях, давайте продемонстрируем, как определять транзакции.

В PostgreSQL каждый оператор за пределами явно помеченной транзакции фактически выполняется в отдельной транзакции, состоящей из одного оператора. Чтобы явно запустить блок транзакции, вы можете использовать команды BEGIN или START TRANSACTION (они являются синонимами). Чтобы зафиксировать транзакцию, введите команду COMMIT.

BEGIN;

statements

COMMIT;

В рамках транзакции либо все операторы, либо ни один из них не будут зафиксированы в базе данных. Отказ от операторов и изменений, внесенных в транзакции называется «откатом» транзакции.

Транзакции можно откатить автоматически или вручную. PostgreSQL автоматически откатывает транзакции, если один из операторов внутри приводит к ошибке. Он также откатывает транзакцию, если произойдет ошибка сериализации, если выбранный уровень изоляции не позволяет этого.

Чтобы вручную откатить инструкции, заданные во время текущей транзакции, вы можете использовать команду ROLLBACK.

BEGIN;

statements

ROLLBACK;

По умолчанию команда ROLLBACK сбрасывает транзакцию туда, где она была при первом вызове команд BEGIN или START TRANSACTION.

Хотя вы не можете указать произвольные места для отката при вводе команды ROLLBACK, вы можете выполнить откат к любым «точкам сохранения», которые вы установили во время транзакции. Вы можете заранее пометить места в своей транзакции с помощью команды SAVEPOINT, а затем ссылаться на эти конкретные места, когда вам нужно откатиться.

Эти точки сохранения позволяют создать промежуточную точку отката. Затем вы можете при желании отменить любые утверждения, сделанные между вашим текущим местоположением и точкой сохранения, а затем продолжить работу над своей транзакцией.

Чтобы указать точку сохранения, введите команду SAVEPOINT, за которой следует имя точки сохранения:

SAVEPOINT save\_1;

Для отката к точке сохранения используется команда ROLLBACK TO

ROLLBACK TO save\_1;

Чтобы установить уровень изоляции, который вы хотите для транзакции, вы можете добавить предложение ISOLATION LEVEL в команду START TRANSACTION или BEGIN. Основной синтаксис выглядит следующим образом:

BEGIN ISOLATION LEVEL <isolation\_level>;

statements

COMMIT;

<isolation\_level> - любой из 4 уровней изоляции, описанных выше.

Если у вас есть несколько транзакций, которые должны выполняться последовательно, вы можете связать их вместе с помощью команды COMMIT AND CHAIN.

Команда COMMIT AND CHAIN ​​завершает текущую транзакцию, фиксируя содержащиеся в ней действия. После обработки фиксации немедленно открывается новая транзакция. Это позволяет вам группировать в транзакции еще один набор операторов.

Этот оператор работает точно так же, как если бы вы выполнили COMMIT; BEGIN

**ЗАДАНИЕ**

**Исследуем блокировки в PostgreSQL**

Понимание того как работают блокировки, является ключом к написанию правильных запросов способных выполняться параллельно. Чтобы узнать как работают блокировки и увидеть, что происходит внутри базы данных, давайте рассмотрим наглядный пример.

Ex. Создадим «тестовую базу»:

create database sandbox;

Ex. Создадим «тестовую таблицу»:

create table toys (

id serial not null,

name character varying(36),

usage integer not null default 0,

constraint toys\_pkey primary key (id)

);

insert into toys(name) values('car'),('digger'),('shovel');



Откроем два терминала, в каждом из них подключимся к только что созданной базе данных sandbox. Чтобы не путаться, дадим им имена. Пусть это будут Алиса и Боб. Изменить подсказку командной строки можно с помощью команды

Первой появляется Алиса и осматривает игрушки:

begin;

select \* from toys;

id | name | usage

----+--------+-------

1 | car | 0

2 | digger | 0

3 | shovel | 0

(3 rows)

Обратите внимание, что оператор begin начинает транзакцию явно. В этом случае она будет продолжаться до тех пор, пока мы не зафиксируем её, сделаем commit, или не откатим, сделаем rollback.

Если бы Боб («второй терминал») сейчас посмотрел на игрушки, он увидел бы то же самое:

begin;

select \* from toys;

id | name | usage

----+--------+-------

1 | car | 0

2 | digger | 0

3 | shovel | 0

(3 rows)

Таким образом параллельное выполнение двух операторов select не мешает работе каждого из них. Именно такого поведения мы ожидаем от надёжной и высокопроизводительной базы данных.

**pg\_lock**

Однако, транзакции Алисы и Боба до сих пор открыты. Чтобы посмотреть какие блокировки были установлены, откроем третий терминал и назовём его Ева:

RENAME PANEL



select

lock.locktype,

lock.relation::regclass,

lock.mode,

lock.transactionid as tid,

lock.virtualtransaction as vtid,

lock.pid,

lock.granted

from pg\_catalog.pg\_locks lock

left join pg\_catalog.pg\_database db

on db.oid = lock.database

where (db.datname = 'sandbox' or db.datname is null)

and not lock.pid = pg\_backend\_pid()

order by lock.pid;

locktype | relation | mode | tid | vtid | pid | granted

------------+-----------+-----------------+-----+-------+-------+---------

relation | toys\_pkey | AccessShareLock | | 6/268 | 45265 | t

relation | toys | AccessShareLock | | 6/268 | 45265 | t

virtualxid | | ExclusiveLock | | 6/268 | 45265 | t

relation | toys\_pkey | AccessShareLock | | 1/282 | 45263 | t

relation | toys | AccessShareLock | | 1/282 | 45263 | t

virtualxid | | ExclusiveLock | | 1/282 | 45263 | t

(6 rows)

Представление pg\_lock показывает активные блокировки. Условие (db.datname = 'sandbox' or db.datname is null) оставляет только те записи которые относятся к «песочнице», а условие not pid = pg\_backend\_pid() исключает записи текущей сессии. Наконец, чтобы колонка relation стала более информативной было использовано приведение типа к regclass.

Посмотрим на пятую строку:

locktype | relation | mode | tid | vtid | pid | granted

------------+-----------+-----------------+-----+-------+-------+---------

relation | toys | AccessShareLock | | 1/282 | 45263 | t

Виртуальной транзакцией 1/282, на таблицу toys наложена блокировка AccessShareLock, при этом блокировка считается выданной (is granted). Пока всё идёт хорошо, Боб и Алиса счастливы, ведь они оба видят — игрушки можно взять.

Обратите внимание, каждая транзакция удерживает блокировку ExclusiveLock на своей виртуальной транзакции virtualxid.

Алиса решает взять машинку:

update toys set usage = usage + 1 where id = 1;

UPDATE 1

Никаких проблем. Посмотрим как выглядит таблица блокировок теперь:

locktype | relation | mode | tid | vtid | pid | granted

---------------+-----------+------------------+----------+-------+-------+---------

relation | toys\_pkey | AccessShareLock | | 6/268 | 45265 | t

relation | toys | AccessShareLock | | 6/268 | 45265 | t

virtualxid | | ExclusiveLock | | 6/268 | 45265 | t

relation | toys\_pkey | AccessShareLock | | 1/282 | 45263 | t

relation | toys\_pkey | RowExclusiveLock | | 1/282 | 45263 | t

relation | toys | AccessShareLock | | 1/282 | 45263 | t

relation | toys | RowExclusiveLock | | 1/282 | 45263 | t

virtualxid | | ExclusiveLock | | 1/282 | 45263 | t

transactionid | | ExclusiveLock | 24273800 | 1/282 | 45263 | t

(9 rows)

**transactionid**

В таблице toys на записи с машинкой теперь стоит блокировка RowExclusiveLock. Также появился реальный идентификатор транзакции transactionid на котором удерживается блокировка ExclusiveLock. Такой идентификатор появляется у каждой транзакции потенциально меняющей состояние базы данных.

**MVCC**

Поскольку транзакция Алисы не зафиксирована, Боб видит прежние данные:

select \* from toys;

id | name | usage

----+--------+-------

1 | car | 0

2 | digger | 0

3 | shovel | 0

(3 rows)

Мы не знаем, будет ли Алиса фиксировать или откатывать свою транзакцию. Следовательно, Боб видит содержимое таблицы неизменным.

Для того, чтобы каждый пользователь видел согласованное стостояние базы данных, постгрес использует механизм управления конкурентным доступом с помощью многоверсионности MVCC (Multi Version Concurrency Control).

**Блокирующие запросы**

Допустим Боб тоже хочет поиграть машинкой (типичная ситуация для детей). Боб выполняет следующий запрос:

update toys set usage = usage + 1 where id = 1;

но ничего не происходит. Ему нужно подождать пока Алиса завершит свою транзакцию. Снова посмотрим в таблицу блокировок:

locktype | relation | mode | tid | vtid | pid | granted

---------------+-----------+------------------+----------+-------+-------+---------

relation | toys\_pkey | AccessShareLock | | 6/268 | 45265 | t

relation | toys\_pkey | RowExclusiveLock | | 6/268 | 45265 | t

relation | toys | AccessShareLock | | 6/268 | 45265 | t

relation | toys | RowExclusiveLock | | 6/268 | 45265 | t

virtualxid | | ExclusiveLock | | 6/268 | 45265 | t

relation | toys\_pkey | AccessShareLock | | 1/282 | 45263 | t

relation | toys\_pkey | RowExclusiveLock | | 1/282 | 45263 | t

relation | toys | AccessShareLock | | 1/282 | 45263 | t

relation | toys | RowExclusiveLock | | 1/282 | 45263 | t

virtualxid | | ExclusiveLock | | 1/282 | 45263 | t

transactionid | | ExclusiveLock | 24273800 | 1/282 | 45263 | t

tuple | toys | ExclusiveLock | | 6/268 | 45265 | t

transactionid | | ExclusiveLock | 24273801 | 6/268 | 45265 | t

transactionid | | ShareLock | 24273800 | 6/268 | 45265 | f

(14 rows)

Теперь у Боба тоже есть transactionid и он просит выдать ему ShareLock на transactionid Алисы — «Мам, я тоже хочу поиграть машинкой». Поскольку две блокировки конфликтуют друг c другом, запрос Боба не удовлетворён (is not granted). Он будет висеть в таком состоянии до тех пор, пока Алиса не снимет ExclusiveLock, завершив свою транзакцию.

**pg\_stats\_activity**

[pg\_stat\_activity](http://www.postgresql.org/docs/9.4/static/monitoring-stats.html#PG-STAT-ACTIVITY-VIEW) ещё одно интересное представление (view) из pg\_catalog’а. Оно показывает запросы выполняющиеся в данный момент:

select query, state, pid

from pg\_stat\_activity

where datname = 'sandbox'

and not (state = 'idle' or pid = pg\_backend\_pid());

query | state | waiting | pid

-------------------------------------------------+---------------------+---------+-------

update toys set usage = usage + 1 where id = 1; | active | t | 45265

update toys set usage = usage + 1 where id = 1; | idle in transaction | f | 45263

(2 rows)

Мы видим, что запрос Алисы простаивает в ожидании поддтверждения транзакции (idle in transaction), в то время как запрос Боба активен и подвис (is waiting).

Чтобы увидеть кто кого заблокировал, объединим два запроса в один:

select

bda.pid as blocked\_pid,

bda.query as blocked\_query,

bga.pid as blocking\_pid,

bga.query as blocking\_query

from pg\_catalog.pg\_locks bdl

join pg\_stat\_activity bda

on bda.pid = bdl.pid

join pg\_catalog.pg\_locks bgl

on bgl.pid != bdl.pid

and bgl.transactionid = bdl.transactionid

join pg\_stat\_activity bga

on bga.pid = bgl.pid

where not bdl.granted

and bga.datname = 'sandbox';

blocked\_pid | blocked\_query | blocking\_pid | blocking\_query

-------------+-------------------------------------------------+--------------+-------------------------------------------------

45265 | update toys set usage = usage + 1 where id = 1; | 45263 | update toys set usage = usage + 1 where id = 1;

(1 row)

Если бы Алиса решила откатить или зафиксировать свою транзакцию, блокировка ExclusiveLock была бы снята и Боб получил бы ShareLock. После этого он мог бы зафиксировать свою транзакцию, и запись в таблице была бы обновлена независимо от решения Алисы.

rollback;

ROLLBACK

[Bob] sandbox> commit;

COMMIT

[Bob] sandbox> select \* from toys;

id | name | usage

----+--------+-------

2 | digger | 0

3 | shovel | 0

1 | car | 1

(3 rows)

Конечно, если бы Боб и Алиса решили играть разными игрушками, конфликтной ситуации между ними не возникло бы вообще.

**Явные блокировки**

Другая типичная ситуация для детей, когда один из них хочет забрать все игрушки без реальной необходимости:

[Alice] sandbox> begin;

BEGIN

[Alice] sandbox> lock table toys in access exclusive mode;

LOCK TABLE

Хотя Алиса и не взяла ни одной игрушки, Боб всё равно должен ждать.

[Bob] sandbox> begin; update toys set usage = usage + 1 where id = 2;

BEGIN

Таблица блокировок теперь выглядит так:

locktype | relation | mode | tid | vtid | pid | granted

------------+----------+---------------------+-----+-------+-------+---------

virtualxid | | ExclusiveLock | | 6/284 | 45265 | t

virtualxid | | ExclusiveLock | | 1/294 | 45263 | t

relation | toys | RowExclusiveLock | | 6/284 | 45265 | f

relation | toys | AccessExclusiveLock | | 1/294 | 45263 | t

(4 rows)

Поскольку Алиса удерживает AccessExclusiveLock без изменения состояния базы данных, то она не получила свой transactionid. У Боба его тоже нет, потому что он не получил RowExclusiveLock на таблицу toys. В этой ситуации, запрос для отображения блокировок который мы использовали ранее, нам не поможет, т. к. он использует объединение по transactionid.

blocked\_pid | blocked\_query | blocking\_pid | blocking\_query

-------------+---------------+--------------+----------------

(0 rows)

Таким образом, Ева думает, что всё хорошо, в то время как следующий запрос:

select pid, query, now() - query\_start as waiting\_duration

from pg\_catalog.pg\_stat\_activity

where datname = 'sandbox'

and waiting;

pid | query | waiting\_duration

-------+-------------------------------------------------+------------------

35929 | update toys set usage = usage + 1 where id = 2; | 00:01:34.519518

(1 row)

показывает обратное. Обратите внимание на столбец waiting\_duration, он рассчитывается как разница между now() и query\_start. Благодаря этому, видно сколько времени запрос уже висит.

Сделав объединение по столбцам relation и locktype, мы снова можем видеть кто кого блокирует:

select

bgl.relation::regclass,

bda.pid as blocked\_pid,

bda.query as blocked\_query,

bdl.mode as blocked\_mode,

bga.pid AS blocking\_pid,

bga.query as blocking\_query,

bgl.mode as blocking\_mode

from pg\_catalog.pg\_locks bdl

join pg\_stat\_activity bda

on bda.pid = bdl.pid

join pg\_catalog.pg\_locks bgl

on bdl.pid != bgl.pid

and bgl.relation = bdl.relation

and bgl.locktype = bdl.locktype

join pg\_stat\_activity bga

on bga.pid = bgl.pid

where not bdl.granted

and bga.datname = 'sandbox';

relation | blocked\_pid | blocked\_query | blocked\_mode | blocking\_pid | blocking\_query | blocking\_mode

----------+-------------+-------------------------------------------------+------------------+--------------+-------------------------------------------+---------------------

toys | 35929 | update toys set usage = usage + 1 where id = 2; | RowExclusiveLock | 35937 | lock table toys in access exclusive mode; | AccessExclusiveLock

(1 row)

Алисе было сказано, что некрасиво делать явную блокировку без видимой на то причины. Она фиксирует свою транзакцию без каких-либо изменений, а Боб может взять игрушку.

[Alice] sandbox> commit;

COMMIT

UPDATE 1

[Bob] sandbox>

Но его транзакция всё ещё открыта. Если мы посмотрим в таблицу блокировок, то увидим следующее:

locktype | relation | mode | tid | vtid | pid | granted

---------------+-----------+------------------+-------+------+-------+---------

relation | toys\_pkey | RowExclusiveLock | | 4/51 | 35929 | t

virtualxid | | ExclusiveLock | | 4/51 | 35929 | t

relation | toys | RowExclusiveLock | | 4/51 | 35929 | t

transactionid | | ExclusiveLock | 19307 | 4/51 | 35929 | t

(4 rows)

Лишь после того как Боб получил RowExclusiveLock, к его транзакции был добавлен transactionid. Боб рад и делает коммит:

[Bob] sandbox> commit;

COMMIT

**RowExclusiveLock**

Поскольку Алиса не знает какую игрушку она хочет взять, а ставить явную блокировку ей не разрешили, она пробует другой подход:

[Alice] sandbox> begin; select \* from toys for update;

BEGIN

id | name | usage

----+--------+-------

2 | digger | 1

3 | shovel | 0

1 | car | 1

(3 rows)

На детском языке это бы звучало примерно так: «Хочу видеть все игрушки и может быть я возьму одну, но пока не знаю какую. А до тех пор я не хочу чтобы кто-то другой прикасался к ним».

Тем временем Боб хочет взять лопатку, но конечно не может этого сделать, его транзакция подвисает:

[Bob] sandbox> begin; update toys set usage = usage + 1 where id = 3;

BEGIN

Ева видит следующую ситуацию:

locktype | relation | mode | tid | vtid | pid | granted

---------------+-----------+------------------+-------+------+-------+---------

transactionid | | ShareLock | 19309 | 4/55 | 35929 | f

relation | toys | RowExclusiveLock | | 4/55 | 35929 | t

virtualxid | | ExclusiveLock | | 4/55 | 35929 | t

transactionid | | ExclusiveLock | 19310 | 4/55 | 35929 | t

tuple | toys | ExclusiveLock | | 4/55 | 35929 | t

relation | toys\_pkey | RowExclusiveLock | | 4/55 | 35929 | t

relation | toys | RowShareLock | | 5/17 | 35937 | t

virtualxid | | ExclusiveLock | | 5/17 | 35937 | t

relation | toys\_pkey | AccessShareLock | | 5/17 | 35937 | t

transactionid | | ExclusiveLock | 19309 | 5/17 | 35937 | t

(10 rows)

Боб совершенно ясно хочет изменить состояние базы данных поэтому он получил transactionid равный 19310, но снова вынужден ждать получения ShareLock на транзакцию Алисы с номером 19309.

**Объединяем блокировки и активности**

Пришло время объединить таблицу блокировок и таблицу активности вместе, так, чтобы всегда видеть кто кого заблокировал:

select

coalesce(bgl.relation::regclass::text, bgl.locktype) as locked\_item,

now() - bda.query\_start as waiting\_duration,

bda.pid as blocked\_pid,

bda.query as blocked\_query,

bdl.mode as blocked\_mode,

bga.pid as blocking\_pid,

bga.query as blocking\_query,

bgl.mode as blocking\_mode

from pg\_catalog.pg\_locks bdl

join pg\_stat\_activity bda

on bda.pid = bdl.pid

join pg\_catalog.pg\_locks bgl

on bgl.pid != bdl.pid

and (bgl.transactionid = bdl.transactionid

or bgl.relation = bdl.relation and bgl.locktype = bdl.locktype)

join pg\_stat\_activity bga

on bga.pid = bgl.pid

and bga.datid = bda.datid

where not bdl.granted

and bga.datname = current\_database();

locked\_item | waiting\_duration | blocked\_pid | blocked\_query | blocked\_mode | blocking\_pid | blocking\_query | blocking\_mode

---------------+------------------+-------------+-------------------------------------------------+--------------+--------------+--------------------------------+---------------

transactionid | 00:03:32.330397 | 35929 | update toys set usage = usage + 1 where id = 3; | ShareLock | 35937 | select \* from toys for update; | ExclusiveLock

(1 row)

Для оценки времени блокирования запроса был добавлен столбец waiting\_duration, а также функция current\_database() используемая в условии.

Ева не может запомнить этот длиннющий запрос и создаёт представление:

create view lock\_monitor as (

select

coalesce(bgl.relation::regclass::text, bgl.locktype) as locked\_item,

now() - bda.query\_start as waiting\_duration,

bda.pid as blocked\_pid,

bda.query as blocked\_query,

bdl.mode as blocked\_mode,

bga.pid as blocking\_pid,

bga.query as blocking\_query,

bgl.mode as blocking\_mode

from pg\_catalog.pg\_locks bdl

join pg\_stat\_activity bda

on bda.pid = bdl.pid

join pg\_catalog.pg\_locks bgl

on bgl.pid != bdl.pid

and (bgl.transactionid = bdl.transactionid

or bgl.relation = bdl.relation and bgl.locktype = bdl.locktype)

join pg\_stat\_activity bga

on bga.pid = bgl.pid

and bga.datid = bda.datid

where not bdl.granted

and bga.datname = current\_database()

);

С помощью него, она легко узнает что задумали её дети:

[eve] sandbox> select \* from lock\_monitor;

locked\_item | waiting\_duration | blocked\_pid | blocked\_query | blocked\_mode | blocking\_pid | blocking\_query | blocking\_mode

---------------+------------------+-------------+-------------------------------------------------+--------------+--------------+--------------------------------+---------------

transactionid | 00:06:19.986426 | 35929 | update toys set usage = usage + 1 where id = 3; | ShareLock | 35937 | select \* from toys for update; | ExclusiveLock

(1 row)

Выпив чашку чая и успокоившись, Ева решает почитать [руководство по явным блокировкам в постгресе](http://www.postgresql.org/docs/9.1/static/explicit-locking.html), узнать какие бывают виды блокировок и то, как они конфликтуют друг с другом.

**ОЗНАКОМИТЬСЯ**

MVCC-1. Изоляция

[Блог компании Postgres Professional](https://habr.com/ru/companies/postgrespro/articles/)[PostgreSQL\*](https://habr.com/ru/hubs/postgresql/)[SQL\*](https://habr.com/ru/hubs/sql/)

Привет, Хабр! Этой статьей я начинаю серию циклов (или цикл серий? в общем, задумка грандиозная) о внутреннем устройстве PostgreSQL.  
  
Материал будет основан на [учебных курсах](https://postgrespro.ru/education/courses) по администрированию, которые делаем мы с Павлом [pluzanov](https://habr.com/ru/users/pluzanov/). Смотреть видео не все любят (я точно не люблю), а читать слайды, пусть даже с комментариями, — совсем «не то».  
  
Конечно, статьи не будут повторять содержание курсов один в один. Я буду говорить только о том, как все устроено, опуская собственно администрирование, зато постараюсь делать это более подробно и обстоятельно. И я верю в то, что такие знания полезны прикладному разработчику не меньше, чем администратору.  
  
Ориентироваться я буду на тех, кто уже имеет определенный опыт использования PostgreSQL и хотя бы в общих чертах представляет себе, что к чему. Для совсем новичков текст будет тяжеловат. Например, я ни слова не скажу о том, как установить PostgreSQL и запустить psql.  
  
Вещи, о которых пойдет речь, не сильно меняются от версии к версии, но использовать я буду текущий, 11-й «ванильный» PostgreSQL.  
  
Первый цикл посвящен вопросам, связанным с изоляцией и многоверсионностью, и план его таков:

1. Изоляция, как ее понимают стандарт и PostgreSQL (эта статья);
2. [Слои, файлы, страницы](https://habr.com/ru/company/postgrespro/blog/444536/) — что творится на физическом уровне;
3. [Версии строк, виртуальные и вложенные транзакции](https://habr.com/ru/company/postgrespro/blog/445820/);
4. [Снимки данных и видимость версий строк, горизонт событий](https://habr.com/ru/company/postgrespro/blog/446652/);
5. [Внутристраничная очистка и HOT-обновления](https://habr.com/ru/company/postgrespro/blog/449704/);
6. [Обычная очистка](https://habr.com/ru/company/postgrespro/blog/452320/) (vacuum);
7. [Автоматическая очистка](https://habr.com/ru/company/postgrespro/blog/452762/) (autovacuum);
8. [Переполнение счетчика транзакций и заморозка](https://habr.com/ru/company/postgrespro/blog/455590/).

Ну, поехали.

Читайте и другие серии.  
  
Индексы:

1. [Механизм индексирования](https://habr.com/ru/company/postgrespro/blog/326096/);
2. [Интерфейс метода доступа](https://habrahabr.ru/company/postgrespro/blog/326106/), классы и семейства операторов;
3. [Hash](https://habrahabr.ru/company/postgrespro/blog/328280/);
4. [B-tree](https://habrahabr.ru/company/postgrespro/blog/330544/);
5. [GiST](https://habrahabr.ru/company/postgrespro/blog/333878/);
6. [SP-GiST](https://habrahabr.ru/company/postgrespro/blog/337502/);
7. [GIN](https://habrahabr.ru/company/postgrespro/blog/340978/);
8. [RUM](https://habrahabr.ru/company/postgrespro/blog/343488/);
9. [BRIN](https://habrahabr.ru/company/postgrespro/blog/346460/);
10. [Bloom](https://habrahabr.ru/company/postgrespro/blog/349224/).

Журналирование:

1. [Буферный кеш](https://habr.com/ru/company/postgrespro/blog/458186/);
2. [Журнал предзаписи](https://habr.com/ru/company/postgrespro/blog/459250/) — как устроен и как используется при восстановлении;
3. [Контрольная точка](https://habr.com/ru/company/postgrespro/blog/460423/) и фоновая запись — зачем нужны и как настраиваются;
4. [Настройка журнала](https://habr.com/ru/company/postgrespro/blog/461523/) — уровни и решаемые задачи, надежность и производительность.

Блокировки:

1. [Блокировки отношений](https://habr.com/ru/company/postgrespro/blog/462877/);
2. [Блокировки строк](https://habr.com/ru/company/postgrespro/blog/463819/);
3. [Блокировки других объектов](https://habr.com/ru/company/postgrespro/blog/465263/) и предикатные блокировки;
4. [Блокировки в оперативной памяти](https://habr.com/ru/company/postgrespro/blog/466199/).

Что такое изоляция и почему это важно

Наверное, все как минимум знают про существование транзакций, встречали аббревиатуру ACID и слышали про уровни изоляции. Но приходится еще встречать мнение, что это-де теория, на практике не нужная. Поэтому я потрачу некоторое время на попытку объяснить, почему это действительно важно.  
  
Вряд ли вы обрадуетесь, если приложение получит из базы некорректные данные, или если приложение запишет некорректные данные в базу.  
  
Но что такое “корректные” данные? Известно, что на уровне базы данных можно создавать *ограничения целостности* (integrity constraints, такие как NOT NULL или UNIQUE). Если данные всегда удовлетворяют ограничениям целостности (а это так, потому что СУБД это гарантирует), то они целостны.  
  
*Корректны* и *целостны* — одно и то же? Не совсем. Не все ограничения можно сформулировать на уровне базы данных. Часть ограничений слишком сложна, например, охватывает сразу несколько таблиц. И даже если ограничение в принципе можно было бы определить в базе данных, но из каких-то соображений это не сделали, это ведь не означает, что его можно нарушать.  
  
Итак, *корректность* строже, чем *целостность*, но что это такое, мы точно не знаем. Остается признать, что эталоном корректности является приложение, которое, как нам хочется верить, написано *корректно* и никогда не ошибается. Во всяком случае, если приложение не нарушит целостность, но нарушит корректность, СУБД не узнает об этом и не поймает его за руку.  
  
Дальше мы будет называть корректность термином *согласованность* (consistency).  
  
Давайте, однако, предположим, что приложение выполняет только корректные последовательности операторов. В чем тогда роль СУБД, если приложение и так корректно?  
  
Во-первых, оказывается, что корректная последовательность операторов может временно нарушать согласованность данных, и это — как ни странно — нормально. Заезженный, но понятный пример состоит в переводе средств с одного счета на другой. Правило согласованности может звучать так: *перевод никогда не меняет общей суммы денег на счетах* (такое правило довольно трудно записать на SQL в виде ограничения целостности, так что оно существует на уровне приложения и невидимо для СУБД). Перевод состоит из двух операций: первая уменьшает средства на одном счете, вторая — увеличивает на другом. Первая операция нарушает согласованность данных, вторая — восстанавливает.

Хорошее упражнение — реализовать описанное выше правило на уровне ограничений целостности. А вам слабо? ©

Что, если первая операция выполнится, а вторая — нет? Ведь запросто: во время второй операции может пропасть электричество, упасть сервер, произойти деление на ноль — да мало ли. Понятно, что согласованность нарушится, а этого допускать нельзя. Разрешать такие ситуации на уровне приложения в принципе можно ценой неимоверных усилий, но, к счастью, не нужно: это берет на себя СУБД. Но для этого она должна знать, что две операции составляют неделимое целое. То есть *транзакцию*.  
  
Получается интересное: зная, что операции составляют транзакцию, СУБД помогает поддерживать согласованность, гарантируя атомарность транзакций, при этом ничего не зная о конкретных правилах согласованности.  
  
Но есть и второй, более тонкий момент. Как только в системе появляется несколько одновременно работающих транзакций, абсолютно корректных поодиночке, совместно они могут работать некорректно. Это происходит из-за того, что перемешивается порядок выполнения операций: нельзя считать, что сначала выполняются все операции одной транзакции, а только потом — все операции другой.  
  
Замечание по поводу одновременности. Действительно одновременно транзакции могут работать на системе с многоядерным процессором, с дисковым массивом и т. д. Но все те же рассуждения справедливы и для сервера, который выполняет команды последовательно, в режиме разделения времени: столько-то тактов выполняется одна транзакция, столько тактов другая. Иногда для обобщения используют термин *конкурентное* выполнение.  
  
Ситуации, когда корректные транзакции некорректно работают вместе, называются *аномалиями* одновременного выполнения.  
  
Простой пример: если приложение хочет получить из базы корректные данные, то оно, как минимум, не должно видеть изменения других незафиксированных транзакций. Иначе можно не просто получить несогласованные данные, но и увидеть что-то такое, чего в базе данных никогда не было (если транзакция будет отменена). Такая аномалия называется *грязным чтением*.  
  
Если и другие, более сложные аномалии, с которыми мы разберемся чуть позже.  
  
Отказываться от одновременного выполнения, конечно, нельзя: иначе о какой производительности может идти речь? Но нельзя и работать с некорректными данными.  
  
И снова на помощь приходит СУБД. Можно сделать так, чтобы транзакции выполнялись *как будто* последовательно, *как будто* одна за другой. Иными словами — *изолированно* друг от друга. В реальности СУБД может выполнять операции вперемешку, но гарантировать при этом, что результат одновременного выполнения будет совпадать с результатом какого-нибудь из возможных последовательных выполнений. А это устраняет любые возможные аномалии.  
  
Итак, мы подошли к определению:

Транзакцией называется множество операций, выполняемое приложением, которое переводит базу данных из одного корректного состояния в другое корректное состояние (согласованность) при условии, что транзакция выполнена полностью (атомарность) и без помех со стороны других транзакций (изоляция).

Это определение объединяет три первые буквы аббревиатуры ACID. Они настолько тесно связаны друг с другом, что рассматривать одно без другого просто нет смысла. На самом деле сложно оторвать и букву D (durability). Ведь при крахе системы в ней остаются изменения незафиксированных транзакций, с которыми приходится что-то делать, чтобы восстановить согласованность данных.  
  
Все было бы хорошо, но реализация полной изоляции — технически сложная задача, сопряженная с уменьшением пропускной способности системы. Поэтому на практике очень часто (не всегда, но почти всегда) применяется ослабленная изоляция, которая предотвращает некоторые, но не все аномалии. А это означает, что часть работы по обеспечению корректности данных ложится на приложение. Именно поэтому очень важно понимать, какой уровень изоляции используется в системе, какие гарантии он дает, а какие — не дает, и как в таких условиях писать корректный код.

Уровни изоляции и аномалии в стандарте SQL

Стандарт SQL с давних пор описывает четыре уровня изоляции. Эти уровни определяются перечислением аномалий, которые допускаются или не допускаются при одновременном выполнении транзакций на этом уровне. Поэтому чтобы говорить об этих уровнях, надо познакомиться с аномалиями.  
  
Подчеркну, что в этой части речь идет о стандарте, то есть о некой теории, на которую практика существенно опирается, но с которой в то же время изрядно расходится. Поэтому и все примеры здесь умозрительные. Они будут использовать те же операции над счетами клиентов: это довольно наглядно, хотя, надо признать, не имеет ни малейшего отношения к тому, как банковские операции устроены в действительности.

Потерянное обновление

Начнем с *потерянного обновления*. Такая аномалия возникает, когда две транзакции читают одну и ту же строку таблицы, затем одна транзакция обновляет эту строку, а после этого вторая транзакция тоже обновляет ту же строку, не учитывая изменений, сделанных первой транзакцией.  
  
Например, две транзакции собираются увеличить сумму на одном и том же счете на 100 ₽. Первая транзакция читает текущее значение (1000 ₽), затем вторая транзакция читает то же значение. Первая транзакция увеличивает сумму (получается 1100 ₽) и записывает это значение. Вторая транзакция поступает так же — получает те же 1100 ₽ и записывает их. В результате клиент потерял 100 ₽.  
  
Потерянное обновление не допускается стандартом ни на одном уровне изоляции.

Грязное чтение и Read Uncommitted

С *грязным чтением* мы уже познакомились выше. Такая аномалия возникает, когда транзакция читает еще не зафиксированные изменения, сделанные другой транзакцией.  
  
Например, первая транзакция переводит все деньги со счета клиента на другой счет, но не фиксирует изменение. Другая транзакция читает состояние счета, получает 0 ₽ и отказывает клиенту в выдаче наличных — несмотря на то, что первая транзакция прерывается и отменяет свои изменения, так что значения 0 никогда не существовало в базе данных.  
  
Грязное чтение допускается стандартом на уровне Read Uncommitted.

Неповторяющееся чтение и Read Committed

Аномалия *неповторяющегося чтения* возникает, когда транзакция читает одну и ту же строку два раза, и в промежутке между чтениями вторая транзакция изменяет (или удаляет) эту строку и фиксирует изменения. Тогда первая транзакция получит разные результаты.  
  
Например, пусть правило согласованности *запрещает отрицательные суммы на счетах клиентов*. Первая транзакция собирается уменьшить сумму на счете на 100 ₽. Она проверяет текущее значение, получает 1000 ₽ и решает, что уменьшение возможно. В это время вторая транзакция уменьшает сумму на счете до нуля и фиксирует изменения. Если бы теперь первая транзакция повторно проверила сумму, она получила бы 0 ₽ (но она уже приняла решение уменьшить значение, и счет “уходит в минус”).  
  
Неповторяющееся чтение допускается стандартом на уровнях Read Uncommitted и Read Committed. А вот грязное чтение Read Committed не допускает.

Фантомное чтение и Repeatable Read

*Фантомное чтение* возникает, когда транзакция два раза читает набор строк по одному и тому же условию, и в промежутке между чтениями вторая транзакция добавляет строки, удовлетворяющие этому условию (и фиксирует изменения). Тогда первая транзакция получит разные наборы строк.  
  
Например, пусть правило согласованности *запрещает клиенту иметь более 3 счетов*. Первая транзакция собирается открыть новый счет, проверяет их текущее количество (скажем, 2) и решает, что открытие возможно. В это время вторая транзакция тоже открывает клиенту новый счет и фиксирует изменения. Если бы теперь первая транзакция перепроверила количество, она получила бы 3 (но она уже выполняет открытие еще одного счета и у клиента их оказывается 4).  
  
Фантомное чтение допускается стандартом на уровнях Read Uncommitted, Read Committed и Repeatable Read. Но на уровне Repeatable Read не допускается неповторяющееся чтение.

Отсутствие аномалий и Serializable

Стандарт определяет и еще один уровень — Serializable, — на котором не допускаются никакие аномалии. И это совсем не то же самое, что запрет на потерянное обновление и на грязное, неповторяющееся и фантомное чтения.  
  
Дело в том, что существует значительно больше известных аномалий, чем перечислено в стандарте, и еще неизвестное число пока неизвестных.  
  
Уровень Serializable должен предотвращать *вообще все* аномалии. Это означает, что на таком уровне разработчику приложения не надо думать об одновременном выполнении. Если транзакции выполняют корректные последовательности операторов, работая в одиночку, данные будут согласованы и при одновременной работе этих транзакций.

Итоговая табличка

Вот теперь можно и привести всем известную таблицу. Но здесь для ясности к ней добавлен последний столбец, которого нет в стандарте.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **потерянные изменения** | **грязное чтение** | **неповторяющееся чтение** | **фантомное чтение** | **другие аномалии** |
| **Read Uncommitted** | **—** | **да** | **да** | **да** | **да** |
| **Read Committed** | **—** | **—** | **да** | **да** | **да** |
| **Repeatable Read** | **—** | **—** | **—** | **да** | **да** |
| **Serializable** | **—** | **—** | **—** | **—** | **—** |

Почему именно эти аномалии?

Почему из множества возможных аномалий в стандарте перечислены только несколько и почему именно такие?  
  
Достоверно этого, видимо, никто не знает. Но практика здесь точно обогнала теорию, так что не исключено, что о других аномалиях тогда (речь о стандарте SQL:92) просто не задумывались.  
  
Кроме того, предполагалось, что изоляция должна быть построена на блокировках. Идея широко применявшегося *протокола двухфазного блокирования* (2PL) состоит в том, что в процессе выполнения транзакция блокирует строки, с которыми работает, а при завершении — освобождает блокировки. Сильно упрощая, чем больше блокировок захватывает транзакция, тем лучше она изолирована от других транзакций. Но и тем сильнее страдает производительность системы, поскольку вместо совместной работы транзакции начинают выстраиваться в очередь за одними и теми же строками.  
  
Как мне представляется, разница между уровнями изоляции стандарта объясняется как раз количеством необходимых блокировок.  
  
Если транзакция блокирует изменяемые строки от изменения, но не от чтения, получаем уровень Read Uncommitted: потерянные изменения не допускаются, но можно прочитать незафиксированные данные.  
  
Если транзакция блокирует изменяемые строки и от чтения, и от изменения, получаем уровень Read Committed: незафиксированные данные прочитать нельзя, но при повторном обращении к строке можно получить другое значение (неповторяющееся чтение).  
  
Если транзакция блокирует и читаемые, и изменяемые строки и от чтения, и от изменения, получаем уровень Repeatable Read: повторное чтение строки будет выдавать то же значение.  
  
Но с Serializable проблема: невозможно заблокировать строку, которой еще нет. Из-за этого остается возможность фантомного чтения: другая транзакция может добавить (но не удалить) строку, попадающую под условия выполненного ранее запроса, и эта строка попадет в повторную выборку.  
  
Поэтому для реализации уровня Serializable обычных блокировок не хватает — нужно блокировать не строки, а условия (предикаты). Такие блокировки и были названы *предикатными*. Предложены они были еще в 1976 году, но их практическая применимость ограничена достаточно простыми условиями, для которых понятно, как объединять два разных предиката. До реализации таких блокировок в какой-либо системе дело, насколько мне известно, не дошло.

Уровни изоляции в PostgreSQL

Со временем на смену блокировочным протоколам управления транзакциями пришел *протокол изоляции на основе снимков* (Snapshot Isolation). Его идея состоит в том, что каждая транзакция работает с согласованным снимком данных на определенный момент времени, в который попадают только те изменения, которые были зафиксированы до момента создания снимка.  
  
Такая изоляция автоматически не допускает грязное чтение. Формально в PostgreSQL можно указать уровень Read Uncommitted, но работать она будет точно так же, как Read Committed. Поэтому про уровень Read Uncommitted мы дальше вообще не будем говорить.  
  
В PostgreSQL реализован *многоверсионный* вариант такого протокола. Идея многоверсионности состоит в том, что в СУБД могут сосуществовать несколько версий одной и той же строки. Это позволяет строить снимок данных, используя имеющиеся версии, и обходиться минимумом блокировок. Фактически блокируется только повторное изменение одной и той же строки. Все остальные операции выполняются одновременно: пишущие транзакции никогда не блокируют читающих транзакций, а читающие никогда не блокируют никого.  
  
За счет использования снимков данных изоляция в PostgreSQL получается строже, чем того требует стандарт: уровень Repeatable Read не допускает не только неповторяющегося, но и фантомного чтения (хотя и не обеспечивает полную изоляцию). И достигается это без потери эффективности.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **потерянные изменения** | **грязное чтение** | **неповторяющееся чтение** | **фантомное чтение** | **другие аномалии** |
| **Read Uncommitted** | **—** | **—** | **да** | **да** | **да** |
| **Read Committed** | **—** | **—** | **да** | **да** | **да** |
| **Repeatable Read** | **—** | **—** | **—** | **—** | **да** |
| **Serializable** | **—** | **—** | **—** | **—** | **—** |

Как многоверсионность реализована “под капотом”, мы поговорим в следующих статьях, а сейчас подробно посмотрим на каждый из трех уровней глазами пользователя (как вы понимаете, самое интересное скрывается за «другими аномалиями»). Для этого создадим таблицу счетов. У Алисы и Боба по 1000 ₽, но у Боба открыто два счета:

=> **CREATE** **TABLE** accounts(

id integer **PRIMARY KEY** **GENERATED** **BY** **DEFAULT** **AS** **IDENTITY**,

number text **UNIQUE**,

client text,

amount numeric

);

=> **INSERT** **INTO** accounts **VALUES**

(1, '1001', 'alice', 1000.00),

(2, '2001', 'bob', 100.00),

(3, '2002', 'bob', 900.00);

Read Committed

Отсутствие грязного чтения

Легко убедиться в том, что грязные данные прочитать невозможно. Начнем транзакцию. По умолчанию она будет использовать уровень изоляции Read Committed:

=> **BEGIN**;

=> **SHOW** transaction\_isolation;

transaction\_isolation

-----------------------

read committed

(1 row)

Говоря точнее, умолчательный уровень задается параметром, его при необходимости можно изменить:

=> **SHOW** default\_transaction\_isolation;

default\_transaction\_isolation

-------------------------------

read committed

(1 row)

Итак, в открытой транзакции снимаем средства со счета, но не фиксируем изменения. Свои собственные изменения транзакция видит:

=> **UPDATE** accounts **SET** amount = amount - 200 **WHERE** id = 1;

=> **SELECT** \* **FROM** accounts **WHERE** client = 'alice';

id | number | client | amount

----+--------+--------+--------

1 | 1001 | alice | 800.00

(1 row)

Во втором сеансе начнем еще одну транзакцию с тем же уровнем Read Committed. Чтобы отличать разные транзакции, команды второй транзакции будут показаны с отступом и отчеркнуты.  
  
Для того, чтобы повторить приведенные команды (а это полезно), надо открыть два терминала и в каждом запустить psql. В первом можно вводить команды одной транзакции, а во втором — команды другой.

| => **BEGIN**;

| => **SELECT** \* **FROM** accounts **WHERE** client = 'alice';

| id | number | client | amount

| ----+--------+--------+---------

| 1 | 1001 | alice | 1000.00

| (1 row)

Как и ожидалось, другая транзакция не видит незафиксированные изменения — грязное чтение не допускается.

Неповторяющееся чтение

Пусть теперь первая транзакция зафиксирует изменения, а вторая повторно выполнит тот же самый запрос.

=> **COMMIT**;

| => **SELECT** \* **FROM** accounts **WHERE** client = 'alice';

| id | number | client | amount

| ----+--------+--------+--------

| 1 | 1001 | alice | 800.00

| (1 row)

| => **COMMIT**;

Запрос получает уже новые данные — это и есть аномалия *неповторяющегося чтения*, которая допускается на уровне Read Committed.  
  
*Практический вывод*: в транзакции нельзя принимать решения на основании данных, прочитанных предыдущим оператором — потому что за время между выполнением операторов все может измениться. Вот пример, вариации которого встречается в прикладном коде так часто, что он является классическим антипаттерном:

**IF** (**SELECT** amount **FROM** accounts **WHERE** id = 1) >= 1000 **THEN**

**UPDATE** accounts **SET** amount = amount - 1000 **WHERE** id = 1;

**END** **IF**;

За время, которое проходит между проверкой и обновлением, другие транзакции могут как угодно изменить состояние счета, так что такая «проверка» ни от чего не спасает. Удобно представлять себе, что между операторами одной транзакции могут «вклиниться» любые другие операторы других транзакций, например вот так:

**IF** (**SELECT** amount **FROM** accounts **WHERE** id = 1) >= 1000 **THEN**

-----

| **UPDATE** accounts **SET** amount = amount - 200 **WHERE** id = 1;

| **COMMIT**;

-----

**UPDATE** accounts **SET** amount = amount - 1000 **WHERE** id = 1;

**END** **IF**;

Если, переставив операторы, можно все испортить, значит код написан некорректно. И не стоит обманывать себя, что такого стечения обстоятельств не произойдет — произойдет обязательно.  
  
А как написать код корректно? Возможности, как правило, сводятся к следующему:

* Не писать код.  
  Это не шутка. Например, в данном случае проверка легко превращается в ограничение целостности:  
  ALTER TABLE accounts ADD CHECK amount >= 0;  
  Теперь никакие проверки не нужны: достаточно просто выполнить действие и при необходимости обработать исключение, которое возникнет в случае попытки нарушения целостности.
* Использовать один SQL-оператор.  
  Проблемы с согласованностью возникают из-за того, что в промежутке между операторами может завершиться другая транзакция и изменятся видимые данные. А если оператор один, то и промежутков никаких нет.  
  В PostgreSQL достаточно средств, чтобы одним SQL-оператором решать сложные задачи. Отметим общие табличные выражения (CTE), в которых в том числе можно использовать операторы INSERT/UPDATE/DELETE, а также оператор INSERT ON CONFLICT, который реализует логику «вставить, а если строка уже есть, то обновить» в одном операторе.
* Пользовательские блокировки.  
  Последнее средство — вручную установить исключительную блокировку или на все нужные строки (SELECT FOR UPDATE), или вообще на всю таблицу (LOCK TABLE). Это всегда работает, но сводит на нет преимущества многоверсионности: вместо одновременного выполнения часть операций будет выполняться последовательно.

Несогласованное чтение

Прежде чем приступать к следующему уровню изоляции, придется признать, что не все так просто. Реализация PostgreSQL такова, что допускает другие, менее известные аномалии, которые не регламентируются стандартом.  
  
Допустим, первая транзакция начала перевод средств с одного счета Боба на другой:

=> **BEGIN**;

=> **UPDATE** accounts **SET** amount = amount - 100 **WHERE** id = 2;

В это время другая транзакция подсчитывает баланс Боба, причем подсчет выполняется в цикле по всем счетам Боба. Фактически транзакция начинает с первого счета (и, очевидно, видит прежнее состояние):

| => **BEGIN**;

| => **SELECT** amount **FROM** accounts **WHERE** id = 2;

| amount

| --------

| 100.00

| (1 row)

В этот момент первая транзакция успешно завершается:

=> **UPDATE** accounts **SET** amount = amount + 100 **WHERE** id = 3;

=> **COMMIT**;

А другая читает состояние второго счета (и видит уже новое значение):

| => **SELECT** amount **FROM** accounts **WHERE** id = 3;

| amount

| ---------

| 1000.00

| (1 row)

| => **COMMIT**;

Таким образом вторая транзакция получила в сумме 1100 ₽, то есть некорректные данные. Это и есть аномалия *несогласованного чтения*.  
  
Как избежать такой аномалии, оставаясь на уровне Read Committed? Конечно, использовать один оператор. Например так:

**SELECT** sum(amount) **FROM** accounts **WHERE** client = 'bob';

До сих пор я утверждал, что видимость данных может поменяться только между операторами, но так ли это очевидно? А если запрос выполняется долго, может ли он увидеть часть данных в одном состоянии, а часть — в другом?  
  
Проверим. Удобный способ для этого — вставить в оператор искусственную задержку, вызвав функцию pg\_sleep. Ее параметр задает время задержки в секундах.

=> **SELECT** amount, pg\_sleep(2) **FROM** accounts **WHERE** client = 'bob';

Пока эта конструкция выполняется, в другой транзакции выполняем перенос средств обратно:

| => **BEGIN**;

| => **UPDATE** accounts **SET** amount = amount + 100 **WHERE** id = 2;

| => **UPDATE** accounts **SET** amount = amount - 100 **WHERE** id = 3;

| => **COMMIT**;

Результат показывает, что оператор видит данные в таком состоянии, в котором они находились на момент начала его выполнения. Это, безусловно, правильно.

amount | pg\_sleep

---------+----------

0.00 |

1000.00 |

(2 rows)

Но и тут не все так просто. PostgreSQL позволяет определять функции, а у функций есть понятие *категории изменчивости*. Если в запросе вызывается *изменчивая* функция (с категорией VOLATILE), и в этой функции выполняется другой запрос, то этот запрос внутри функции будет видеть данные, не согласованные с данными основного запроса.

=> **CREATE** **FUNCTION** get\_amount(id integer) **RETURNS** numeric **AS** $$

**SELECT** amount **FROM** accounts a **WHERE** a.id = get\_amount.id;

$$ **VOLATILE** **LANGUAGE** **sql**;

=> **SELECT** get\_amount(id), pg\_sleep(2)

**FROM** accounts **WHERE** client = 'bob';

| => **BEGIN**;

| => **UPDATE** accounts **SET** amount = amount + 100 **WHERE** id = 2;

| => **UPDATE** accounts **SET** amount = amount - 100 **WHERE** id = 3;

| => **COMMIT**;

В этом случае получим некорректные данные — 100 ₽ потерялись:

get\_amount | pg\_sleep

------------+----------

100.00 |

800.00 |

(2 rows)

Подчеркну, что такой эффект возможен только на уровне изоляции Read Committed, и только с категорией изменчивость VOLATILE. Беда в том, что умолчанию используется именно этот уровень изоляции и именно эта категория изменчивости, так что надо признать — грабли лежат очень удачно. Не наступайте!

Несогласованное чтение в обмен на потерянные изменения

Несогласованное чтение в рамках одного оператора можно — несколько неожиданным образом — получить и при обновлении.  
  
Посмотрим, что происходит при попытке изменения одной и той же строки двумя транзакциями. Сейчас у Боба 1000 ₽ на двух счетах:

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob';

id | number | client | amount

----+--------+--------+--------

2 | 2001 | bob | 200.00

3 | 2002 | bob | 800.00

(2 rows)

Начинаем транзакцию, которая уменьшает баланс Боба:

=> **BEGIN**;

=> **UPDATE** accounts **SET** amount = amount - 100 **WHERE** id = 3;

В это же время другая транзакция начисляет проценты на все счета клиентов с общим балансом, равным или превышающим 1000 ₽:

| => **UPDATE** accounts **SET** amount = amount \* 1.01

| **WHERE** client **IN** (

| **SELECT** client

| **FROM** accounts

| **GROUP** **BY** client

| **HAVING** sum(amount) >= 1000

| );

Выполнение оператора UPDATE состоит из двух частей. Сначала фактически выполняется SELECT, который отбирает для обновления строки, соответствующие условию. Поскольку изменение первой транзакции не зафиксировано, вторая транзакция не может его видеть и оно никак не влияет на выбор строк для начисления процентов. Итак, счета Боба попадают под условие и после выполнения обновления его баланс должен увеличиться на 10 ₽.  
  
Второй этап выполнения — выбранные строки обновляются одна за другой. Тут вторая транзакция вынуждена «подвиснуть», поскольку строка id = 3 уже заблокирована первой транзакцией.  
  
Между тем первая транзакция фиксирует изменения:

=> **COMMIT**;

Что же получится в результате?

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob';

id | number | client | amount

----+--------+--------+----------

2 | 2001 | bob | 202.0000

3 | 2002 | bob | 707.0000

(2 rows)

Да, с одной стороны, команда UPDATE не должна видеть изменений второй транзакции. Но с другой — она не должна потерять изменения, зафиксированные во второй транзакции.  
  
После снятия блокировки UPDATE перечитывает строку, которую пытается обновить (но только ее одну!). В результате получается, что Бобу начислено 9 ₽, исходя из суммы 900 ₽. Но если бы у Боба было 900 ₽, его счета вообще не должны были попасть в выборку.  
  
Итак, транзакция получает некорректные данные: часть строк видна на один момент времени, часть — на другой. Взамен потерянного обновления мы снова получаем аномалию *несогласованного чтения*.

Внимательные читатели отмечают, что при некоторой помощи со стороны приложения на уровне Read Committed можно получить и потерянное обновление. Например так:

x := (**SELECT** amount **FROM** accounts **WHERE** id = 1);

**UPDATE** accounts **SET** amount = x + 100 **WHERE** id = 1;

База данных не виновата: она получает два SQL-оператора и ничего не знает о том, что значение x + 100 как-то связано с accounts.amount. Не пишите код таким образом.

Repeatable Read

Отсутствие неповторяющегося и фантомного чтений

Само название уровня изоляции говорит о том, что чтение является повторяемым. Проверим это, а заодно убедимся и в отсутствии фантомных чтений. Для этого в первой транзакции вернем счета Боба в прежнее состояние и создадим новый счет для Чарли:

=> **BEGIN**;

=> **UPDATE** accounts **SET** amount = 200.00 **WHERE** id = 2;

=> **UPDATE** accounts **SET** amount = 800.00 **WHERE** id = 3;

=> **INSERT** **INTO** accounts **VALUES**

(4, '3001', 'charlie', 100.00);

=> **SELECT** \* **FROM** accounts **ORDER** **BY** id;

id | number | client | amount

----+--------+---------+--------

1 | 1001 | alice | 800.00

2 | 2001 | bob | 200.00

3 | 2002 | bob | 800.00

4 | 3001 | charlie | 100.00

(4 rows)

Во втором сеансе начнем транзакцию с уровнем Repeatable Read, указав его в команде BEGIN (уровень первой транзакции не важен).

| => **BEGIN** **ISOLATION** **LEVEL** **REPEATABLE** **READ**;

| => **SELECT** \* **FROM** accounts **ORDER** **BY** id;

| id | number | client | amount

| ----+--------+--------+----------

| 1 | 1001 | alice | 800.00

| 2 | 2001 | bob | 202.0000

| 3 | 2002 | bob | 707.0000

| (3 rows)

Теперь первая транзакция фиксирует изменения, а вторая повторно выполняет тот же самый запрос.

=> **COMMIT**;

| => **SELECT** \* **FROM** accounts **ORDER** **BY** id;

| id | number | client | amount

| ----+--------+--------+----------

| 1 | 1001 | alice | 800.00

| 2 | 2001 | bob | 202.0000

| 3 | 2002 | bob | 707.0000

| (3 rows)

| => **COMMIT**;

Вторая транзакция продолжает видеть ровно те же данные, что и в начале: не видно ни изменений в существующих строках, ни новых строк.  
  
На таком уровне можно не беспокоиться о том, что между двумя операторами что-то поменяется.

Ошибка сериализации в обмен на потерянные изменения

Выше мы говорили о том, что при обновлении одной и той же строки двумя транзакциями на уровне Read Committed может возникнуть аномалия несогласованного чтения. Это происходит из-за того, что ожидающая транзакция перечитывает заблокированную строку и, таким образом, видит ее не на тот же самый момент времени, что остальные строки.  
  
На уровне Repeatable Read такая аномалия не допускается, но, если она все-таки возникает, сделать уже ничего нельзя — поэтому транзакция обрывается с ошибкой сериализации. Проверим, повторив тот же сценарий с процентами:

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob';

id | number | client | amount

----+--------+--------+--------

2 | 2001 | bob | 200.00

3 | 2002 | bob | 800.00

(2 rows)

=> **BEGIN**;

=> **UPDATE** accounts **SET** amount = amount - 100.00 **WHERE** id = 3;

| => **BEGIN** **ISOLATION** **LEVEL** **REPEATABLE** **READ**;

| => **UPDATE** accounts **SET** amount = amount \* 1.01

| **WHERE** client **IN** (

| **SELECT** client

| **FROM** accounts

| **GROUP** **BY** client

| **HAVING** sum(amount) >= 1000

| );

=> **COMMIT**;

| ERROR: could not serialize access due to concurrent update

| => **ROLLBACK**;

Данные остались согласованными:

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob';

id | number | client | amount

----+--------+--------+--------

2 | 2001 | bob | 200.00

3 | 2002 | bob | 700.00

(2 rows)

Такая же ошибка будет и в случае любого другого конкурентного изменения строки, даже если интересующие нас столбцы фактически не изменились.  
  
*Практический вывод*: если приложение использует уровень изоляции Repeatable Read для пишущих транзакций, оно должно быть готово повторять транзакции, завершившиеся ошибкой сериализации. Для только читающих транзакций такой исход невозможен.

Несогласованная запись

Итак, в PostgreSQL на уровне изоляции Repeatable Read предотвращаются все аномалии, описанные в стандарте. Но не все вообще. Оказывается, есть *ровно две* аномалии, которые остаются возможными. (Это верно не только для PostgreSQL, но и для других реализаций изоляции на основе снимков.)  
  
Первая из этих аномалий — *несогласованная запись*.  
  
Пусть действует такое правило согласованности: *допускаются отрицательные суммы на счетах клиента, если общая сумма на всех счетах этого клиента остается неотрицательной*.  
  
Первая транзакция получает сумму на счетах Боба: 900 ₽.

=> **BEGIN** **ISOLATION** **LEVEL** **REPEATABLE** **READ**;

=> **SELECT** sum(amount) **FROM** accounts **WHERE** client = 'bob';

sum

--------

900.00

(1 row)

Вторая транзакция получает ту же сумму.

| => **BEGIN** **ISOLATION** **LEVEL** **REPEATABLE** **READ**;

| => **SELECT** sum(amount) **FROM** accounts **WHERE** client = 'bob';

| sum

| --------

| 900.00

| (1 row)

Первая транзакция справедливо полагает, что сумму одного из счетов можно уменьшить на 600 ₽.

=> **UPDATE** accounts **SET** amount = amount - 600.00 **WHERE** id = 2;

И вторая транзакция приходит к такому же выводу. Но уменьшает другой счет:

| => **UPDATE** accounts **SET** amount = amount - 600.00 **WHERE** id = 3;

| => **COMMIT**;

=> **COMMIT**;

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob';

id | number | client | amount

----+--------+--------+---------

2 | 2001 | bob | -400.00

3 | 2002 | bob | 100.00

(2 rows)

У нас получилось увести баланс Боба в минус, хотя каждая из транзакций работает корректно поодиночке.

Аномалия только читающей транзакции

Это вторая и последняя из аномалий, возможных на уровне Repeatable Read. Чтобы продемонстрировать ее, потребуются три транзакции, две из которых будут изменять данные, а третья — только читать.  
  
Но сначала восстановим состояние счетов Боба:

=> **UPDATE** accounts **SET** amount = 900.00 **WHERE** id = 2;

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob';

id | number | client | amount

----+--------+--------+--------

3 | 2002 | bob | 100.00

2 | 2001 | bob | 900.00

(2 rows)

Первая транзакция начисляет Бобу проценты на сумму средств на всех счетах. Проценты зачисляются на один из его счетов:

=> **BEGIN** **ISOLATION** **LEVEL** **REPEATABLE** **READ**; -- 1

=> **UPDATE** accounts **SET** amount = amount + (

**SELECT** sum(amount) **FROM** accounts **WHERE** client = 'bob'

) \* 0.01

**WHERE** id = 2;

Затем другая транзакция снимает деньги с другого счета Боба и фиксирует свои изменения:

| => **BEGIN** **ISOLATION** **LEVEL** **REPEATABLE** **READ**; -- 2

| => **UPDATE** accounts **SET** amount = amount - 100.00 **WHERE** id = 3;

| => **COMMIT**;

Если в этот момент первая транзакция будет зафиксирована, никакой аномалии не возникнет: мы могли бы считать, что сначала выполнена первая транзакция, а затем вторая (но не наоборот, потому что первая транзакция увидела состояние счета id = 3 до того, как этот счет был изменен второй транзакцией).  
  
Но представим, что в этот момент начинается третья (только читающая) транзакция, которая читает состояние какого-нибудь счета, не затронутого первыми двумя транзакциями:

| => **BEGIN** **ISOLATION** **LEVEL** **REPEATABLE** **READ**; -- 3

| => **SELECT** \* **FROM** accounts **WHERE** client = 'alice';

| id | number | client | amount

| ----+--------+--------+--------

| 1 | 1001 | alice | 800.00

| (1 row)

И только после этого первая транзакция завершается:

=> **COMMIT**;

Какое состояние теперь должна увидеть третья транзакция?

| **SELECT** \* **FROM** accounts **WHERE** client = ‘bob’;

Начавшись, третья транзакция могла видеть изменения второй транзакции (которая уже была зафиксирована), но не первой (которая еще не была зафиксирована). С другой стороны, мы уже установили выше, что вторую транзакцию следует считать начавшейся после первой. Какое состояние ни увидит третья транзакция, оно будет несогласованным — это и есть аномалия только читающей транзакции. Но на уровне Repeatable Read она допускается:

| id | number | client | amount

| ----+--------+--------+--------

| 2 | 2001 | bob | 900.00

| 3 | 2002 | bob | 0.00

| (2 rows)

| => **COMMIT**;

Serializable

На уровне Serializable предотвращаются все возможные аномалии. Фактически Serializable реализован как надстройка над изоляцией на основе снимков данных. Те аномалии, которые не возникают при Repeatable Read (такие, как грязное, неповторяемое, фантомное чтение), не возникают и на уровне Serializable. А те аномалии, которые возникают (несогласованная запись и аномалия только читающей транзакции), обнаруживаются и транзакция прерывается — возникает уже знакомая ошибка сериализации could not serialize access.

Несогласованная запись

Для иллюстрации повторим сценарий с аномалией несогласованной записи:

=> **BEGIN** **ISOLATION** **LEVEL** **SERIALIZABLE**;

=> **SELECT** sum(amount) **FROM** accounts **WHERE** client = 'bob';

sum

----------

910.0000

(1 row)

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

| => **SELECT** sum(amount) **FROM** accounts **WHERE** client = 'bob';

| sum

| ----------

| 910.0000

| (1 row)

=> **UPDATE** accounts **SET** amount = amount - 600.00 **WHERE** id = 2;

| => **UPDATE** accounts **SET** amount = amount - 600.00 **WHERE** id = 3;

| => **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.

Так же, как и на уровне Repeatable Read, приложение, использующее уровень изоляции Serializable, должно повторять транзакции, завершившиеся ошибкой сериализации, о чем нам сообщает и подсказка в сообщении об ошибке.  
  
Мы приобретаем простоту программирования, но цена за нее — принудительный обрыв некоторой доли транзакций и необходимость их повторения. Весь вопрос, конечно, в том, насколько велика эта доля. Если бы обрывались только те транзакции, которые действительно несовместимо пересекаются по данным с другими транзакциями, все было бы неплохо. Но такая реализация неизбежно оказалась бы ресурсоемкой и неэффективной, поскольку пришлось бы отслеживать операции с каждой строкой.  
  
В действительности реализация PostgreSQL такова, что допускает ложноотрицательные срабатывания: будут обрываться и некоторые совершенно нормальные транзакции, которым просто «не повезло». Как мы увидим позже, это зависит от многих причин, например, от наличия подходящих индексов или доступного объема оперативной памяти. Кроме того, есть и некоторые другие (довольно серьезные) ограничения реализации, например, запросы на уровне Serializable не будут работать на репликах, для них не будут использоваться параллельные планы выполнения. И, хотя работа над улучшением реализации не прекращается, но имеющиеся ограничения снижают привлекательность такого уровня изоляции.

Параллельные планы появятся уже в PostgreSQL 12 ([патч](https://commitfest.postgresql.org/22/1004/)). А запросы на репликах могут заработать в PostgreSQL 13 ([другой патч](https://commitfest.postgresql.org/22/1799/)).

Аномалия только читающей транзакции

Чтобы только читающая транзакция не могла привести к аномалии и не могла пострадать от нее, PostgreSQL предлагает интересный механизм: такая транзакция может быть заблокирована до тех пор, пока ее выполнение не будет безопасным. Это единственный случай, когда оператор SELECT может быть заблокирован обновлениями строк. Вот как это выглядит:

=> **UPDATE** accounts **SET** amount = 900.00 **WHERE** id = 2;

=> **UPDATE** accounts **SET** amount = 100.00 **WHERE** id = 3;

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob' **ORDER** **BY** id;

id | number | client | amount

----+--------+--------+--------

2 | 2001 | bob | 900.00

3 | 2002 | bob | 100.00

(2 rows)

=> **BEGIN** **ISOLATION** **LEVEL** **SERIALIZABLE**; -- 1

=> **UPDATE** accounts **SET** amount = amount + (

**SELECT** sum(amount) **FROM** accounts **WHERE** client = 'bob'

) \* 0.01

**WHERE** id = 2;

| => **BEGIN** **ISOLATION** **LEVEL** **SERIALIZABLE**; -- 2

| => **UPDATE** accounts **SET** amount = amount - 100.00 **WHERE** id = 3;

| => **COMMIT**;

Третью транзакцию явно объявляем только читающий (READ ONLY) и откладываемой (DEFERRABLE):

| => **BEGIN** **ISOLATION** **LEVEL** **SERIALIZABLE** **READ** **ONLY** **DEFERRABLE**; -- 3

| => **SELECT** \* **FROM** accounts **WHERE** client = 'alice';

При попытке выполнить запрос транзакция блокируется, потому что иначе ее выполнение приведет к аномалии.

=> **COMMIT**;

И только после того, как первая транзакция зафиксирована, третья продолжает выполнение:

| id | number | client | amount

| ----+--------+--------+--------

| 1 | 1001 | alice | 800.00

| (1 row)

| => **SELECT** \* **FROM** accounts **WHERE** client = 'bob';

| id | number | client | amount

| ----+--------+--------+----------

| 2 | 2001 | bob | 910.0000

| 3 | 2002 | bob | 0.00

| (2 rows)

| => **COMMIT**;

Еще одно важное замечание: если используется изоляция Serializable, то все транзакции в приложении должны использовать этот уровень. Нельзя смешивать транзакции Read Committed (или Repeatable Read) с Serializable. То есть смешивать-то можно, но тогда Serializable будет без всяких предупреждений вести себя, как Repeatable Read. Почему так происходит, мы рассмотрим позже, когда будем говорить о реализации.  
  
Так что если решили использовать Serializble — лучше всего глобально установить умолчательный уровень (хотя это, конечно, не запретит указать неправильный уровень явно):

**ALTER** **SYSTEM** **SET** default\_transaction\_isolation = 'serializable';

Более строгое изложение вопросов, связанных с транзакциями, согласованностью и аномалиями, вы найдете в [книге](https://postgrespro.ru/education/books/dbtech) и [курсе лекций](https://postgrespro.ru/education/university/dbtech) Бориса Асеновича Новикова «Основы технологий баз данных».

Какой уровень изоляции использовать?

Уровень изоляции Read Committed используется в PostgreSQL по умолчанию и, по всей видимости, именно этот уровень используется в абсолютном большинстве приложений. Он удобен тем, что на нем обрыв транзакции возможен только в случае сбоя, но не для предотвращения несогласованности. Иными словами, ошибка сериализации возникнуть не может.  
  
Обратной стороной медали является большое число возможных аномалий, которые были подробно рассмотрены выше. Разработчику приходится постоянно иметь их ввиду и писать код так, чтобы не допускать их появления. Если не получается сформулировать нужные действия в одном SQL-операторе, приходится прибегать к явной установке блокировок. Самое неприятное то, что код сложно тестировать на наличие ошибок, связанных с получением несогласованных данных, а сами ошибки могут возникать непредсказуемым и невоспроизводимым образом и поэтому сложны в исправлении.  
  
Уровень изоляции Repeatable Read снимает часть проблем несогласованности, но, увы, не все. Поэтому приходится не только помнить об оставшихся аномалиях, но и изменять приложение так, чтобы оно корректно обрабатывало ошибки сериализации. Это, конечно, неудобно. Но для только читающих транзакций этот уровень прекрасно дополняет Read Committed и очень удобен, например, для построения отчетов, использующих несколько SQL-запросов.  
  
Наконец, уровень Serializable позволяет вообще не заботиться о несогласованности, что значительно упрощает написание кода. Единственное, что требуется от приложения — уметь повторять любую транзакций при получении ошибки сериализации. Но доля прерываемых транзакций, дополнительные накладные расходы, а также невозможность распараллеливать запросы могут существенно снизить пропускную способность системы. Также следует учитывать, что уровень Serializable не применим на репликах, и то, что его нельзя смешивать с другими уровнями изоляции.