

# Архитектура 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.

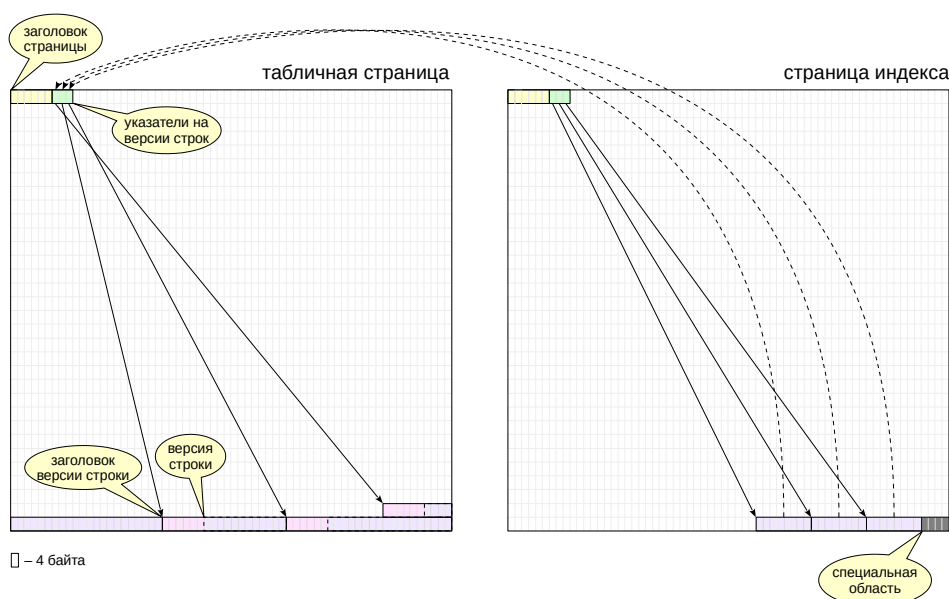
Страницы

Версии строк

Снимки данных

Структура страниц

Формат данных



Размер страницы составляет 8 Кбайт. Это значение можно увеличить (вплоть до 32 Кбайт), но только при сборке. И таблицы, и индексы, и большинство других объектов, которые в PostgreSQL обозначаются термином *relation*, используют одинаковую структуру страниц, чтобы пользоваться общим буферным кешем. В начале страницы идет **заголовок** (24 байта), содержащий общие сведения о странице и размер ее областей: указателей, свободного пространства, версий строк и специальной области.

**Версии строк** содержит те самые данные, которые мы храним в таблицах и других объектах БД, плюс заголовок. «Версия строки» по-английски называется *tuple*; иногда мы будем говорить просто «строка».

**Указатели** имеют фиксированный размер (4 байта) и составляют массив, позиция в котором определяет идентификатор строки (*tuple id*, *tid*). Указатели ссылаются на версии строк (*tuple*), расположенные в конце блока. Такая косвенная адресация удобна тем, что во-первых, позволяет найти нужную строку, не перебирая все содержимое блока (строки имеют разную длину), а во-вторых, позволяет перемещать строку внутри блока, не ломая ссылки из индексов.

Между указателями и версиями строк находится **свободное место**.

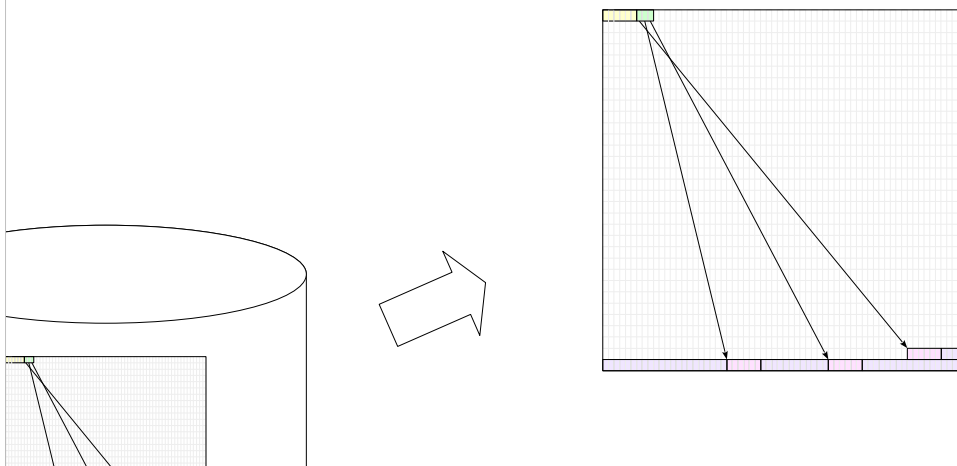
Для некоторых типов индексов нужно хранить служебную информацию; для этого может использоваться нулевая страница, а также **специальная область** в конце каждой страницы.

<https://postgrespro.ru/docs/postgresql/16/storage-page-layout>

## Страницы читаются в оперативную память «как есть»

данные не переносятся между разными платформами

между полями данных возможны пропуски из-за выравнивания



5

Формат данных на диске полностью совпадает с представлением данных в оперативной памяти. Страница читается в буферный кеш «как есть», без преобразований.

Поэтому файлы данных на одной платформе (разрядность, порядок байтов и т. п.) оказываются несовместимыми с другими платформами.

Кроме того, многие архитектуры предусматривают выравнивание данных по границам машинных слов. Например, на 32-битной системе x86 целые числа (тип `integer`, занимает 4 байта) будут выровнены по границе 4-байтных слов, как и числа с плавающей точкой двойной точности (тип `double precision`, 8 байт). А в 64-битной системе значения `double precision` будут выровнены по границе 8-байтных слов.

Из-за этого размер табличной строки зависит от порядка расположения полей. Обычно этот эффект не сильно заметен, но в некоторых случаях он может привести к существенному увеличению размера. Например, если располагать поля типов `char(1)` и `integer` попеременно, между ними, как правило, будет пропадать 3 байта.

<https://pgconf.ru/media/2016/05/13/tuple-internals-ru.pdf>

## Подготовка

```
=> CREATE DATABASE arch_mvcc;
```

```
CREATE DATABASE
```

```
=> \c arch_mvcc
```

You are now connected to database "arch\_mvcc" as user "student".

Создадим таблицу и индекс по одному из полей.

```
=> CREATE TABLE t(  
    id integer,  
    s text  
);
```

```
CREATE TABLE
```

```
=> CREATE INDEX t_s on t(s);
```

```
CREATE INDEX
```

```
=> INSERT INTO t(id, s) VALUES (42, 'FOO');
```

```
INSERT 0 1
```

---

## Структура страницы

Чтобы изучить структуру страницы и версий строк, воспользуемся расширением pageinspect.

```
=> CREATE EXTENSION pageinspect;
```

```
CREATE EXTENSION
```

Посмотрим на поля заголовка страницы, определяющие границы ее областей.

```
=> SELECT lower, upper, special, pagesize  
FROM page_header(get_raw_page('t',0));
```

```
 lower | upper | special | pagesize  
-----+-----+-----+-----  
    28 | 8160 |    8192 |    8192  
(1 row)
```

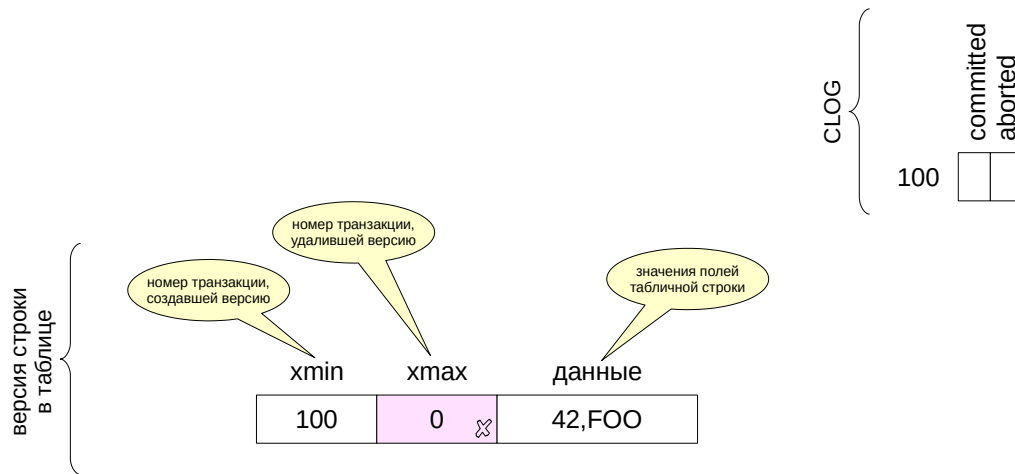
Области занимают следующие диапазоны адресов:

- 0 — начало заголовка страницы и указатели на версии строк,
- lower — начало свободного места,
- upper — начало данных (версий строк),
- special — начало спец. данных (только для индексов),
- pagesize — конец страницы.

Структура версий строк

Как работают операции над данными

NOT-обновления



Рассмотрим, как устроены *версии строк* (tuples) и как выполняются операции со строками на низком уровне. Начнем со вставки.

В нашем примере — таблица с двумя столбцами (число и текст). При вставке строки в *табличной странице* (heap page) появится указатель с номером 1, ссылающийся на первую и единственную версию строки. Каждый такой указатель помимо ссылки содержит длину версии строки и несколько бит, определяющих ее статус.

Версии строк кроме собственно данных имеют также заголовок, занимающий минимум 23 байта. Помимо прочего он содержит:

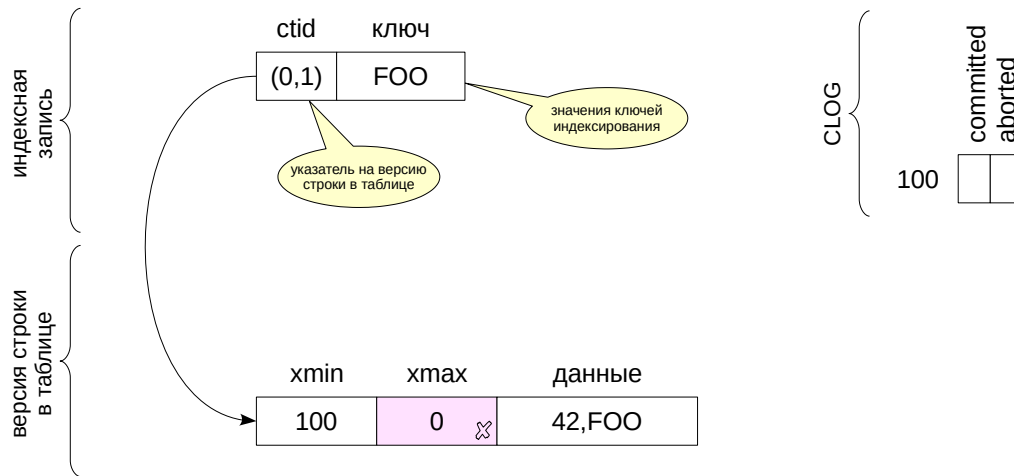
- **xmin** и **xmax** — поля, определяющие видимость данной версии строки в терминах начального и конечного номеров транзакций;
- карту неопределенных значений (в ней отмечены поля со значением NULL);
- ряд признаков, показывающих, например, статус транзакций xmin и xmax, если он уже известен (зафиксированы или оборваны).

В нашей версии строки поле xmin заполнено номером текущей транзакции (100). Поскольку изменения еще не фиксировались и транзакция активна, то в *журнале статуса транзакций* (CLOG) соответствующая запись заполнена нулями. CLOG можно представить себе как массив, в котором для каждой транзакции (начиная с некоторой) отводится ровно два бита.

Поле xmax заполнено фиктивным номером 0. Транзакции не будут обращать внимание на этот номер, поскольку установлен признак, что эта транзакция оборвана.



# Вставка



индекс не содержит информации о версии строки

Пусть также имеется индекс В-дерево, созданный по текстовому полю. Информация в индексной странице сильно зависит от типа индекса. И даже у одного типа индекса бывают разные виды страниц. Например, у В-дерева есть страница с метаданными, страницы для листовых и промежуточных узлов.

Тем не менее, обычно в странице имеется массив указателей и индексные записи (аналогичные версиям строк в табличной странице).

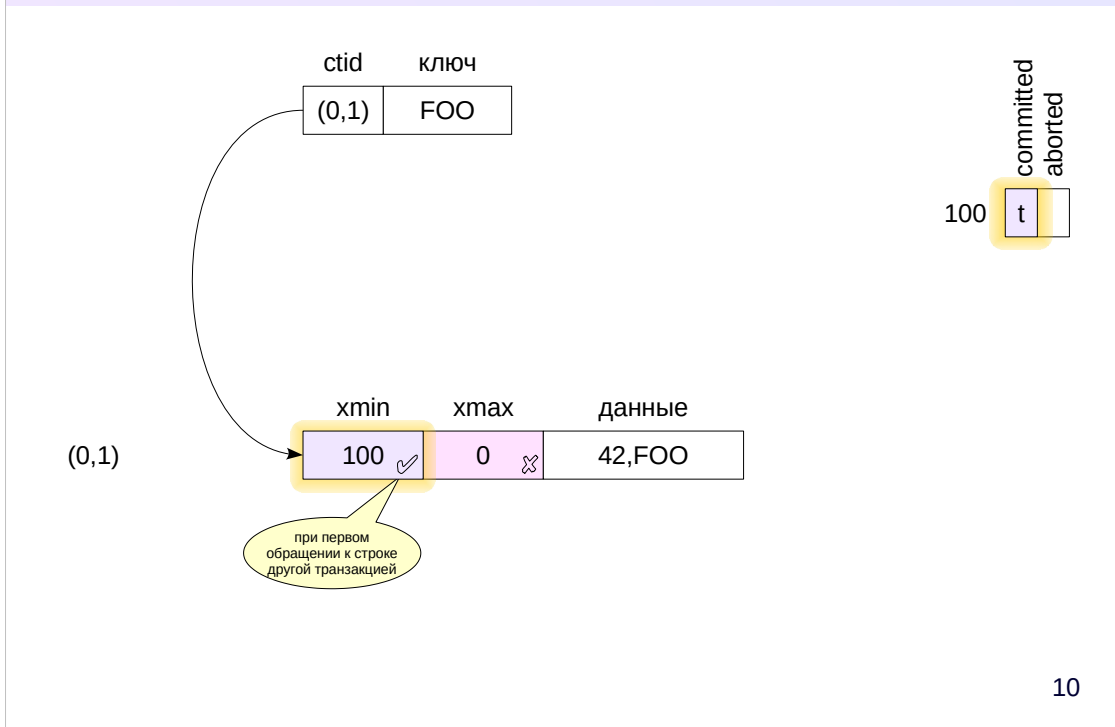
Индексные записи тоже могут иметь очень разную структуру в зависимости от типа индекса. Например, для В-дерева записи в листовых страницах содержат значение ключа индексирования и ссылку (`ctid`) на соответствующую строку таблицы (структура В-дерева разбирается в теме «Классы операторов» и в учебном курсе QPT «Оптимизация запросов»).

В общем случае индекс может быть устроен совсем другим образом, но как правило он все равно будет содержать ссылки на версии строк.

В нашем примере в индексной странице также создается указатель с номером 1, который ссылается на индексную запись, которая, в свою очередь, ссылается на первую строку в табличной странице. Чтобы не загромождать рисунок, указатель и строка таблицы объединены.

Важный момент состоит в том, что никакой индекс не содержит информацию о версии строки (нет полей `xmin` и `xmax`). Прочитав индексную запись, невозможно определить ее видимость, не заглянув в табличную страницу (для оптимизации служит карта видимости).

# Фиксация изменений



Когда транзакция фиксируется, в CLOG для этой транзакции выставляется признак committed. Это, по сути, единственная операция (не считая журнала предзаписи), которая необходима.

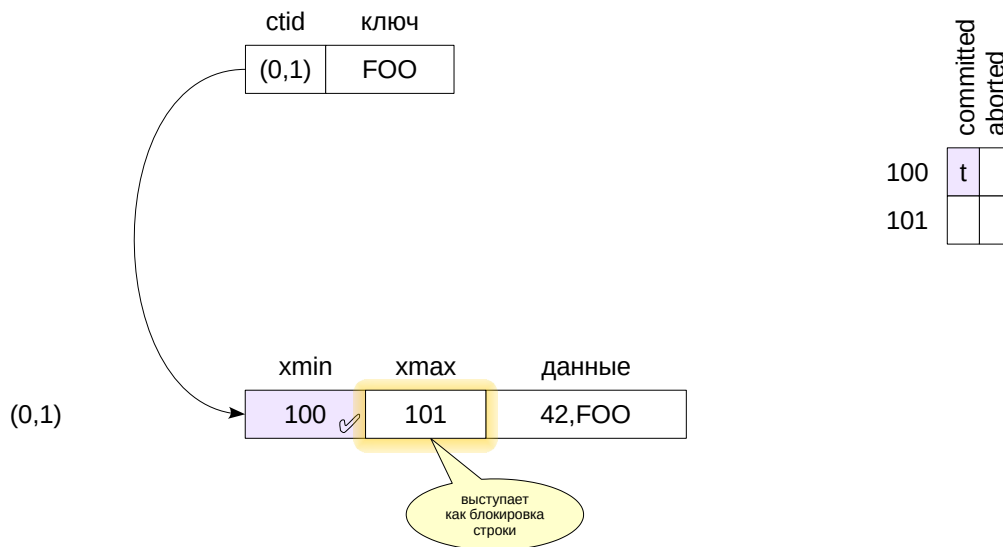
Когда какая-либо другая транзакция обратится к версии строки, ей придется ответить на вопросы:

- 1) завершилась ли транзакция 100 (надо проверить список активных процессов и их транзакций; такая структура в общей памяти имеет название ProcArray),
- 2) а если завершилась, то фиксацией или отменой (свериться с CLOG).

Поскольку выполнять проверку по CLOG каждый раз накладно, выясненный однажды статус транзакции записывается в информационные биты-подсказки заголовка строки. Если один из этих битов установлен, то состояние транзакции xmin считается известным и следующей транзакции уже не придется обращаться к CLOG.

Почему эти биты не устанавливаются той транзакцией, которая выполняла вставку? В момент, когда транзакция фиксируется или отменяется, уже непонятно, какие именно строки в каких именно страницах транзакция успела поменять. Кроме того, часть этих страниц может быть вытеснена из буферного кеша на диск; читать их заново, чтобы изменить биты, означало бы существенно замедлить фиксацию.

Обратная сторона состоит в том, что любая транзакция (даже выполняющая простое чтение — SELECT) может загрязнить данные в буферном кеше и породить новые журнальные записи.

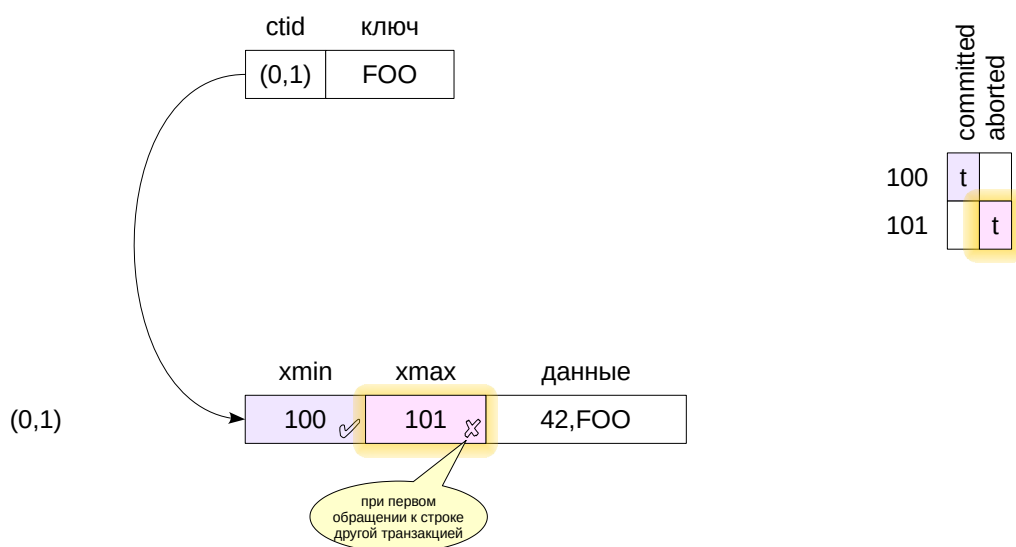


При удалении строки в поле xmax текущей версии записывается номер текущей удаляющей транзакции, а признак оборванной транзакции сбрасывается. Больше ничего не происходит.

Заметим, что установленное значение xmax, соответствующее активной транзакции (что определяется другими транзакциями по ProcArray), выступает в качестве блокировки. Если другая транзакция намерена обновить или удалить эту строку, она будет вынуждена дожидаться завершения транзакции xmax.

Подробнее блокировки рассматриваются в теме «Блокировки» этого модуля. Пока отметим только, что число блокировок строк ничем не ограничено. Они не занимают место в оперативной памяти, производительность системы не страдает от их количества (разумеется, за исключением того, что первый процесс, обратившийся к версии строки, должен будет проставить биты-подсказки).

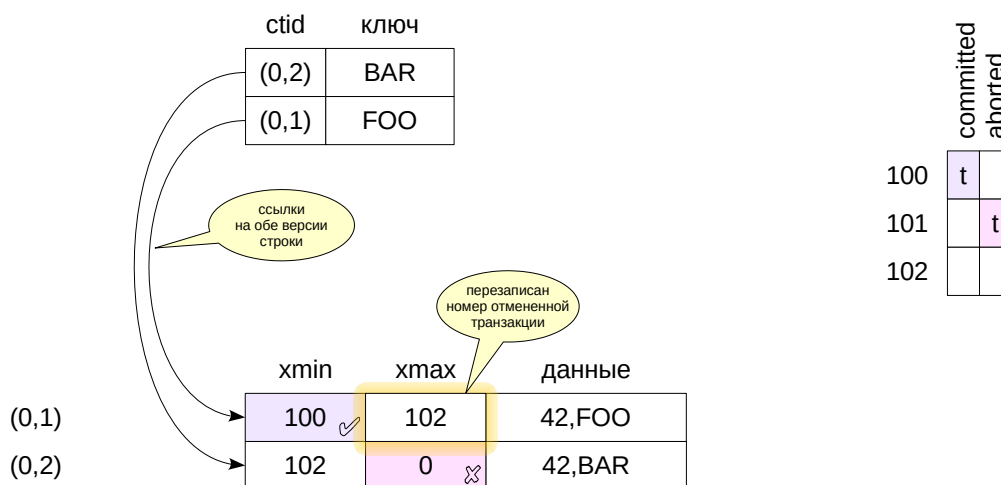
# Отмена изменений



12

Отмена изменений работает аналогично фиксации, только в CLOG для транзакции выставляется бит оборванной транзакции. Отмена выполняется так же быстро, как и фиксация — не требуется выполнять откат выполненных действий.

Номер прерванной транзакции остается в поле xmax. Когда другая транзакция обратится к версии строки, она проверит статус транзакции 101 и установит в версию строки признак-подсказку, что транзакция оборвана. Это будет означать, что на значение в поле xmax смотреть не нужно.



при любом обновлении строки надо изменять все индексы на таблице  
в индексе накапливаются ссылки на неактуальные версии

13

Обновление работает так, как будто сначала выполняется удаление старой версии строки, а затем — вставка новой.

Старая версия помечается номером текущей транзакции в поле xmax. Обратите внимание, что новое значение 102 записалось поверх старого 101, так как транзакция 101 была отменена. Кроме того, биты-подсказки сброшены, так как статус текущей транзакции еще не известен.

В индексной странице появляется второй указатель и вторая запись, ссылающаяся на вторую версию в табличной странице.

Так же, как и при удалении, значение xmax в первой версии строки служит признаком того, что строка заблокирована.

Чем плохо такое обновление?

Во-первых, при любом изменении строки приходится обновлять все индексы, созданные для таблицы (даже если измененные поля не входят в индекс). Очевидно, это снижает производительность.

Во-вторых, в индексах накапливаются ссылки на исторические версии строки, которые потом приходится очищать.

Более того, есть особенность реализации B-дерева в PostgreSQL. Если на индексной странице недостаточно места для вставки новой записи, страница делится на две и все данные перераспределяются между ними. Однако при удалении (или очистке) записей две индексные страницы не «склеиваются» в одну. Из-за этого размер индекса может не уменьшиться даже при удалении существенной части данных.

## Версии строк

Начнем с чистого листа.

```
=> TRUNCATE TABLE t;
```

TRUNCATE TABLE

Вставим одну строку, предварительно начав транзакцию.

```
=> BEGIN;
```

BEGIN

```
=> INSERT INTO t(id, s) VALUES (42, 'FOO');
```

INSERT 0 1

Номер нашей текущей транзакции:

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

```
pg_current_xact_id
-----
                810
(1 row)
```

Посмотрим на таблицу, добавив системные столбцы:

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

```
 ctid | xmin | xmax | id | s
-----+-----+-----+----+--
(0,1) | 810 |    0 | 42 | F00
(1 row)
```

Xmin совпадает с номером нашей транзакции, а xmax равен фиктивному значению 0.

Чтобы заглянуть в индекс, воспользуемся расширением pageinspect. Нулевая страница индекса содержит метainформацию, поэтому смотрим в первую.

```
=> SELECT itemoffset, ctid, data FROM bt_page_items('t_s',1);
```

```
itemoffset | ctid | data
-----+-----+-----
          1 | (0,1) | 09 46 4f 4f 00 00 00 00
(1 row)
```

Видим один указатель на единственную строку таблицы. (В поле data можно угадать ASCII-коды букв FOO.)

Pageinspect позволяет увидеть и полную информацию о том, что находится в табличной странице, но в этой теме мы не будем углубляться в эти детали.

Зафиксируем изменение.

```
=> COMMIT;
```

COMMIT

Теперь проверим, как работает обновление.

Начнем второй сеанс.

```
student$ psql arch_mvcc
```

В нем прочитаем таблицу на уровне Repeatable Read:

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

```
| BEGIN
```

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

```
|  ctid | xmin | xmax | id | s
|  -----+-----+-----+----+--
|  (0,1) | 810 |    0 | 42 | F00
|  (1 row)
```

Теперь обновляем строку.

```
=> BEGIN;
```

```
BEGIN
```

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

```
pg_current_xact_id
-----
                811
(1 row)
```

```
=> UPDATE t SET s = 'BAR';
```

```
UPDATE 1
```

Запрос в первом сеансе выдает одну строку (новую версию):

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

```
 ctid | xmin | xmax | id | s
-----+-----+-----+----+--
(0,2) | 811 |    0 | 42 | BAR
(1 row)
```

Но в странице присутствуют обе версии. Предыдущую продолжает видеть второй сеанс:

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

```
|  ctid | xmin | xmax | id | s
|  -----+-----+-----+----+--
|  (0,1) | 810 |  811 | 42 | FOO
|  (1 row)
```

Номер транзакции, удалившей первую версию строки, записался в xmax.

При этом в индексной странице обнаруживаем указатели на обе версии:

```
=> SELECT itemoffset, ctid, data FROM bt_page_items('t_s',1);
```

```
itemoffset | ctid | data
-----+-----+-----
          1 | (0,2) | 09 42 41 52 00 00 00 00
          2 | (0,1) | 09 46 4f 4f 00 00 00 00
(2 rows)
```

Индексные записи внутри страницы упорядочены. Поэтому первой идет запись с ключом BAR (ссылается на версию (0,2)), а второй — запись с ключом FOO (ссылается на версию (0,1)).

Завершим транзакции.

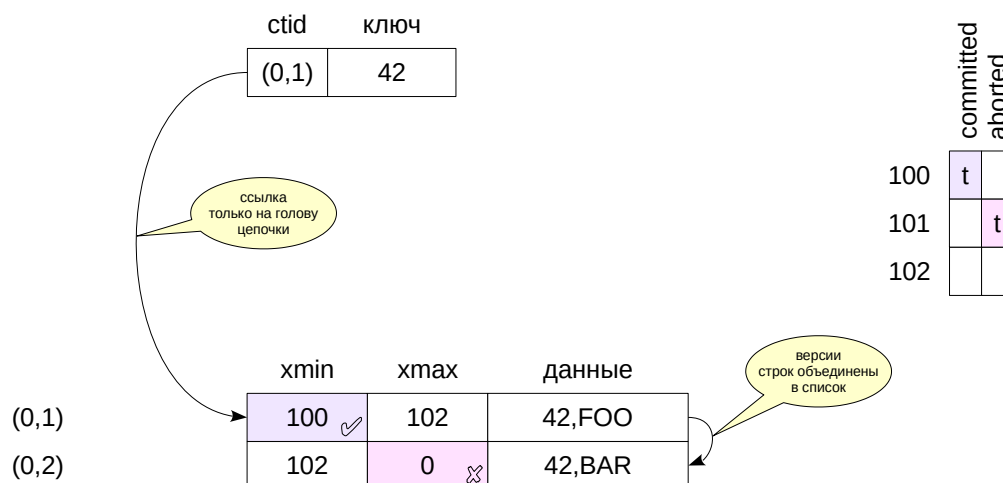
```
=> COMMIT;
```

```
COMMIT
```

```
| => COMMIT;
```

```
| COMMIT
```

# НОТ-обновление



обновляемые столбцы не должны входить ни в один индекс  
цепочка обновлений — только в пределах одной страницы

15

Однако если индекс создан по полю, значение которого не изменилось в результате обновления строки, то нет смысла создавать дополнительную запись в В-дереве, содержащую то же самое значение ключа. Именно так работает оптимизация, называемая НОТ-обновлением — Heap-Only Tuple Update.

При таком обновлении в индексной странице находится лишь одна запись, ссылающаяся на первую версию строки табличной страницы. А внутри табличной страницы организуется цепочка из версий, на которые гарантированно нет внешних ссылок (heap-only tuples).

Если при сканировании индекса PostgreSQL попадает в табличную страницу и обнаруживает версию, начинающую цепочку, он проходит по всей цепочке обновлений (для всех полученных таким образом версий строк проверяется видимость, прежде чем они будут возвращены).

Подчеркнем, что НОТ-обновления работают в случае, если обновляемые поля не входят *ни в один индекс* — иначе в каком-либо индексе оказалась бы ссылка непосредственно на новую версию строки, что противоречит идее этой оптимизации. В частности, НОТ-обновление используется, если у таблицы вообще нет индексов.

Оптимизация действует только в пределах одной страницы, поэтому дополнительный обход цепочки не требует обращения к другим страницам и не ухудшает производительность. Однако если на странице не хватит свободного места, чтобы разместить новую версию строки, цепочка прервется. На версию строки, размещенную на другой странице, будет сделана новая ссылка из индекса.



## НОТ-обновление

Снова очистим таблицу.

```
=> TRUNCATE TABLE t;
```

TRUNCATE TABLE

Но пусть теперь индекс будет построен по столбцу id:

```
=> DROP INDEX t_s;
```

DROP INDEX

```
=> CREATE INDEX t_id on t(id);
```

CREATE INDEX

```
=> INSERT INTO t(id, s) VALUES (42, 'FOO');
```

INSERT 0 1

Начнем транзакцию с уровнем изоляции Repeatable Read.

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

```
| BEGIN
```

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

```
|  ctid | xmin | xmax | id | s  
|-----+-----+-----+----+--  
| (0,1) | 815  |    0 | 42 | FOO  
| (1 row)
```

До этого мы обновляли проиндексированный столбец, а теперь столбец s не входит ни в один индекс.

```
=> BEGIN;
```

BEGIN

```
=> UPDATE t SET s = 'BAR';
```

UPDATE 1

Появляется вторая версия строки.

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

```
  ctid | xmin | xmax | id | s  
-----+-----+-----+----+--  
(0,2) | 816  |    0 | 42 | BAR  
(1 row)
```

Но другой сеанс продолжает видеть самую первую версию:

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

```
|  ctid | xmin | xmax | id | s  
|-----+-----+-----+----+--  
| (0,1) | 815  | 816  | 42 | FOO  
| (1 row)
```

Что же обнаружится в индексе?

```
=> SELECT itemoffset, ctid, data FROM bt_page_items('t_id',1);
```

```
itemoffset | ctid | data  
-----+-----+-----  
1 | (0,1) | 2a 00 00 00 00 00 00 00  
(1 row)
```

Мы видим, что в индексе не добавилась новая запись. Индекс продолжает ссылаться на первую версию, а уже внутри табличной страницы первая версия связана со второй в цепочку.

Еще один способ убедиться в том, что выполнилось НОТ-обновление — заглянуть в статистику текущей транзакции:

```
=> SELECT n_tup_upd, n_tup_hot_upd
FROM pg_stat_xact_all_tables
WHERE relid = 't'::regclass;
```

n_tup_upd	n_tup_hot_upd
1	1

(1 row)

Здесь мы видим, что всего транзакция обновила одну строку (n\_tup\_upd), причем с помощью HOT (n\_tup\_hot\_upd).

```
=> ROLLBACK;
```

```
ROLLBACK
```

```
| => ROLLBACK;
```

```
| ROLLBACK
```

Снимок данных

Видимость версий строк

Горизонт снимка

Горизонт транзакции

Горизонт базы данных

Дает согласованную картину данных на момент времени

видны только зафиксированные на этот момент данные

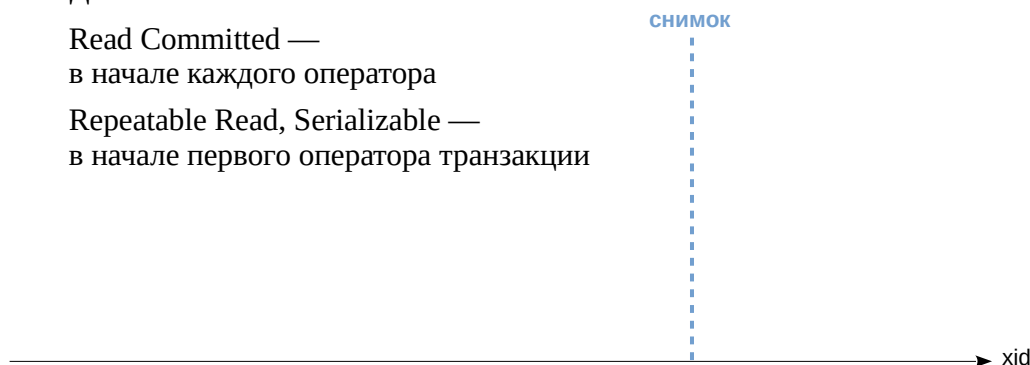
Создается:

Read Committed —

в начале каждого оператора

Repeatable Read, Serializable —

в начале первого оператора транзакции



Мы видели, что в страницах данных физически могут 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).

Согласованность здесь понимается в обычном ACID-смысле: данные должны быть корректны, что, по сути, означает, что видеть нужно только те данные, которые были зафиксированы на момент создания снимка. Это полностью исключает аномалию грязного чтения на любом уровне изоляции.

На уровне изоляции Read Committed снимок создается в начале каждого оператора транзакции. Снимок активен, пока выполняется оператор. Таким образом каждый оператор видит согласованные данные, но если между двумя операторами другие транзакции зафиксируют изменения, возможны неповторяемое и фантомное чтения.

На уровне Repeatable Read (и Serializable) снимок создается один раз в начале первого оператора транзакции. Такой снимок остается активным до самого конца транзакции. В этом случае неповторяемое и фантомное чтения исключаются.

Снимок данных не является физической копией версий строк, принадлежность версии строки к снимку определяется с помощью *правил видимости*.

Видимость версии строки ограничена  $x_{min}$  и  $x_{max}$

Версия попадает в снимок, когда

- изменения транзакции  $x_{min}$  видны для снимка,
- изменения транзакции  $x_{max}$  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 ( $cm_{in}/cm_{max}$ )

Будет или нет данная версия строки видна в снимке, определяется двумя полями ее заголовка —  $x_{min}$  и  $x_{max}$ , — 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  $x_{min}$ , и *не видны* изменения, сделанные транзакцией  $x_{max}$ . Иными словами, создание версии строки должно быть видно, а удаление (если оно было) — нет.

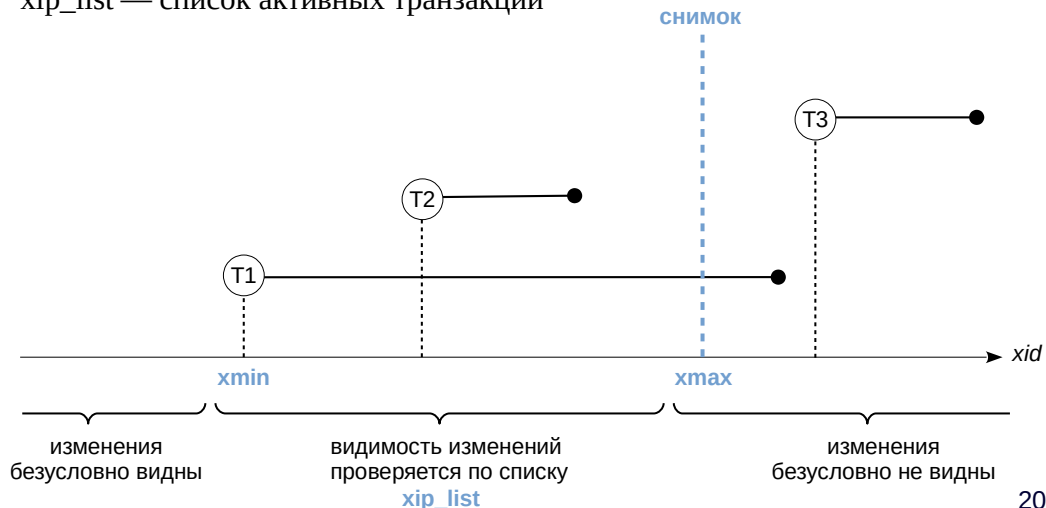
В свою очередь, изменения транзакции видны в снимке, если либо это та же самая транзакция, что создала снимок (транзакция видит свои же изменения), либо транзакция была зафиксирована до создания снимка.

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  $cm_{in}$  и  $cm_{max}$ ), показывающее порядковый номер операции внутри транзакции, и этот номер тоже принимается во внимание.

`xmin` — номер самой ранней активной транзакции

`xmax` — номер, следующий за последней зафиксированной транзакцией

`xip_list` — список активных транзакций



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.

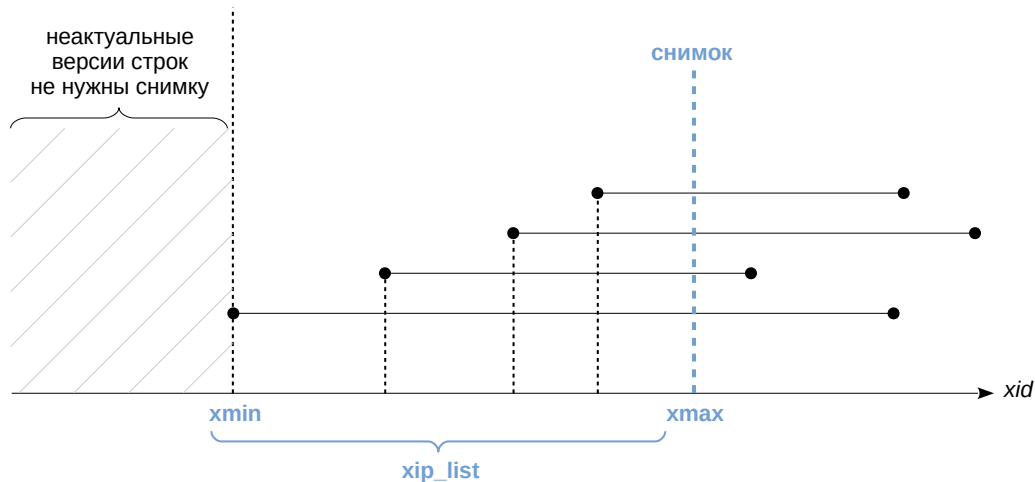
Все транзакции с номерами, большими или равными `xmax`, не существовали в момент создания снимка, поэтому изменения таких транзакций точно не видны.

- **`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:

<https://postgrespro.ru/docs/postgresql/16/functions-info>

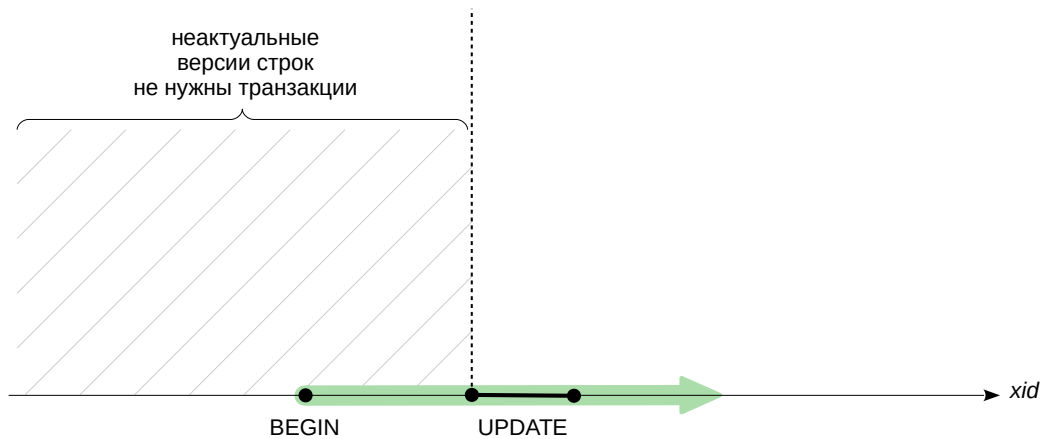


Номер самой ранней из активных транзакций ( $x_{min}$  снимка) имеет важный смысл — он определяет *горизонт снимка*.

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

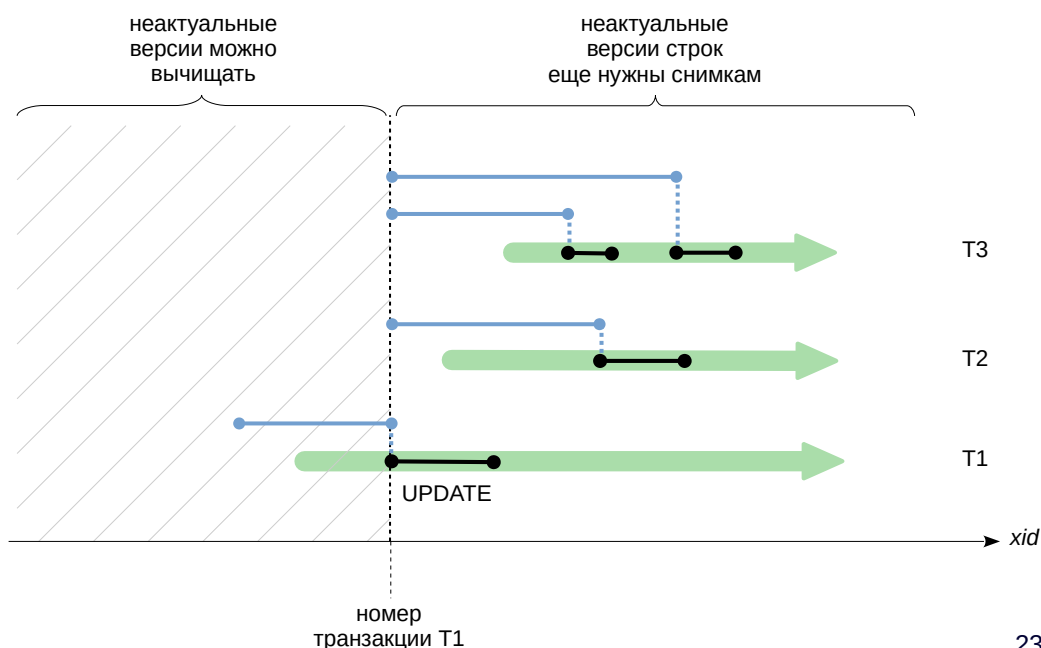


Если транзакция меняет данные, она держит горизонт на уровне своего номера на случай отката.

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



# Горизонт базы данных



Также можно определить и горизонт базы данных как минимальный из горизонтов всех транзакций и снимков. 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 обрабатывает, 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 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.

Это одна из причин, по которым транзакции не следует делать длиннее, чем абсолютно необходимо.

## Снимки данных

И еще раз начнем транзакцию в отдельном сеансе.

```
| => BEGIN ISOLATION LEVEL REPEATABLE READ;
|
| BEGIN
|
| => SELECT ctid, xmin, xmax, * FROM t;
|
|   ctid | xmin | xmax | id | s
|-----+-----+-----+----+--
|   (0,1) | 815 | 816 | 42 | F00
|   (1 row)
```

Одновременно с этим выполним несколько других транзакций.

```
=> UPDATE t SET s = 'BAR';

UPDATE 1

=> INSERT INTO t(id, s) VALUES (43, 'BAZ');

INSERT 0 1
```

Горизонт транзакций можно увидеть следующим образом:

```
=> SELECT query,backend_xmin
FROM pg_stat_activity
WHERE backend_xmin IS NOT NULL
AND datname='arch_mvcc';

      query                               | backend_xmin
-----+-----+-----
SELECT ctid, xmin, xmax, * FROM t; |          817
SELECT query,backend_xmin          +|          819
FROM pg_stat_activity              +|
WHERE backend_xmin IS NOT NULL      +|
AND datname='arch_mvcc';           |
(2 rows)
```

Видно, что открытая транзакция удерживает горизонт, поскольку ей нужны старые версии строк.

Завершим ее.

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

После этого горизонт базы данных продвигается вперед, позволяя очищать неактуальные версии:

```
=> SELECT query,backend_xmin
FROM pg_stat_activity
WHERE backend_xmin IS NOT NULL
AND datname='arch_mvcc';

      query                               | backend_xmin
-----+-----+-----
SELECT query,backend_xmin          +|          819
FROM pg_stat_activity              +|
WHERE backend_xmin IS NOT NULL+|
AND datname='arch_mvcc';           |
(1 row)
```

PostgreSQL использует многоверсионность и изоляцию на основе снимков данных

Индексы не содержат информации о версионности

чтобы вернуть данные, нужно проверить видимость  
(по таблице или карте видимости)

Лишние индексы не только влекут накладные расходы, но и препятствуют HOT-обновлениям

Транзакции должны быть насколько длинными, насколько необходимо, но не более

1. Функционал точек сохранения должен позволять откатывать *часть* транзакции. Чтобы разобраться, как это работает, выполните операции внутри одной транзакции:

- вставьте строку в таблицу;
- установите точку сохранения (SAVEPOINT);
- вставьте в таблицу еще одну строку;
- откатите изменения до точки сохранения;
- вставьте еще одну строку;
- зафиксируйте изменения.

При этом после каждой операции выводите номер текущей транзакции и проверяйте значения столбцов `xmin` и `xmax`.

## 1. Точки сохранения

```
=> CREATE DATABASE arch_mvcc;
```

```
CREATE DATABASE
```

```
=> \c arch_mvcc
```

You are now connected to database "arch\_mvcc" as user "student".

Создадим таблицу:

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

```
CREATE TABLE
```

Начинаем транзакцию.

```
=> BEGIN;
```

```
BEGIN
```

Вставляем первую строку:

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

```
INSERT 0 1
```

Вот номер нашей транзакции и состояние таблицы:

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

```
pg_current_xact_id
-----
                806
(1 row)
```

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

```
   s   | xmin | xmax
-----+-----+-----
first | 806  |    0
(1 row)
```

Пока все как обычно.

Устанавливаем точку сохранения и вставляем вторую строку:

```
=> SAVEPOINT sp;
```

```
SAVEPOINT
```

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

```
INSERT 0 1
```

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

```
pg_current_xact_id
-----
                806
(1 row)
```

Номер транзакции прежний.

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

```
   s     | xmin | xmax
-----+-----+-----
first  | 806  |    0
second | 807  |    0
(2 rows)
```

А вот в xmin второй строки указан другой номер. Это вложенная транзакция, она начинается сразу после установки точки сохранения.

Откатим изменения до точки сохранения и вставим третью строку:

```
=> ROLLBACK TO sp;
```

ROLLBACK

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

INSERT 0 1

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

pg_current_xact_id
806

(1 row)

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

s	xmin	xmax
first	806	0
third	808	0

(2 rows)

У третьей строки опять другой номер. Это еще одна вложенная транзакция, которая началась сразу после отката к точке сохранения.

Зафиксируем изменения:

```
=> COMMIT;
```

COMMIT

Каждая вложенная транзакция имеет собственный статус и может быть оборвана независимо от статуса основной транзакции. Откат части транзакции — это фактически откат одной или нескольких вложенных транзакций.

Но при завершении основной транзакции завершаются и все еще активные вложенные транзакции: при фиксации — фиксируются, при откате — откатываются.