

# Архитектура Изоляция и многоверсионность

Postgres  
PROFESSIONAL

16

## Авторские права

© Postgres Professional, 2017–2024

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

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

## Использование материалов курса

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

## Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

[edu@postgrespro.ru](mailto:edu@postgrespro.ru)

## Отказ от ответственности

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

Многоверсионность

Снимок данных

Уровни изоляции

Очистка и ее горизонт

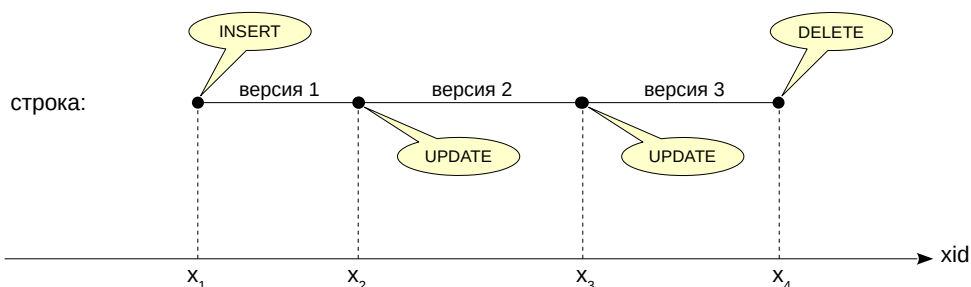
Блокировки

Статус транзакций

## Наличие нескольких версий одной и той же строки

версии различаются временем действия

время = номер транзакции (номера выдаются по возрастанию)



3

При одновременной работе нескольких сеансов возникает вопрос: что делать, если две транзакции одновременно обращаются к одной и той же строке? Если обе транзакции читающие, сложностей нет. Если обе пишущие — тоже (в этом случае они выстраиваются в очередь и выполняют изменения друг за другом). Самый интересный вариант — как взаимодействуют пишущая и читающая транзакции.

Простых пути два. Такие транзакции могут блокировать друг друга — но тогда страдает производительность. Либо читающая транзакция сразу может видеть изменения, сделанные пишущей транзакцией, даже если они не зафиксированы (это называется «грязным чтением») — но это очень плохо, ведь изменения могут быть отменены.

PostgreSQL идет сложным путем и использует *многоверсионность* — хранит несколько версий одной и той же строки. При этом пишущая транзакция работает со своей версией, а читающая видит свою.

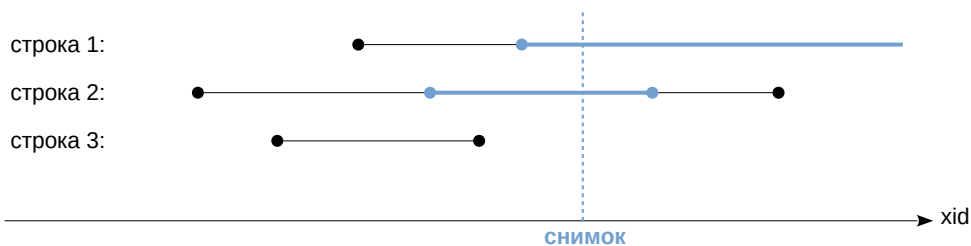
Версии надо как-то отличать друг от друга. Для этого каждая из них хранит две отметки, определяющие «время действия» данной версии.

В качестве времени здесь используются всегда возрастающие номера транзакций (в действительности все немного сложнее, но это детали). Когда строка создается, она помечается номером транзакции, выполнившей команду INSERT. Когда удаляется — версия помечается номером транзакции, выполнившей DELETE (но физически не удаляется). UPDATE состоит из двух операций DELETE и INSERT.

<https://postgrespro.ru/docs/postgresql/16/mvcc-intro>

## Согласованный срез на определенный момент времени

номер транзакции — определяет момент времени  
список активных транзакций — чтобы не смотреть  
на еще не зафиксированные изменения



4

В PostgreSQL применяется *изоляция на основе снимков данных*.

Транзакция, обращаясь к таблице, должна видеть только одну из имеющихся версий каждой строки (или не видеть ни одной). Для этого транзакция работает со *снимком данных*, созданным в определенный момент времени. В снимке видны самые последние версии уже зафиксированных данных, а еще не зафиксированные данные не видны. Иными словами, от каждой строки в снимок попадает версия, соответствующая моменту создания снимка.

Снимок — это не физическая копия данных, а всего несколько чисел:

- номер последней зафиксированной транзакции на момент создания снимка;
- список активных транзакций на этот момент.

Список нужен для того, чтобы исключить из снимка изменения тех транзакций, которые начались до создания снимка, но еще не были зафиксированы.

Зная эти числа, мы всегда можем сказать, какая из версий строки будет видна в снимке. Иногда это будет актуальная (самая последняя зафиксированная) версия, как для строки 1 на иллюстрации. Иногда не самая последняя: строка 2 удалена (и изменение уже зафиксировано), но транзакция еще продолжает видеть эту строку, пока работает со своим снимком. Это правильное поведение — оно дает согласованную картину данных на выбранный момент времени.

Какие-то строки вовсе не попадут в снимок: строка 3 удалена до того, как был построен снимок, поэтому в снимке ее нет.

## Read Uncommitted

не поддерживается PostgreSQL: работает как Read Committed

## Read Committed — *используется по умолчанию*

снимок строится на момент начала оператора

одинаковые запросы могут каждый раз получать разные данные

## Repeatable Read

снимок строится на момент начала первого оператора транзакции

транзакция может завершиться ошибкой сериализации

## Serializable

полная изоляция, но дополнительные накладные расходы

транзакция может завершиться ошибкой сериализации

Стандарт SQL определяет четыре уровня изоляции: чем строже уровень, тем меньше влияния оказывают параллельно работающие транзакции друг на друга. Во времена, когда стандарт принимался, считалось, что чем строже уровень, тем сложнее его реализовать и тем сильнее его влияние на производительность (с тех пор эти представления несколько изменились).

Самый нестрогий уровень **Read Uncommitted** допускает грязные чтения. Он не поддерживается PostgreSQL, поскольку не представляет практической ценности и не дает выигрыша в производительности.

Уровень **Read Committed** является уровнем изоляции по умолчанию в PostgreSQL. На этом уровне снимки данных строятся в начале выполнения каждого оператора SQL. Таким образом, оператор работает с неизменной и согласованной картиной данных, но два одинаковых запроса, следующих один за другим, могут показать разные данные.

На уровне **Repeatable Read** снимок строится в начале транзакции (при выполнении первого оператора) — поэтому все запросы в одной транзакции видят одни и те же данные. Этот уровень удобен, например, для отчетов, состоящих из нескольких запросов.

Уровень **Serializable** гарантирует полную изоляцию: можно писать операторы так, как будто транзакция работает одна. Но при этом некоторая доля транзакций будет прерываться; приложение должно уметь повторять такие транзакции.

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

## Видимость версий строк

Как убедиться в том, что одна и та же строка может существовать в нескольких версиях?

Создадим таблицу:

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

```
CREATE TABLE
```

И вставим одну строку. Напомним, что если не начать транзакцию явно командой BEGIN, psql выполняет команду и немедленно фиксирует результат:

```
=> INSERT INTO t VALUES ('Первая версия');
```

```
INSERT 0 1
```

Начнем транзакцию и выведем ее номер:

```
=> BEGIN;
```

```
BEGIN
```

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

```
pg_current_xact_id
-----
736
(1 row)
```

Транзакция видит первую (и пока единственную) версию строки:

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

```
      s      | xmin | xmax
-----+-----+-----
Первая версия | 735  |    0
(1 row)
```

Здесь скрытые столбцы показывают номера транзакций, ограничивающих видимость версии строки: xmin — номер предыдущей транзакции, которая создала версию, а xmax=0 означает, что эта версия актуальна.

Теперь начнем другую транзакцию в другом сеансе:

```
| => BEGIN;
|
| BEGIN
|
| => SELECT pg_current_xact_id();
|
| pg_current_xact_id
| -----
| 737
| (1 row)
```

Транзакция видит ту же единственную версию:

```
| => SELECT *, xmin, xmax FROM t;
|
|      s      | xmin | xmax
| -----+-----+-----
| Первая версия | 735  |    0
| (1 row)
```

Теперь изменим строку во второй транзакции.

```
| => UPDATE t SET s = 'Вторая версия';
|
| UPDATE 1
```

Вот что получилось:

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

s	xmin	xmax
Вторая версия (1 row)	737	0

А что увидит первая транзакция?

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

s	xmin	xmax
Первая версия (1 row)	735	737

Поскольку изменение не зафиксировано, первая транзакция продолжает видеть первую версию строки.

Обратите внимание на xmax — значение показывает, что в настоящий момент другая транзакция меняет строку. Вообще говоря, такое «подглядывание» нарушает изоляцию, поэтому поля xmin и xmax скрыты и в реальной работе их использовать не стоит.

Теперь зафиксируем изменения.

```
=> COMMIT;
COMMIT
```

Что теперь увидит первая транзакция?

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

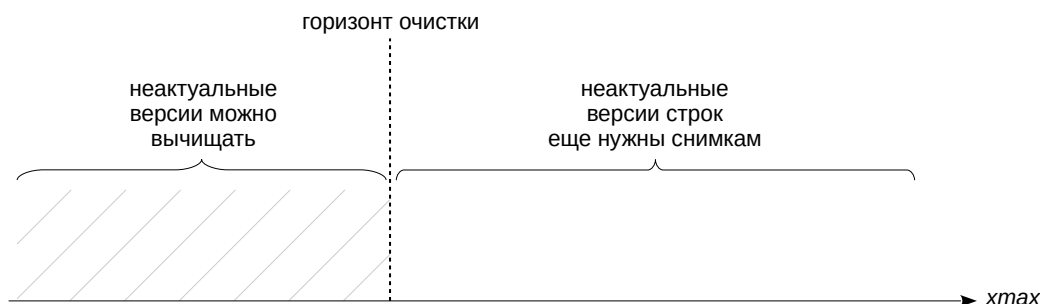
s	xmin	xmax
Вторая версия (1 row)	737	0

Теперь и первая транзакция видит вторую версию строки.

После фиксации первая версия строки больше не будет видна ни в одной транзакции.

```
=> COMMIT;
COMMIT
```

# Очистка и ее горизонт



Горизонт один на каждую базу данных

Долгие транзакции могут удерживать горизонт

мешая тем самым вычищать неактуальные версии строк

7

Механизм многоверсионности позволяет эффективно реализовать изоляцию на основе снимков, но в результате в табличных страницах накапливаются старые («мертвые») версии строк.

Какое-то время исторические версии нужны транзакциям, чтобы они могли работать со своим снимком данных, а в случае отката — вернуться к старым значениям. При этом для каждой базы данных существует такой номер `xid`, что все исторические версии, удаленные транзакциями с меньшими номерами, уже не видны ни в одном снимке. Этот номер называется *горизонтом очистки*.

Очистка производится группой специальных фоновых процессов; можно также выполнить ее вручную командой `VACUUM`.

Долгие транзакции и долго выполняющиеся запросы могут удерживать горизонт очистки, что не дает удалять накопившиеся версии строк. Если своевременно не вычищать исторические данные, таблицы и индексы будут разрастаться, занимая лишнее место на диске, а поиск в них актуальных версий строк будет замедляться.

<https://postgrespro.ru/docs/postgresql/16/routine-vacuuming>



## Блокировки строк

чтение никогда не блокирует строки  
изменение строк блокирует их для изменений, но не для чтений

## Блокировки таблиц

запрещают изменение или удаление таблицы, пока с ней идет работа  
запрещают чтение таблицы при перестроении или перемещении  
и т. п.

## Время жизни блокировок

устанавливаются по мере необходимости или вручную  
снимаются автоматически при завершении транзакции или при откате  
к точке сохранения

Что же дает многоверсионность? Она позволяет обойтись необходимым минимумом блокировок, тем самым увеличивая производительность системы.

Основные блокировки устанавливаются на уровне строк. При этом чтение никогда не блокирует ни читающие, ни пишущие транзакции. Изменение строки не блокирует ее чтение. Единственный случай, когда транзакция будет ждать освобождения блокировки — если она пытается менять строку, которая уже изменена другой, еще не зафиксированной транзакцией.

Блокировки также устанавливаются на более высоком уровне, в частности, на таблицах. Они нужны для того, чтобы никто не смог удалить таблицу, пока другие транзакции читают из нее данные, или чтобы запретить доступ к перестраиваемой таблице. На практике удаление и перестроение таблиц выполняются редко, поэтому обычно не вызывают проблем. Однако нужно учитывать, что перестроение таблицы происходит неявно при некоторых изменениях ее структуры и полностью блокирует доступ к таблице и ее индексам. Подробно блокировки рассматриваются в модуле «Блокировки» курса DBA2.

Все необходимые блокировки устанавливаются автоматически и автоматически же снимаются при завершении транзакции. Но если блокировка была получена после установки точки сохранения, она будет снята немедленно в случае отката к этой точке.

Можно также установить и дополнительные пользовательские блокировки; необходимость в этом возникает не часто.

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

## Блокировки

Повторим наш опыт, но теперь пусть обе транзакции попытаются изменить одну и ту же строку.

```
=> BEGIN;
```

```
BEGIN
```

```
=> UPDATE t SET s = 'Третья версия' RETURNING *;
```

```
      s
-----
Третья версия
(1 row)
```

```
UPDATE 1
```

И во второй транзакции:

```
| => BEGIN;
```

```
| BEGIN
```

```
| => UPDATE t SET s = 'Четвертая версия' RETURNING *;
```

Вторая транзакция «повисла»: она не может изменить строку, пока первая транзакция не снимет блокировку.

```
=> COMMIT;
```

```
COMMIT
```

Теперь вторая транзакция может продолжить выполнение:

```
|      s
| -----
| Четвертая версия
| (1 row)
```

```
| UPDATE 1
```

```
| => COMMIT;
```

```
| COMMIT
```

Обе транзакции зафиксировали свои изменения. Первый сеанс снова читает таблицу и видит актуальную строку — это результат, зафиксированный второй транзакцией:

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

```
      s
-----
Четвертая версия
(1 row)
```

## Статус транзакций (clog)

служебная информация; два бита на транзакцию  
специальные файлы на диске  
буферы в общей памяти

## Фиксация

устанавливается бит «транзакция зафиксирована»

## Обрыв

устанавливается бит «транзакция прервана»  
выполняется так же быстро, как и фиксация (не нужен откат данных)

Для работы многоверсионности серверу надо понимать, в каком статусе находятся транзакции. Транзакция может быть активна или завершена. Завершиться транзакция может либо фиксацией, либо обрывом. Таким образом, для представления состояния каждой транзакции требуются два бита.

Статусы транзакций (commit log, clog) хранятся в специальных служебных файлах в каталоге PGDATA/pg\_xact, а работа с ними происходит в общей памяти сервера, чтобы не приходилось постоянно обращаться к диску.

При любом завершении транзакции (как успешном, так и неуспешном) необходимо всего лишь установить соответствующие биты статуса. И фиксация, и обрыв транзакций происходят одинаково быстро.

Если прерванная транзакция успела создать новые версии строк, эти версии не уничтожаются (не происходит «физического» отката данных). Благодаря информации о статусах другие транзакции увидят, что транзакция, создавшая или удалившая версии строк, была прервана и не станут принимать во внимание сделанные ей изменения.

В файлах данных могут храниться несколько версий каждой строки

Транзакции работают со снимком данных — согласованным срезом на определенный момент времени

Уровни изоляции отличаются временем создания снимка

Мертвые версии строк за горизонтом очистки должны периодически вычищаться

Писатели не блокируют читателей,  
читатели не блокируют никого

1. Создайте таблицу с одной строкой.

Начните первую транзакцию на уровне изоляции Read Committed и выполните запрос к таблице.

Во втором сеансе удалите строку и зафиксируйте изменения. Сколько строк увидит первая транзакция, выполнив тот же запрос повторно? Проверьте.

Завершите первую транзакцию.

2. Повторите то же самое, но пусть теперь транзакция работает на уровне изоляции Repeatable Read:

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
```

Объясните отличия.

## 1. Уровень изоляции Read Committed

Создаем таблицу:

```
=> CREATE TABLE t(n integer);
```

```
CREATE TABLE
```

```
=> INSERT INTO t VALUES (42);
```

```
INSERT 0 1
```

Запрос из первой транзакции (по умолчанию используется уровень изоляции Read Committed):

```
=> BEGIN;
```

```
BEGIN
```

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

```
 n
----
 42
(1 row)
```

Удаляем строку во второй транзакции и фиксируем изменения:

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

```
| DELETE 1
```

Повторим запрос:

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

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

Первая транзакция видит произошедшие изменения: строка удалена.

```
=> COMMIT;
```

```
COMMIT
```

## 2. Уровень изоляции Repeatable Read

Вернем строку:

```
=> INSERT INTO t VALUES (42);
```

```
INSERT 0 1
```

Запрос из первой транзакции:

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

```
BEGIN
```

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

```
 n
----
 42
(1 row)
```

Удаляем строку во второй транзакции и фиксируем изменения:

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

```
| DELETE 1
```

Повторим запрос:

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

```
n
----
42
(1 row)
```

На этом уровне изоляции первая транзакция не видит изменений: для нее строка по-прежнему существует.

=> **COMMIT;**

COMMIT

1. Начните транзакцию и создайте новую таблицу с одной строкой. Не завершая транзакцию, откройте второй сеанс и выполните в нем запрос к таблице. Проверьте, что увидит транзакция во втором сеансе.  
Зафиксируйте транзакцию в первом сеансе и повторите запрос к таблице во втором сеансе.
2. Повторите задание 1, но откатите, а не зафиксируйте транзакцию в первом сеансе. Что изменилось?
3. В первом сеансе начните транзакцию и выполните запрос к созданной ранее таблице. Получится ли удалить эту таблицу во втором сеансе, пока первая транзакция не завершена? Проверьте.



## 1. Транзакции и команды DDL — фиксация

Начнем транзакцию и создадим новую таблицу:

```
=> BEGIN;
```

```
BEGIN
```

```
=> CREATE TABLE t1(n integer);
```

```
CREATE TABLE
```

```
=> INSERT INTO t1 VALUES (42);
```

```
INSERT 0 1
```

Во втором сеансе сделаем запрос к таблице:

```
| => SELECT * FROM t1;
```

```
| ERROR:  relation "t1" does not exist  
| LINE 1: SELECT * FROM t1;  
|                      ^
```

Пока создавшая таблицу транзакция не завершена, все остальные транзакции таблицу не видят.

Таблица будет видна только после завершения создавшей ее транзакции:

```
=> COMMIT;
```

```
COMMIT
```

```
| => SELECT * FROM t1;
```

```
|      n  
|      ----  
|      42  
| (1 row)
```

## 2. Транзакции и команды DDL — откат

Снова начнем транзакцию и создадим новую таблицу:

```
=> BEGIN;
```

```
BEGIN
```

```
=> CREATE TABLE t2(n integer);
```

```
CREATE TABLE
```

```
=> INSERT INTO t2 VALUES (42);
```

```
INSERT 0 1
```

Запрос во втором сеансе ожидаемо не видит изменений:

```
| => SELECT * FROM t2;
```

```
| ERROR:  relation "t2" does not exist  
| LINE 1: SELECT * FROM t2;  
|                      ^
```

При откате первой транзакции команда создания таблицы тоже откатывается:

```
=> ROLLBACK;
```

```
ROLLBACK
```

```
=> SELECT * FROM t2;
```

```
ERROR:  relation "t2" does not exist  
LINE 1: SELECT * FROM t2;  
                      ^
```

Команды DDL в PostgreSQL являются транзакционными.

## 3. Блокировки таблиц

Начав транзакцию, обратимся к таблице:

```
=> BEGIN;
```

```
BEGIN
```

```
=> SELECT * FROM t1;
```

```
  n  
----  
 42  
(1 row)
```

При попытке удалить таблицу вторая транзакция будет заблокирована — нельзя удалить таблицу, которая используется.

```
| => DROP TABLE t1;
```

Таблица будет удалена только после завершения первой транзакции:

```
=> COMMIT;
```

```
COMMIT
```

```
| DROP TABLE
```