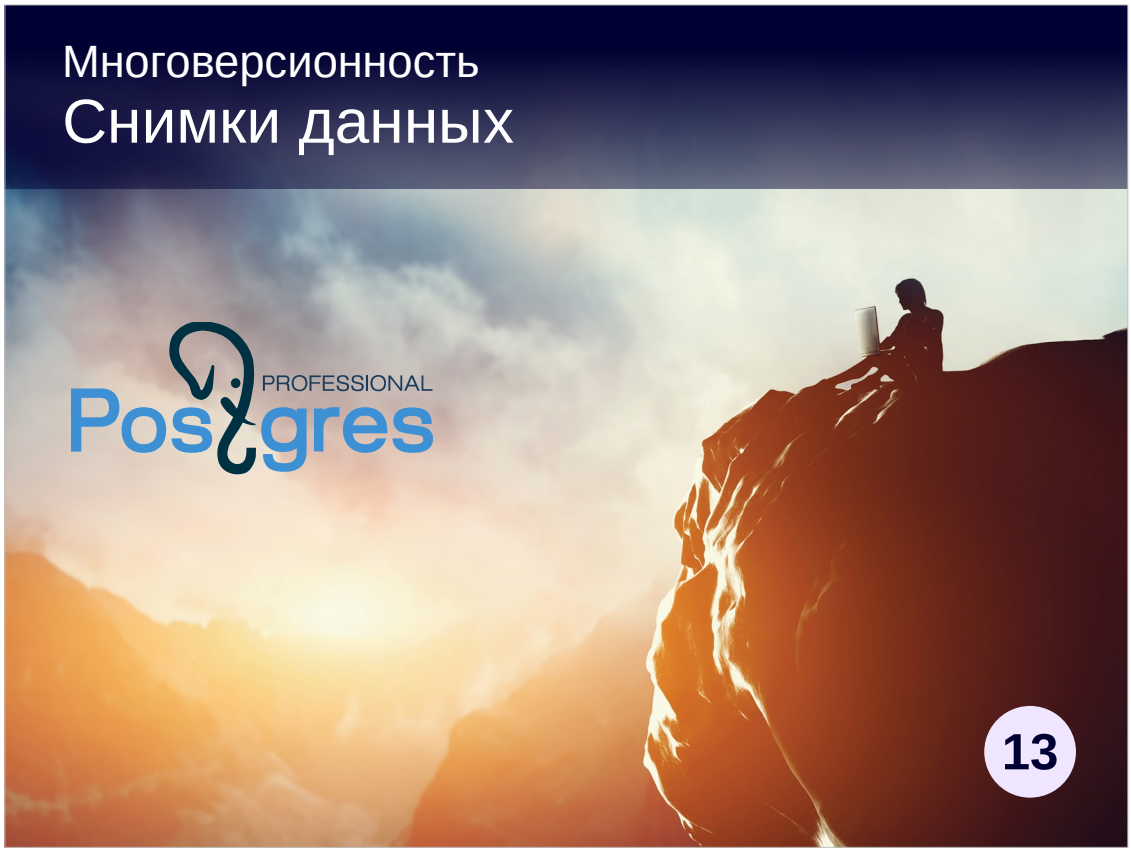


Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов

Обратная связь
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

Темы

Видимость версий строк

Снимок данных

Виртуальный номер транзакции

Горизонт транзакции и базы данных

Экспорт снимка

Снимок

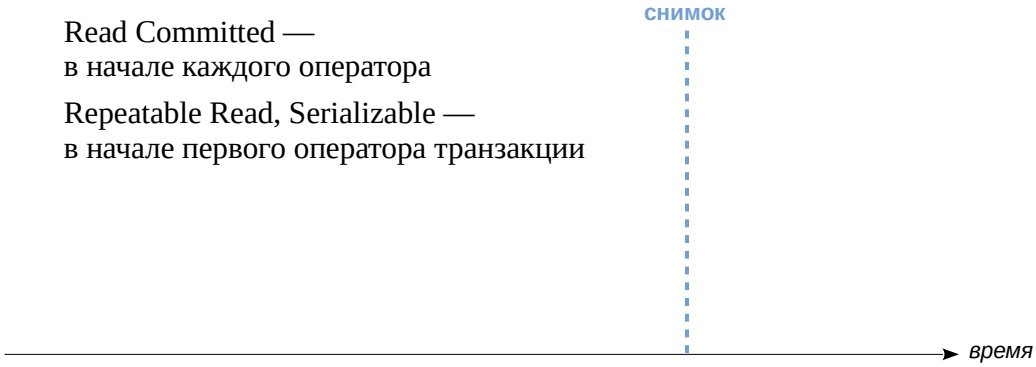
Дает согласованную картину данных на момент времени

неформально: видны только зафиксированные на этот момент данные

Создается:

Read Committed —
в начале каждого оператора

Repeatable Read, Serializable —
в начале первого оператора транзакции



В теме «Страницы и версии строк» мы видели, что в страницах данных физически может находиться несколько версий одной и той же строки.

Транзакции работают со *снимком данных,* который определяет, какие версии должны быть видны, а какие — нет, чтобы обеспечить согласованную картину данных на определенный момент времени (на момент создания снимка — показан на рисунке синим цветом).

Изоляция на основе снимков данных (snapshot isolation) требует, чтобы транзакции были видны только те данные, которые были зафиксированы на момент создания снимка. Это гарантирует, что данные снимка будут согласованными (в обычном ACID-смысле).

На уровне изоляции Read Committed снимок создается в начале каждого оператора транзакции. Снимок активен, пока выполняется оператор.

На уровнях Repeatable Read и Serializable снимок создается один раз в начале первого оператора транзакции. Такой снимок остается активным до самого конца транзакции.

Видимость версий строк

Видимость версии строки ограничена xmin и xmax

Версия попадает в снимок, когда

- изменения транзакции xmin видны для снимка,
- изменения транзакции xmax не видны для снимка

Изменения транзакции видны, когда

- либо это та же самая транзакция, что создала снимок,
- либо она завершилась фиксацией до момента создания снимка

Отдельные правила для видимости собственных изменений

- учитывается порядковый номер операции в транзакции (cmin/cmax)

Снимок данных не является физической копией нужных версий строк.

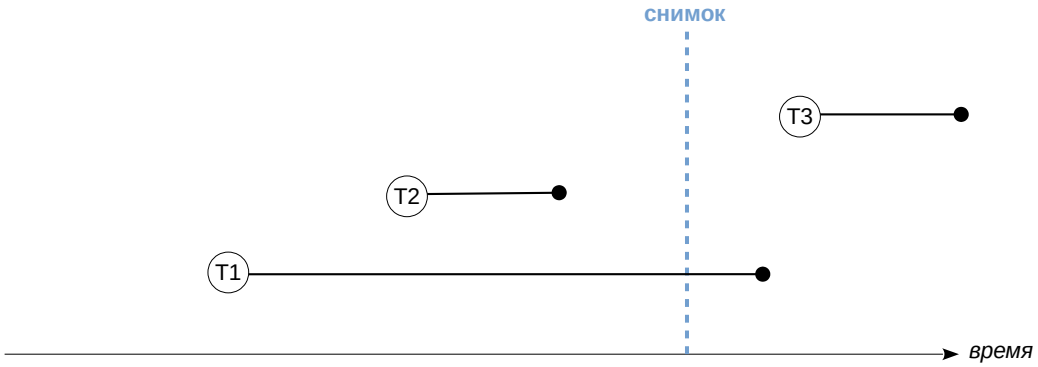
Будет или нет данная версия строки видна в снимке, определяется двумя полями ее заголовка — xmin и xmax, — то есть номерами создавшей и удалившей транзакций. Такие интервалы не пересекаются, поэтому одна строка представлена в любом снимке максимум одной своей версией.

Точные правила видимости довольно сложны и учитывают различные «крайние» случаи. Чуть упрощая, можно сказать, что версия строки видна, когда в снимке *видны* изменения, сделанные транзакцией xmin, и *не видны* изменения, сделанные транзакцией xmax.

В свою очередь, изменения транзакции видны в снимке, если либо это та же самая транзакция, что создала снимок (она сама видит свои же изменения), либо транзакция была зафиксирована до создания снимка.

Несколько усложняет картину случай определения видимости собственных изменений транзакции. Здесь может потребоваться видеть только часть таких изменений. Например, курсор, открытый в определенный момент, ни при каком уровне изоляции не должен увидеть изменений, сделанных после этого момента. Для этого в заголовке версии строки есть специальное поле (псевдостолбцы cmin и cmax), показывающее номер операции внутри транзакции, и этот номер тоже принимается во внимание.

Видимость версий строк



Рассмотрим, как это работает, на простом примере.

На рисунке синим цветом отмечен момент создания снимка. Черные линии показывают транзакции от момента начала до момента фиксации (оборванные транзакции никогда не попадают в снимок).

В данном случае в снимке должны быть видны только изменения, сделанные транзакцией T2, поскольку она зафиксирована до того, как создан снимок.

Изменения транзакции T1 не должны быть видны, поскольку в момент создания снимка она еще не была зафиксирована. А изменения транзакции T3 не должны быть видны, поскольку она еще не началась в момент создания снимка.

Видимость версий строк

Воспроизведем ситуацию, показанную на слайде. Каждая из трех транзакций будет добавлять новую строку в таблицу.

```
=> CREATE DATABASE mvcc_snapshots;
```

```
CREATE DATABASE
```

```
=> \c mvcc_snapshots
```

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

```
=> CREATE TABLE t(s text);
```

```
CREATE TABLE
```

Первая транзакция (не завершится до создания снимка):

```
=> \c mvcc_snapshots
```

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

```
=> BEGIN;
```

```
BEGIN
```

```
=> INSERT INTO t VALUES ('first');
```

```
INSERT 0 1
```

```
=> SELECT txid_current();
```

```
 txid_current
--------------
       695996
(1 row)
```

Вторая транзакция (завершается сразу):

```
=> \c mvcc_snapshots
```

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

```
=> BEGIN;
```

```
BEGIN
```

```
=> INSERT INTO t VALUES ('second');
```

```
INSERT 0 1
```

```
=> SELECT txid_current();
```

```
 txid_current
--------------
       695997
(1 row)
```

```
=> COMMIT;
```

```
COMMIT
```

Создаем снимок в транзакции в отдельном сеансе.

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

```
BEGIN
```

```
=> SELECT * FROM t;
```

```
   s
--------
 second
(1 row)
```

Завершаем первую транзакцию после того, как создан снимок:

```
=> COMMIT;
```

```
COMMIT
```

Третья транзакция начинается после создания снимка:

```
=> \c mvcc_snapshots

You are now connected to database "mvcc_snapshots" as user "student".

=> BEGIN;

BEGIN

=> INSERT INTO t VALUES ('third');

INSERT 0 1

=> SELECT txid_current();

 txid_current
--------------
       695998
(1 row)


=> COMMIT;

COMMIT
```

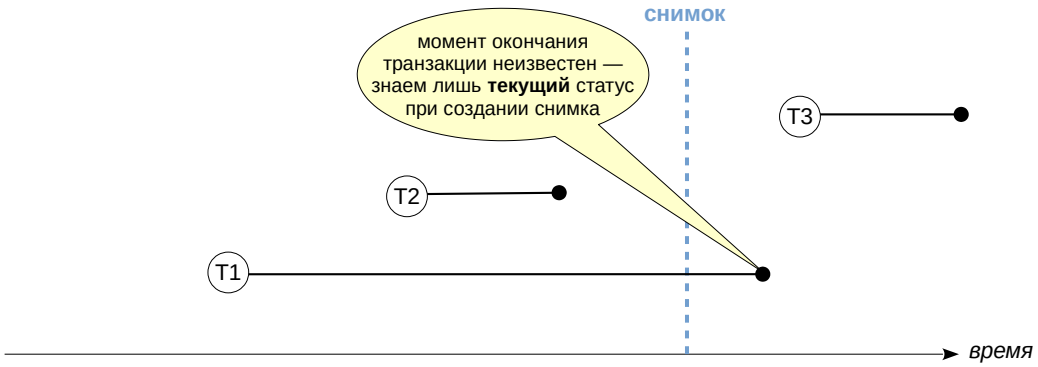
Очевидно, что в нашем снимке по-прежнему видна одна строка:

```
=> SELECT *, xmin, xmax FROM t;

   s    |  xmin  | xmax
--------+--------+------
 second | 695997 |    0
(1 row)
```

Вопрос в том, как это понимает PostgreSQL.

Видимость версий строк



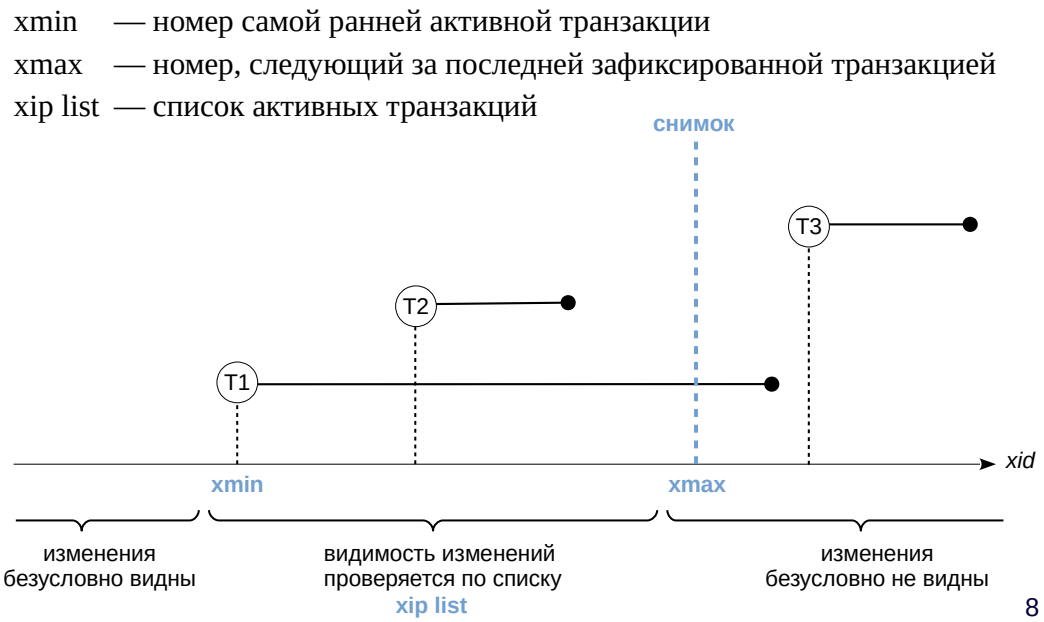
как отличить T1 от T2 после того, как снимок создан?

Сложность реализации состоит в том, что нам известен только момент *начала* транзакции. Он определяется номером транзакции (xid), с которым можно сравнить номер xmin или xmax версии строки. Но момент *завершения* транзакции нигде не записывается. Все, что мы может узнать — это *текущий* статус транзакций.

То есть после того, как транзакция T1 завершится фиксацией, она ничем не будет отличаться от транзакции T2: нет способа выяснить, что одна из них была зафиксирована *до* создания снимка, а другая — *после*.

Поскольку историческая информация об активности транзакций нигде не хранится, снимок можно создать только в текущий момент. Нельзя, например, создать снимок, показывающий согласованные данные по состоянию на пять минут назад, даже если все необходимые версии строк существуют в табличных страницах.

Снимок данных



В момент создания снимка данных запоминаются несколько значений, которые и определяют снимок:

- **xmin** — нижняя граница снимка, в качестве которой выступает номер самой ранней активной транзакции.

Все транзакции с меньшими номерами либо зафиксированы, и тогда их изменения безусловно видны в снимке, либо отменены, и тогда изменения игнорируются.

- **xmax** — верхняя граница снимка, в качестве которой берется номер, следующий за номером последней зафиксированной транзакции. Верхняя граница определяет момент, в который был сделан снимок. Обратите внимание, что момент задается не временем (как было условно показано на предыдущих слайдах), а увеличивающимися номерами транзакций.

Все транзакции с номерами, большими или равными xmax, не существовали в момент создания снимка, поэтому изменения таких транзакций точно не видны.

- **xip_list** (xid-in-progress list) — список активных транзакций.

Этот список используется для того, чтобы не показывать в снимке изменения транзакций, которые уже завершились, но в момент создания снимка были еще активны.

Информацию о снимке можно получить с помощью функции pg_current_snapshot() и нескольких других, см. https://postgrespro.ru/docs/postgresql/13/functions-info

Снимок данных

Видимость версий строк определяется снимком. Посмотрим на него:

```
=> SELECT pg_current_snapshot(); -- txid_current_snapshot() до v.13
```

```
 pg_current_snapshot
----------------------
 695996:695998:695996
(1 row)
```

Снимок определяется в момент создания тремя основными значениями, которые функция выводит через двоеточия:

- xmin - номер самой ранней активной транзакции;
- xmax - номер, следующий за последней зафиксированной транзакцией (определяет момент создания снимка);
- xip_list - список номеров активных транзакций (в данном случае она одна).

В снимке должны быть видны изменения транзакций с номерами:

- xid < xmin;
- xmin <= xid < xmax, за исключением транзакций из xip_list.

В нашем случае:

- xid < 695996;
- 695996 <= xid < 695998, за исключением 695996.

```
=> SELECT *, xmin, xmax FROM t;
```

```
   s    |  xmin  | xmax
--------+--------+------
 first  | 695996 |    0
 second | 695997 |    0
 third  | 695998 |    0
(3 rows)
```

Первая строка не видна - она создана транзакцией 695996, которая входит в список активных.

Вторая строка видна - она создана транзакцией 695997, которая попадает в диапазон снимка.

Третья строка не видна - она создана транзакцией 695998, которая не входит в диапазон снимка.

Виртуальные транзакции

Только читающая транзакция никак не влияет на видимость

- обслуживающий процесс выделяет виртуальный номер
- виртуальный номер не учитывается в снимках
- виртуальный номер никогда не попадает в страницы
- настоящий номер выделяется при первом изменении данных

На практике PostgreSQL использует оптимизацию, позволяющую «экономить» номера транзакций.

Если транзакция только читает данные, то она никак не влияет на видимость версий строк и ее номер не надо учитывать в снимках данных.

Поэтому вначале обслуживающий процесс выдает транзакции *виртуальный номер* (virtual xid). Номер состоит из идентификатора процесса и последовательного числа.

Выдача этого номера не требует синхронизации между всеми процессами и поэтому выполняется очень быстро. С другой причиной использования виртуальных номеров мы познакомимся в теме «Заморозка» этого модуля.

Такой виртуальный номер нельзя записывать в страницы данных, потому что при следующем обращении к странице он может потерять всякий смысл.

Как только транзакция начинает менять данные, ей сразу же выдается настоящий номер, который может появляться в страницах данных (в полях xmin и xmax версий строк).

Виртуальные транзакции

Только читающие транзакции никак не влияют на видимость версий строк и не учитываются в снимках данных. Они не имеют настоящего номера:

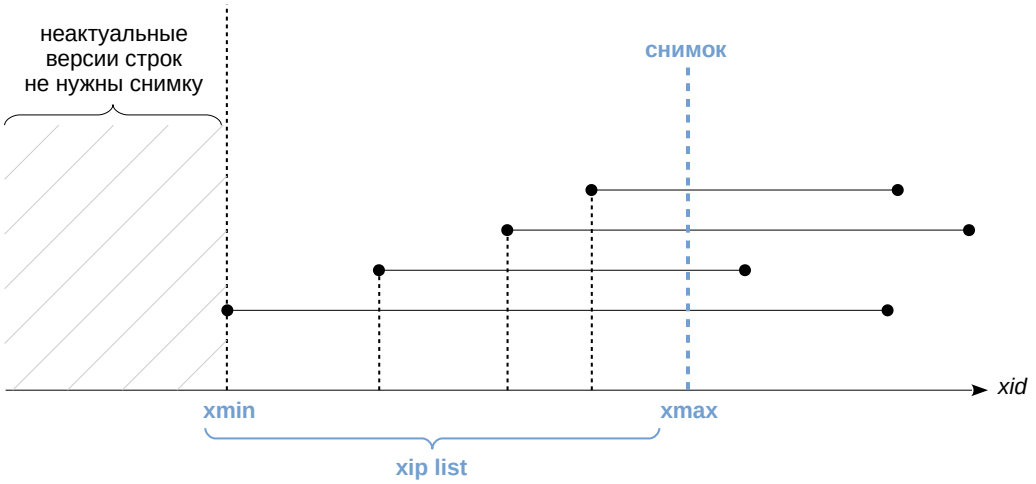
```
=> SELECT pg_current_xact_id_if_assigned(); -- txid_current_if_assigned() до v.13

 pg_current_xact_id_if_assigned
--------------------------------

(1 row)
```

Настоящий номер присваивается при первом изменении, выполненном транзакцией, а также при вызове функции pg_current_xact_id().

Горизонт снимка



12

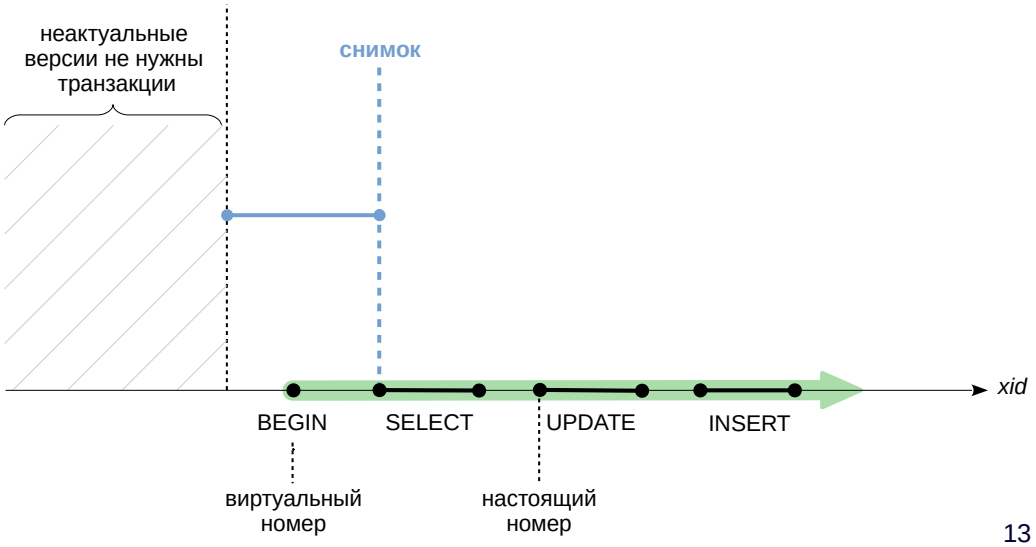
Номер самой ранней из активных транзакций (xmin снимка) имеет важный смысл — он определяет *горизонт снимка*.

Горизонт событий в астрофизике — это граница, за которой наблюдатель не может увидеть никакие события. А в нашем случае горизонт снимка — это граница, за которой оператор, использующий снимок, не может увидеть неактуальные версии строк.

Действительно, видеть неактуальную версию требуется только в том случае, когда актуальная создана еще не завершившейся транзакцией, и поэтому пока не видна. Но за горизонтом все транзакции уже гарантированно завершились.

Горизонт транзакции

Repeatable Read, Serializable

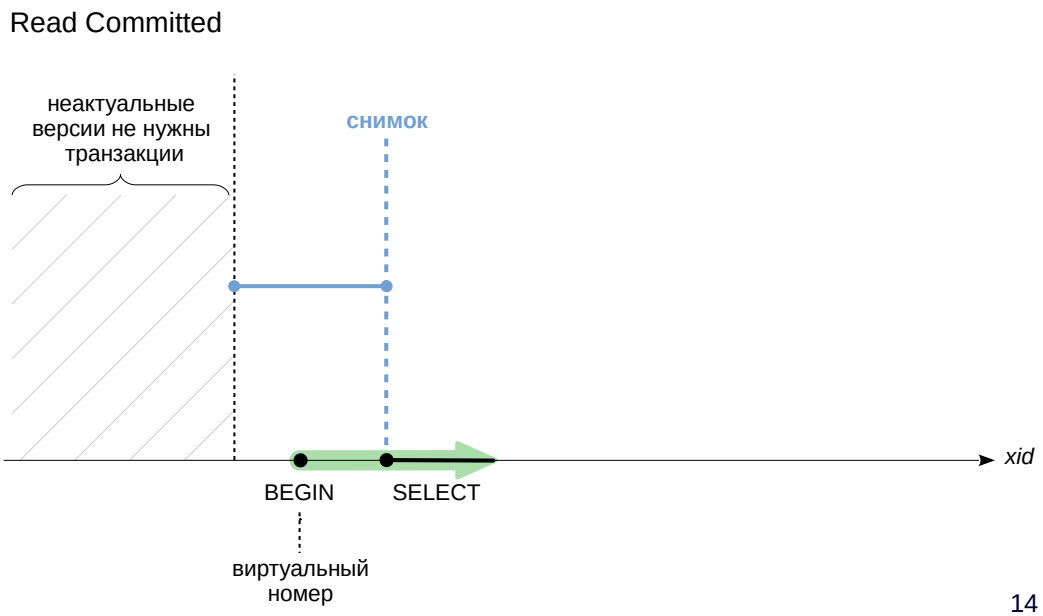


На разных уровнях изоляции транзакции по-разному используют снимки данных.

Для уровней Repeatable Read и Serializable транзакция строит снимок при первом обращении к серверу и использует его до своего окончания.

Левая граница снимка (начало горизонтального отрезка на слайде) определяющая горизонт транзакции, равна номеру самой ранней активной транзакции. Поскольку текущая транзакция сама является активной, горизонт никогда не может оказаться правее настоящего номера данной транзакции.

Горизонт транзакции

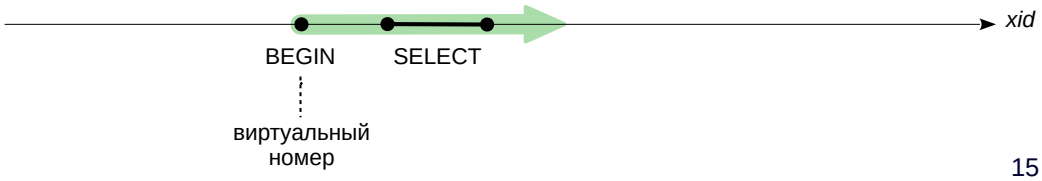


В случае уровня изоляции Read Committed снимков может быть несколько.

Сначала транзакция получает виртуальный номер. При первом обращении к базе данных строится снимок.

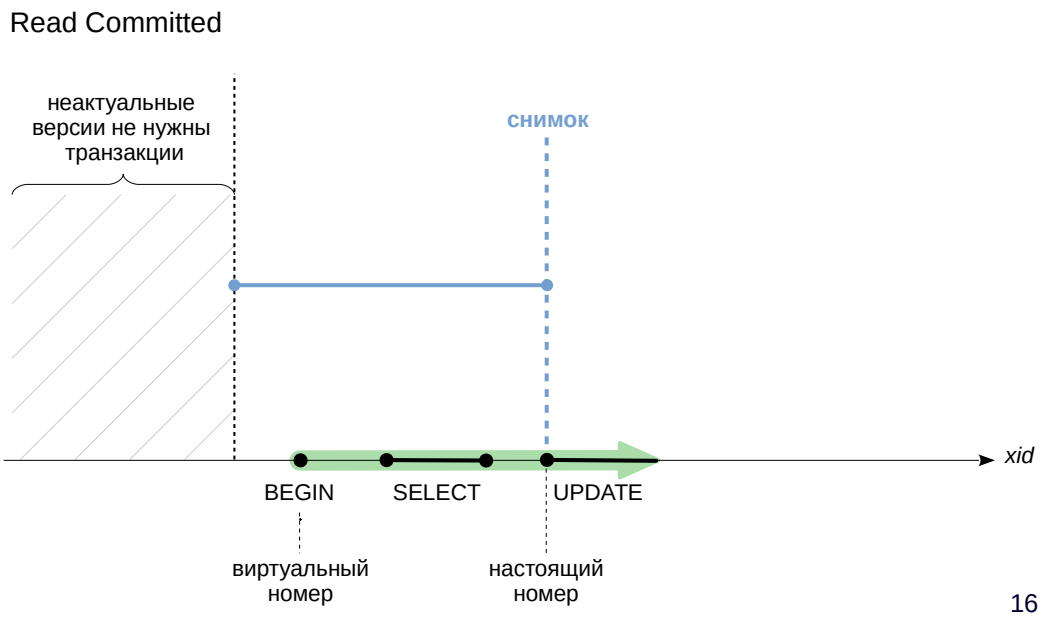
Горизонт транзакции

Read Committed



По окончании запроса снимок освобождается. Поскольку в нашем примере первый оператор транзакции не менял данные, она не получила настоящий номер и не будет удерживать горизонт до начала следующего оператора.

Горизонт транзакции

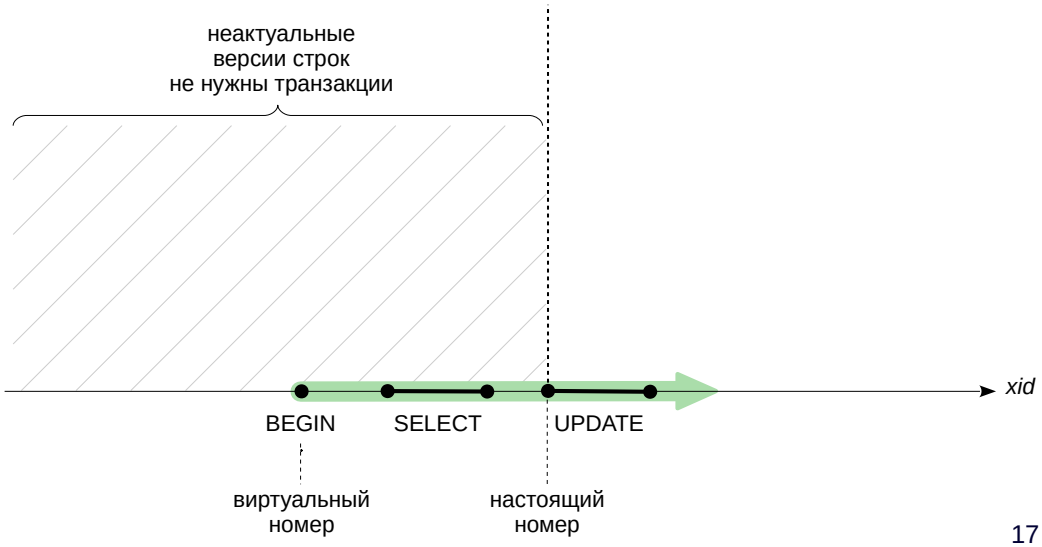


Если транзакция начинает менять данные, ей выдается настоящий номер. В этот же момент строится новый снимок, горизонт которого (xmin), как и всегда, находится на уровне первой активной транзакции.

Поскольку текущая транзакция сама является активной, горизонт никогда не может оказаться правее номера данной транзакции.

Горизонт транзакции

Read Committed

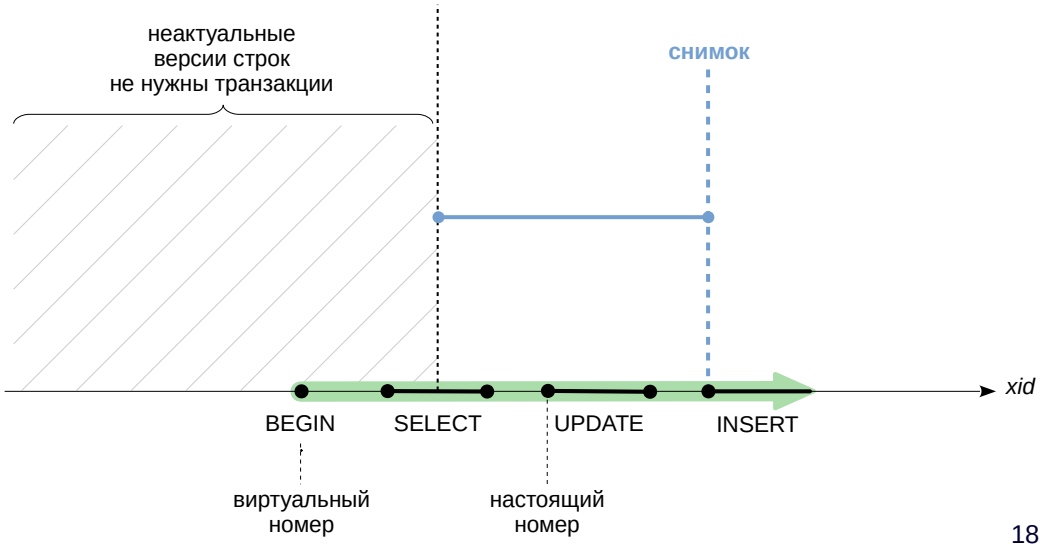


Когда оператор завершает работу, снимок освобождается, а транзакция держит горизонт на уровне своего номера на случай отката.

Если транзакция будет долго находиться в таком состоянии (оно называется idle in transaction), не выполняя никаких команд, горизонт будет оставаться на месте.

Горизонт транзакции

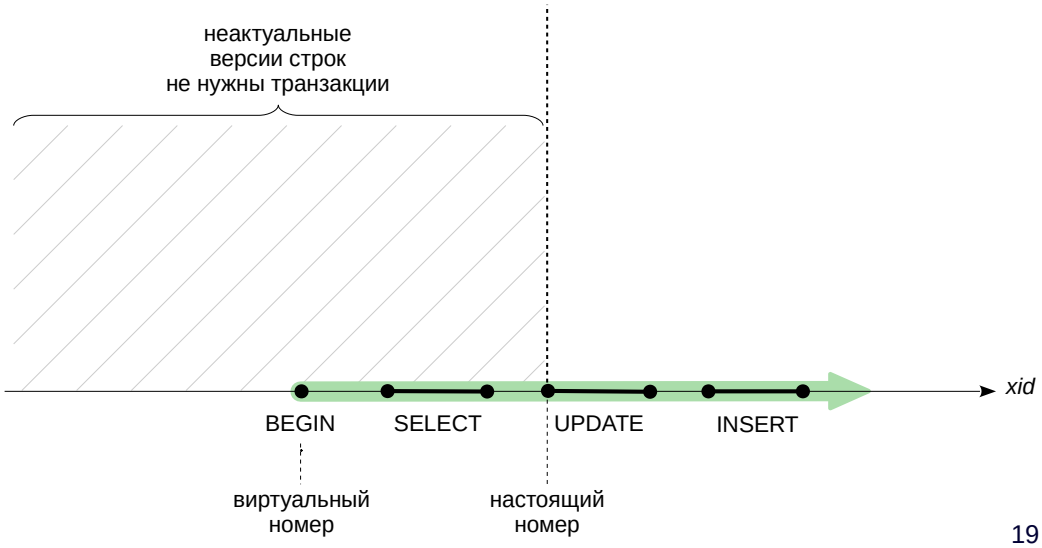
Read Committed



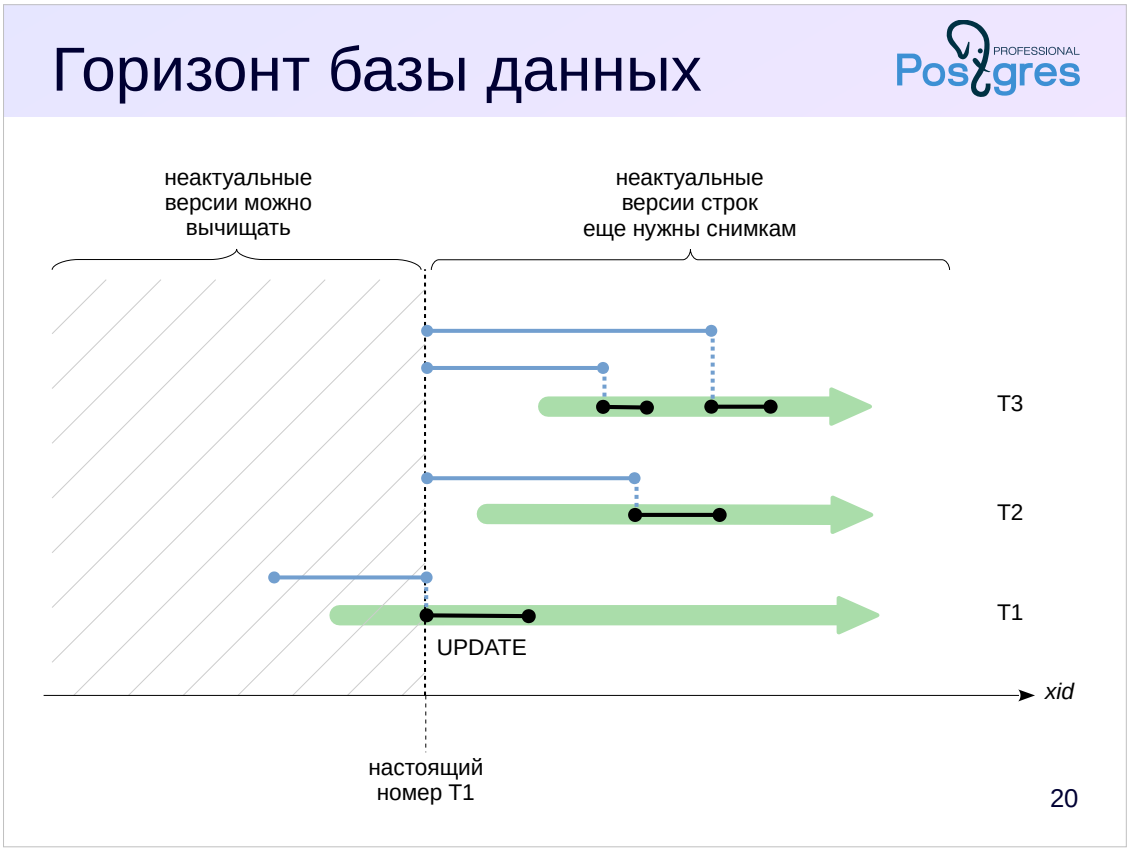
Следующий оператор использует новый снимок, горизонт которого учитывает номера других активных транзакций. Текущая транзакция по-прежнему активна, поэтому горизонт снимка (xmin) не может быть больше ее номера.

Горизонт транзакции

Read Committed



Между операторами, когда нет активного снимка, горизонт опять смещается к номеру транзакции. Если транзакция так и не получила настоящий номер (не начала изменять данные), она вообще не будет держать горизонт.

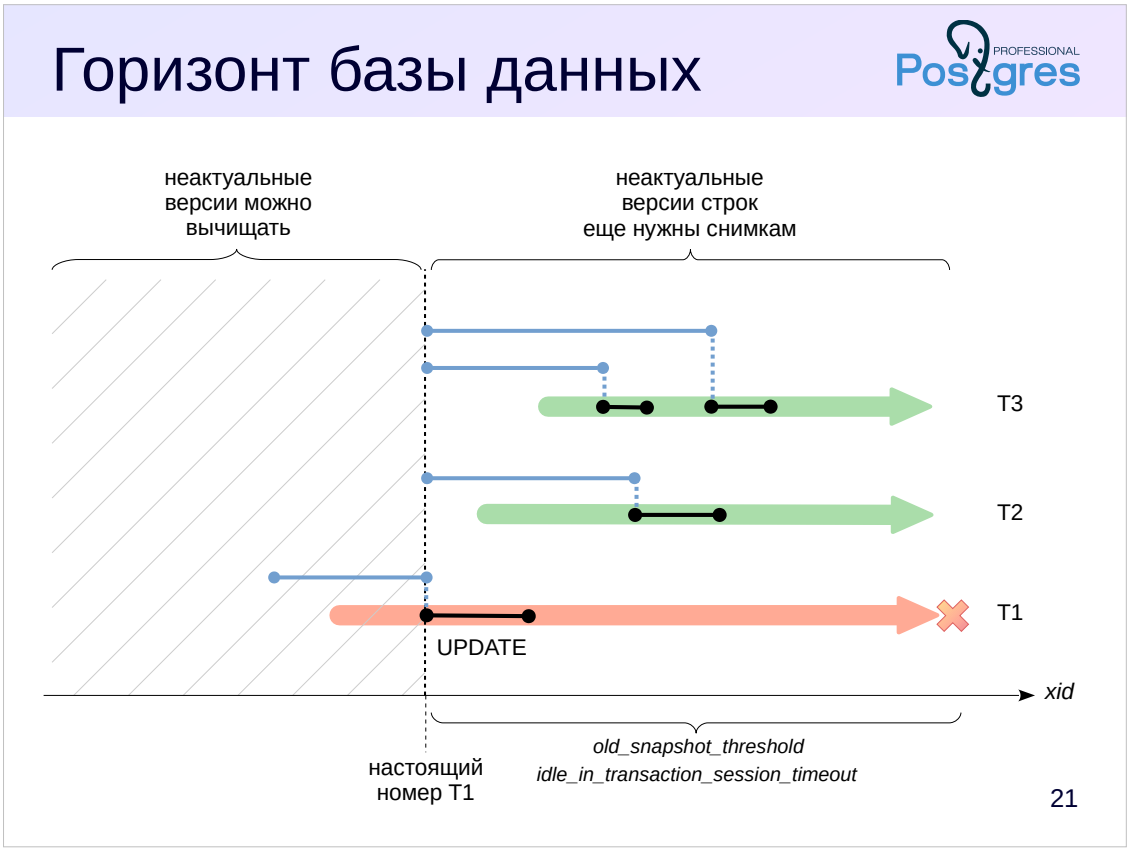


Также можно определить и горизонт базы данных как минимальный из горизонтов всех транзакций. При освобождении снимка горизонт базы данных смещается вправо к горизонту следующего активного снимка. Обратного движения быть не может, поскольку xmin вновь возникающих снимков — минимальный номер активной транзакции — сдвигается только вправо.

Неактуальные версии строк за горизонтом базы данных уже никогда не будут нужны ни одной транзакции. Такие версии строк могут быть безопасно вычищены.

На слайде показан пример трех транзакций с уровнем изоляции Read Committed. Для оператора UPDATE транзакции T1 был построен снимок. Когда оператор отрабатывает, снимок удаляется и перестает удерживать горизонт. Однако транзакция T1 по-прежнему активна, и поэтому горизонты снимков транзакций T2 и T3 не сдвинутся правее номера транзакции T1. Тем самым активная транзакция, если у нее есть настоящий номер, удерживает горизонт самим фактом своего существования.

А это означает, что неактуальные версии строк в этой БД не могут быть очищены. При этом «долгоиграющая» транзакция может никак не пересекаться по данным с другими транзакциями — это совершенно не важно, горизонт базы данных один на всех.



Если описанная ситуация действительно создает проблемы и нет способа избежать ее на уровне приложения, помогут два параметра:

- *old_snapshot_threshold* определяет максимальное время жизни снимка. После этого времени сервер получает право удалять неактуальные версии строк, а если они понадобятся «долгоиграющей» транзакции, то она получит ошибку snapshot too old.

- *idle_in_transaction_session_timeout* определяет максимальное время жизни бездействующей транзакции. После этого времени транзакция прерывается.

Горизонт транзакции и базы данных

Свой горизонт транзакция может увидеть в системном каталоге:

```
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
```

```
 backend_xmin
--------------
       695996
(1 row)
```

Хотя наша транзакция является виртуальной и не влияет на видимость других транзакций, она обращается к данным, используя снимок, и поэтому удерживает горизонт базы данных.

После завершения транзакции горизонт продвигается вперед, позволяя очищать неактуальные версии строк:

```
=> COMMIT;
```

```
COMMIT
```

```
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
```

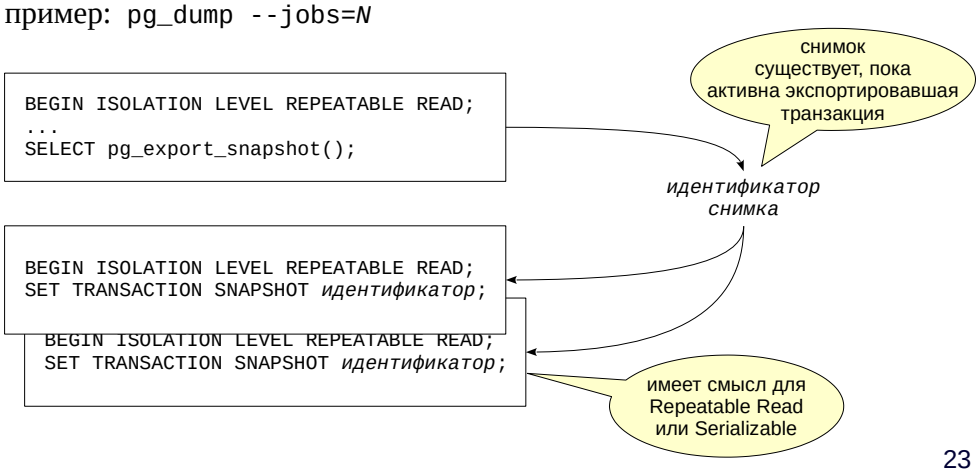
```
 backend_xmin
--------------
       695999
(1 row)
```

Экспорт снимка

Задача

распределить работу между несколькими одновременно работающими транзакциями так, чтобы они видели одни и те же данные

пример: `pg_dump --jobs=N`



Бывают ситуации, когда несколько транзакций должны гарантированно видеть одну и ту же картину данных. Разумеется, нельзя полагаться на то, что картины совпадут просто из-за того, что транзакции запущены «одновременно». Для этого есть механизм экспорта и импорта снимка.

Функция pg_export_snapshot возвращает идентификатор снимка, который может быть передан (внешними по отношению к СУБД средствами) в другую транзакцию.

https://postgrespro.ru/docs/postgresql/13/functions-admin#FUNCTIONS-SNAPSHOT-SYNCHRONIZATION

Другая транзакция может импортировать снимок с помощью команды SET TRANSACTION SNAPSHOT до выполнения первого запроса в ней. Предварительно надо установить и уровень изоляции Repeatable Read или Serializable, потому что на уровне Read Committed операторы будут использовать собственные снимки.

Время жизни экспортированного снимка совпадает со временем жизни экспортирующей транзакции.

https://postgrespro.ru/docs/postgresql/13/sql-set-transaction

Экспорт снимка применяется, например, утилитой pg_dump при параллельном режиме работы или логической репликацией для получения начального состояния таблиц, входящих в подписку (подробнее см. курс DBA3 «Резервное копирование и репликация»).

Итоги

Снимок содержит информацию,
необходимую для определения видимости версий строк

Время создания снимка определяет уровень изоляции

Снимок определяет горизонт транзакции, влияющий
на возможность удаления неактуальных версий

Практика

1. Воспроизведите ситуацию, при которой одна транзакция еще видит удаленную строку, а другая — уже нет.
   Посмотрите снимки данных этих транзакций и значения полей xmin и xmax удаленной строки.
   Объясните видимость на основании этих данных.
2. Если в запросе вызывается функция, содержащая другой запрос, какой снимок данных будет использоваться для «вложенного» запроса?
   Проверьте уровни изоляции Read Committed и Repeatable Read и категории изменчивости функций volatile и stable.
3. В одной транзакции экспортируйте снимок, затем в другой транзакции измените данные. Импортируйте снимок и проверьте, что в нем видны еще не измененные данные.

2. Можно воспользоваться следующим шаблоном функции, в предположении, что создана таблица t:

```
CREATE FUNCTION test() RETURNS bigint AS $$
  SELECT count(*) FROM t;
$$
VOLATILE -- или STABLE
LANGUAGE sql;
```

Также может пригодиться функция pg_sleep(n), вызывающая задержку на n секунд.

1. Снимки данных двух транзакций

Таблица с одной строкой:

```
=> CREATE DATABASE mvcc_snapshots;

CREATE DATABASE

=> \c mvcc_snapshots

You are now connected to database "mvcc_snapshots" as user "student".

=> CREATE TABLE t(n integer);

CREATE TABLE

=> BEGIN;

BEGIN

=> INSERT INTO t(n) VALUES (1);

INSERT 0 1

=> SELECT pg_current_xact_id();

 pg_current_xact_id
--------------------
             749678
(1 row)


=> COMMIT;

COMMIT
```

Первая транзакция видит строку:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;

BEGIN

=> SELECT * FROM t;

 n
---
 1
(1 row)


=> SELECT pg_current_xact_id();

 pg_current_xact_id
--------------------
             749679
(1 row)


=> SELECT pg_current_snapshot();

 pg_current_snapshot
---------------------
 749679:749679:
(1 row)
```

Вторая транзакция в другом сеансе удаляет строку:

```
=> \c mvcc_snapshots

You are now connected to database "mvcc_snapshots" as user "student".

=> BEGIN ISOLATION LEVEL REPEATABLE READ;

BEGIN

=> DELETE FROM t;

DELETE 1

=> SELECT * FROM t;
```

```
 n
---
(0 rows)
```

```
=> SELECT pg_current_xact_id();
```

```
 pg_current_xact_id
--------------------
             749680
(1 row)
```

```
=> SELECT pg_current_snapshot();
```

```
 pg_current_snapshot
---------------------
 749679:749679:
(1 row)
```

Первая транзакция продолжает видеть строку:

```
=> SELECT xmin, xmax, * FROM t;
```

```
  xmin  |  xmax  | n
--------+--------+---
 749678 | 749680 | 1
(1 row)
```

Снимки обеих транзакций одинаковы, и по правилам видимости они обе должны были бы видеть строку, так как

- изменения транзакции xmin = 749678 видны в снимке (749678 < 749679),
- изменения транзакции xmax = 749679 не видны в снимке (749679 > 749679).

Однако вторая транзакция все-таки не видит строку, поскольку она сама ее удалила, а это - исключение из обычных правил видимости.

```
=> COMMIT;
```

```
COMMIT
```

```
=> COMMIT;
```

```
COMMIT
```

2. Снимок вложенного запроса

Внутри функции, подсчитывающей число строк таблицы, вставим односекундную задержку для удобства тестирования:

```
=> CREATE FUNCTION test() RETURNS bigint AS $$
SELECT pg_sleep(1);
SELECT count(*) FROM t;
$$ VOLATILE LANGUAGE sql;
```

```
CREATE FUNCTION
```

Теперь начинаем транзакцию с уровнем изоляции Read Committed и в запросе несколько раз вызываем функцию, одновременно подсчитывая количество строк подзапросом.

```
=> BEGIN ISOLATION LEVEL READ COMMITTED;
```

```
BEGIN
```

```
=> SELECT (SELECT count(*) FROM t) AS cnt, test()
FROM generate_series(1,5);
```

Параллельно выполняем другую транзакцию, которая увеличивает число строк в таблице. Если основной запрос и вложенный запрос используют разные снимки, мы обнаружим это по разнице в результате.

```
=> INSERT INTO t VALUES (1);
```

```
INSERT 0 1
```

```
 cnt | test
-----+------
   0 |    0
   0 |    0
   0 |    1
   0 |    1
   0 |    1
(5 rows)


=> END;

COMMIT
```

Значение первого столбца не изменяется - запрос использует снимок, созданный в начале выполнения. Однако значение во втором столбце изменяется - запросы внутри volatile-функции используют отдельные снимки.

Повторим эксперимент для уровня изоляции Repeatable Read:

```
=> TRUNCATE t;

TRUNCATE TABLE

=> BEGIN ISOLATION LEVEL REPEATABLE READ;

BEGIN

=> SELECT (SELECT count(*) FROM t) AS cnt, test()
FROM generate_series(1,5);
```

```
  => INSERT INTO t VALUES (1);

  INSERT 0 1
```

```
 cnt | test
-----+------
   0 |    0
   0 |    0
   0 |    0
   0 |    0
   0 |    0
(5 rows)


=> END;

COMMIT
```

Теперь еще раз то же самое для функции с категорией изменчивости stable.

```
=> ALTER FUNCTION test STABLE;

ALTER FUNCTION
```

Уровень Read Committed:

```
=> TRUNCATE t;

TRUNCATE TABLE

=> BEGIN ISOLATION LEVEL READ COMMITTED;

BEGIN

=> SELECT (SELECT count(*) FROM t) AS cnt, test()
FROM generate_series(1,5);
```

```
  => INSERT INTO t VALUES (1);

  INSERT 0 1
```

```
 cnt | test
-----+------
   0 |    0
   0 |    0
   0 |    0
   0 |    0
   0 |    0
(5 rows)


=> END;

COMMIT
```

Уровень Repeatable Read:

```
=> TRUNCATE t;
```

```
TRUNCATE TABLE
```

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

```
BEGIN
```

```
=> SELECT (SELECT count(*) FROM t) AS cnt, test()
FROM generate_series(1,5);
```

```
  => INSERT INTO t VALUES (1);
```

```
  INSERT 0 1
```

```
 cnt | test
-----+------
   0 |    0
   0 |    0
   0 |    0
   0 |    0
   0 |    0
(5 rows)
```

```
=> END;
```

```
COMMIT
```

Выводы:

- Запросы в volatile-функциях на уровне изоляции Read Committed используют собственные снимки и могут видеть изменения в процессе работы оператора. Такая возможность скорее опасна, чем полезна.
- При любых других комбинациях категории изменчивости и уровня изоляции вложенные запросы используют снимок основного запроса.

3. Экспорт снимка

Функция pg_export_snapshot возвращает идентификатор снимка, который можно передать в другую транзакцию (внешними по отношению к СУБД средствами):

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

```
BEGIN
```

```
=> SELECT count(*) FROM t;
```

```
 count
-------
     1
(1 row)
```

```
SELECT pg_export_snapshot();
```

```
 pg_export_snapshot
---------------------
 00000005-00000010-1
(1 row)
```

В другом сеансе удаляем все строки из таблицы:

```
  => DELETE FROM t;
```

```
  DELETE 1
```

Затем транзакция импортирует снимок и видит в нем те же данные, которые доступны транзакции, которая экспортировала этот снимок:

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

```
  BEGIN
```

```
  => SET TRANSACTION SNAPSHOT '00000005-00000010-1';
```

```
  SET
```

```
  => SELECT count(*) FROM t;
```

```
 count
-------
     1
(1 row)
```

```
=> COMMIT;
```

```
COMMIT
```

```
=> COMMIT;
```

```
COMMIT
```