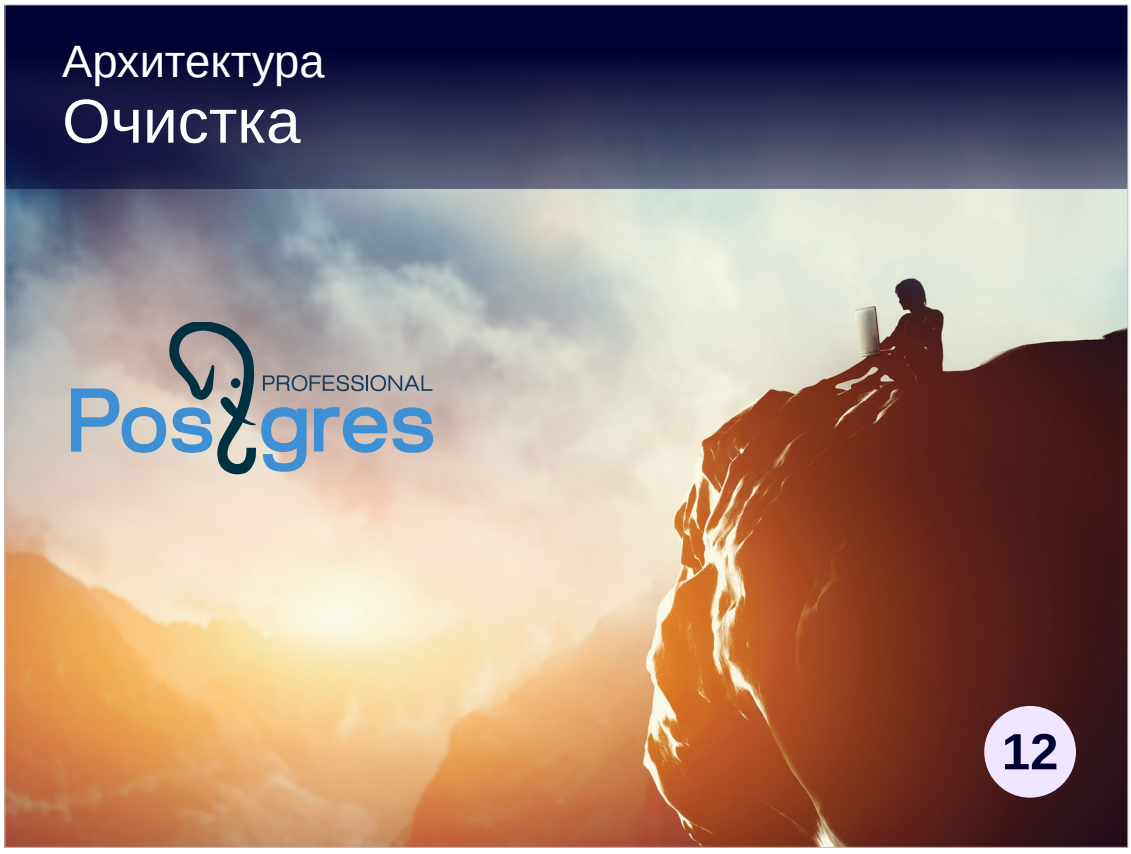


Авторы: Егор Рогов, Павел Лузанов

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

Темы

Очистка версий строк и анализ таблиц

Заморозка версий строк

Очистка и анализ

Как устроен процесс очистки

Анализ таблиц

Полная очистка

Очистка

Выполняется командой VACUUM

- очищает ненужные версии строк в табличных страницах (пропуская страницы, уже отмеченные в карте видимости)
- очищает индексные записи, ссылающиеся на очищенные версии строк
- обновляет карты видимости и свободного пространства

Обычно работает в автоматическом режиме

- частота обработки зависит от интенсивности изменений в таблице
- процессы autovacuum launcher и autovacuum worker

Как мы уже знаем, для реализации многоверсионности в таблице накапливаются исторические версии строк, а в индексах — ссылки на такие исторические версии. Когда версия строки уходит за «горизонт» базы данных, ее можно удалить, освобождая место для других версий.

Очистка, выполняемая командой VACUUM, обрабатывает всю таблицу, включая все созданные на ней индексы. При этом удаляются и ненужные версии строк, и указатели на них в начале страницы.

Обычно очистка запускается не вручную, а работает в автоматическом режиме в зависимости от количества изменений в таблицах. Чем чаще изменяются строки, тем чаще автоочистка обрабатывает таблицу и ее индексы.

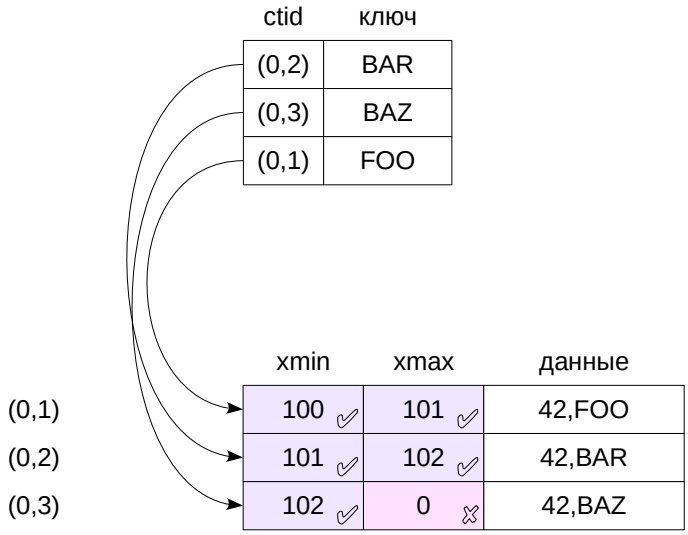
При включенной автоочистке в системе всегда присутствует процесс autovacuum launcher, который занимается планированием работы. Реальную очистку выполняют процессы autovacuum worker, несколько экземпляров которых могут работать параллельно.

Важно, чтобы процесс автоочистки был правильно настроен. Как это делается, рассматривается в курсе DBA2 «Администрирование PostgreSQL. Настройка и мониторинг». Если автоочистка не будет срабатывать вовремя, таблицы будут разрастаться в размерах. Кроме того, важно, чтобы в системе, в которой данные активно изменяются (OLTP), не было длинных транзакций, удерживающих горизонт базы данных и мешающих очистке.

https://postgrespro.ru/docs/postgresql/12/sql-vacuum

https://postgrespro.ru/docs/postgresql/12/routine-vacuuming

Пример

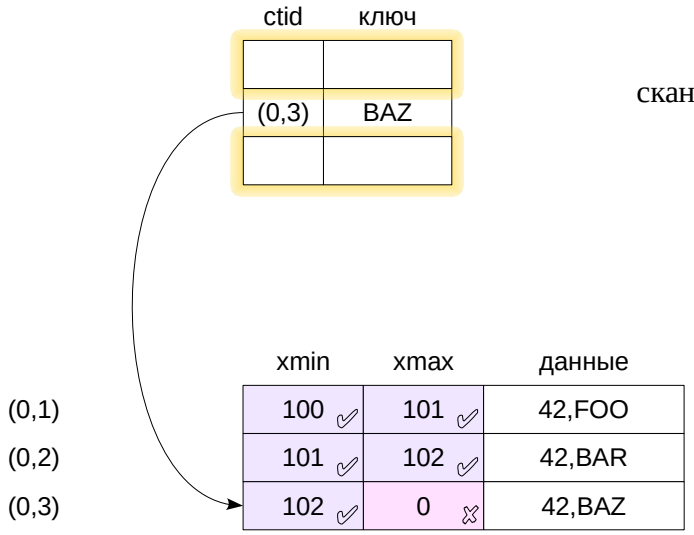


Рассмотрим пример.

На рисунке приведена ситуация до очистки. В табличной странице три версии одной строки. Две из них неактуальны, не видны ни в одном снимке и могут быть удалены.

В индексе есть три ссылки на каждую из версий строки.

Фаза очистки индексов



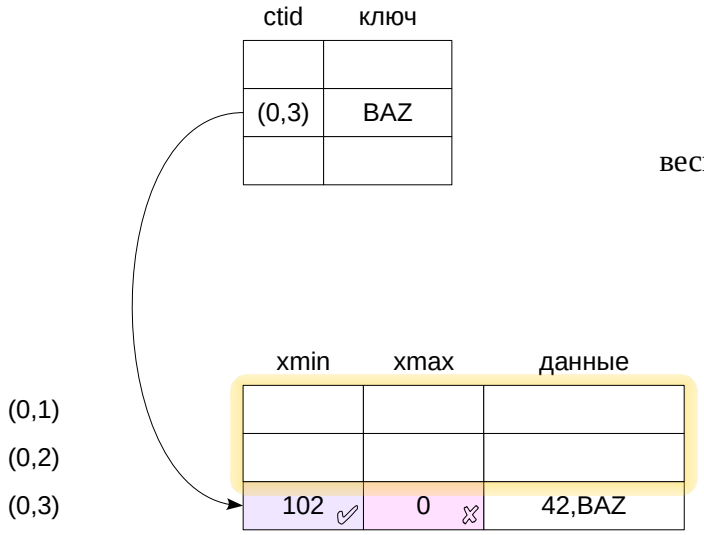
Очистка обрабатывает данные постранично. Таблица при этом может использоваться обычным образом и для чтения, и для изменения (однако одновременное выполнение для таблицы таких команд, как CREATE INDEX, ALTER TABLE и др. будет невозможно — подробнее см. тему «Обзор блокировок»).

Сначала таблица сканируется в поисках версий строк, которые можно очистить. Для этого (с помощью карты видимости) читаются только те страницы, в которых происходила какая-то активность. Идентификаторы очищаемых строк запоминаются.

Затем начинается фаза очистки индексов. *Каждый* из индексов, созданных на таблице *полностью сканируется* в поисках записей, которые ссылаются на очищаемые версии строк (это еще один повод не создавать лишних индексов!). Найденные индексные записи очищаются.

После этой фазы в индексе уже нет ссылок на устаревшие версии строк, но сами версии строк еще присутствуют в таблице.

Фаза очистки таблицы



Затем выполняется фаза очистки таблицы. На этой фазе таблица снова сканируется, и из нее вычищаются версии строк с запомненными ранее идентификаторами.

После этой фазы устаревших версий строк нет ни в индексах, ни в таблице.

Заметим, что фазы очистки индексов и таблицы могут поочередно выполняться несколько раз, если не хватает памяти для того чтобы сразу запомнить все идентификаторы очищаемых строк. Такой ситуации стараются избегать правильной настройкой автоочистки.

В процессе очистки таблицы обновляются и карта видимости (если на странице не остается неактуальных версий строк), и карта свободного пространства, в которой отражается наличие свободного места в страницах.

Очистка

```
=> CREATE DATABASE arch_vacuum;
```

```
CREATE DATABASE
```

```
=> \c arch_vacuum
```

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

Создадим таблицу и запретим для нее автоматическую очистку (чтобы управлять моментом срабатывания).

```
=> CREATE TABLE t(
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    n float
) WITH (autovacuum_enabled = off);
```

```
CREATE TABLE
```

Вставляем в таблицу некоторое количество строк:

```
=> INSERT INTO t(n) SELECT random() FROM generate_series(1,100000);
```

```
INSERT 0 100000
```

```
=> SELECT pg_size_pretty(pg_relation_size('t'));
```

```
 pg_size_pretty
----------------
 4328 kB
(1 row)
```

Теперь, чтобы еще раз напомнить про понятие горизонта, откроем в другом сеансе транзакцию с активным снимком данных.

```
=> \c arch_vacuum
```

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

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

```
BEGIN
```

```
=> SELECT 1; -- не важно, какой запрос
```

```
 ?column?
----------
        1
(1 row)
```

Горизонт для базы данных определяется этим снимком:

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

```
 backend_xmin
--------------
          530
(1 row)
```

Обновим все строки таблицы.

```
=> UPDATE t SET n = n + 1;
```

```
UPDATE 100000
```

Как изменится размер таблицы?

```
=> SELECT pg_size_pretty(pg_relation_size('t'));
```

```
 pg_size_pretty
----------------
 8656 kB
(1 row)
```

Размер увеличился в два раза относительно начального.

Выполним теперь очистку и попросим ее рассказать о том, что происходит:

```
=> VACUUM VERBOSE t;
```

```
INFO:  vacuuming "public.t"
INFO:  index "t_pkey" now contains 200000 row versions in 825 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  "t": found 0 removable, 200000 nonremovable row versions in 1082 out of 1082 pages
DETAIL:  100000 dead row versions cannot be removed yet, oldest xmin: 530
There were 0 unused item identifiers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.01 s, system: 0.00 s, elapsed: 0.02 s.
VACUUM
```

Обратите внимание:

- 20000 nonremovable row versions (ни одна версия строк не может быть очищена),
- 10000 dead row version cannot be removed yet (половина из них — неактуальные),
- oldest xmin показывает текущий горизонт,
- из индекса также ничего не очищено.

Теперь завершим параллельную транзакцию...

```
=> COMMIT;
```

```
COMMIT
```

...и снова вызовем очистку.

```
=> VACUUM VERBOSE t;
```

```
INFO:  vacuuming "public.t"
INFO:  scanned index "t_pkey" to remove 100000 row versions
DETAIL:  CPU: user: 0.01 s, system: 0.00 s, elapsed: 0.01 s
INFO:  "t": removed 100000 row versions in 541 pages
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  index "t_pkey" now contains 100000 row versions in 825 pages
DETAIL:  100000 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  "t": found 100000 removable, 100000 nonremovable row versions in 1082 out of 1082 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 531
There were 0 unused item identifiers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.02 s, system: 0.00 s, elapsed: 0.04 s.
VACUUM
```

Теперь очищены все неактуальные версии строк и указатели на них из индекса. А что с размером таблицы?

```
=> SELECT pg_size_pretty(pg_relation_size('t'));
```

```
 pg_size_pretty
----------------
 8656 kB
(1 row)
```

Он не изменился, поскольку все свободное место находится внутри страниц — оно не возвращается операционной системе, но будет использовано для размещения новых версий строк.

Анализ

Выполняется командой ANALYZE

- собирает статистику для планировщика
- можно вместе с очисткой командой VACUUM ANALYZE

Обычно работает в автоматическом режиме

- вместе с автоматической очисткой

Еще одна задача, которую обычно совмещают с очисткой, — анализ, то есть сбор статистической информации для планировщика запросов. Анализируется число строк в таблице, распределение данных по столбцам и т. п. Более подробно это рассматривается в курсе QPT «Оптимизация запросов».

Вручную анализ выполняется командой ANALYZE (только анализ) или VACUUM ANALYZE (и очистка, и анализ).

Обычно очистка и анализ запускаются не вручную, а работают вместе в автоматическом режиме. Частота автоанализа также зависит от интенсивности изменений и допускает тонкую настройку.

https://postgrespro.ru/docs/postgresql/12/sql-analyze

Полная очистка

Перестраивает файлы и возвращает место файловой системе

Команда VACUUM FULL

- полностью перестраивает таблицу и все ее индексы
- монопольно блокирует таблицу и ее индексы на все время работы

Команда REINDEX

- полностью перестраивает только индексы
- монопольно блокирует индекс и запрещает запись в таблицу
- вариант REINDEX CONCURRENTLY не мешает чтению и записи, но выполняется дольше

Обычная очистка не решает всех задач по освобождению места. Если таблица или индекс сильно выросли в размерах, то очистка не приведет к сокращению числа страниц (и к уменьшению файлов). Вместо этого внутри существующих страниц появятся «дыры», которые могут быть использованы для вставки новых строк или изменения существующих. Единственно исключение составляют полностью очищенные страницы, находящиеся в конце файла — такие страницы «откусываются» и возвращаются операционной системе.

Если размер файлов превышает некие разумные пределы, можно выполнить команду VACUUM FULL для полной очистки. При этом таблица и все ее индексы перестраиваются полностью с нуля, а данные упаковываются максимально компактно.

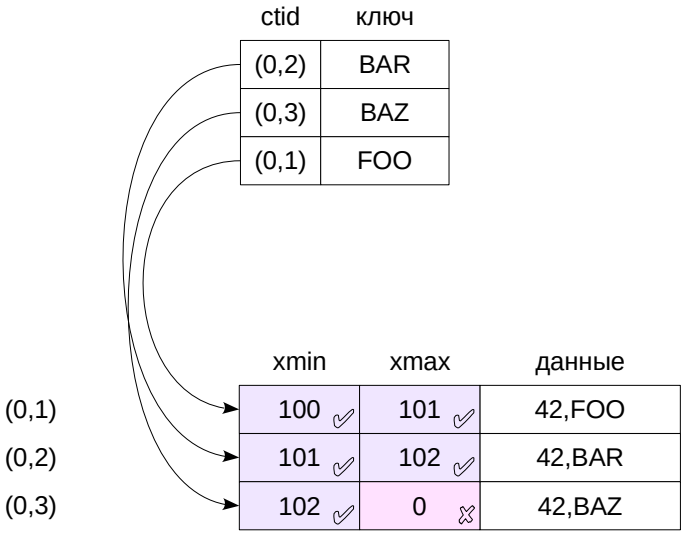
Полная очистка не предполагает регулярного использования, так как полностью блокирует всякую работу с таблицей (включая и выполнение запросов к ней) на все время своей работы.

https://postgrespro.ru/docs/postgresql/12/sql-vacuum

Команда REINDEX перестраивает индексы, не трогая при этом таблицу. Фактически, VACUUM FULL использует эту команду для того, чтобы перестроить индексы. Вариант этой команды с ключевым словом CONCURRENTLY работает дольше, но не блокирует индекс и не мешает чтению и обновлению данных.

https://postgrespro.ru/docs/postgresql/12/sql-reindex

Пример

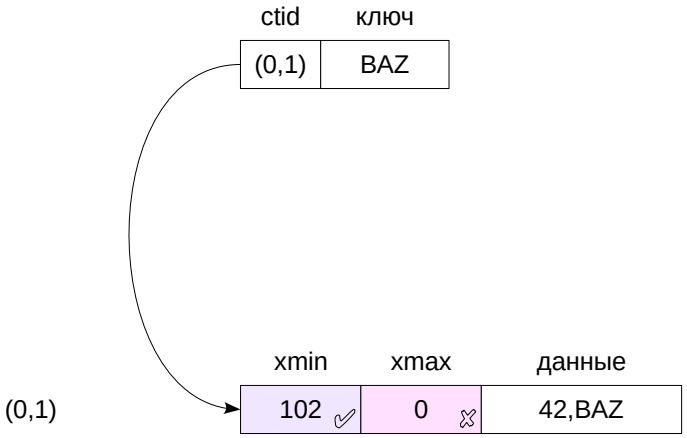


На рисунке приведена та же самая ситуация до очистки, которую мы уже рассматривали.

В табличной странице три версии одной строки. Две из них неактуальны, не видны ни в одном снимке и могут быть удалены.

В индексе есть три ссылки на каждую из версий строки.

После полной очистки



После выполнения полной очистки файлы данных индекса и таблицы перестроены заново, нумерация версий строк изменилась. В страницах отсутствуют «дыры», данные упакованы максимально плотно.

Полная очистка

Вызываем полную очистку таблицы.

```
=> VACUUM FULL VERBOSE t;
```

```
INFO:  vacuuming "public.t"
INFO:  "t": found 0 removable, 100000 nonremovable row versions in 1082 pages
DETAIL:  0 dead row versions cannot be removed yet.
CPU: user: 0.03 s, system: 0.00 s, elapsed: 0.05 s.
VACUUM
```

Таблица и индекс теперь полностью перестроены.

Как изменится размер таблицы?

```
=> SELECT pg_size_pretty(pg_relation_size('t'));
```

```
 pg_size_pretty
----------------
 4328 kB
(1 row)
```

Размер вернулся к первоначальному.

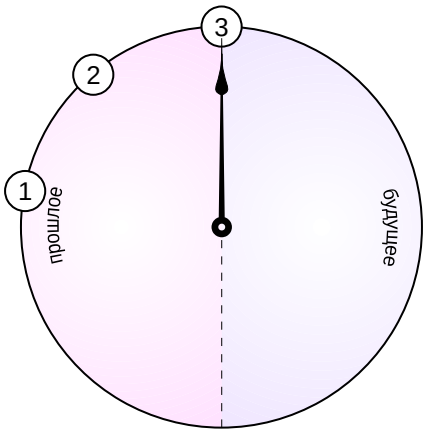
Заморозка

Проблема переполнения счетчика транзакций

Заморозка версий строк

Переполнение счетчика

пространство номеров транзакций (32 бита) закольцовано
половина номеров — прошлое, половина — будущее

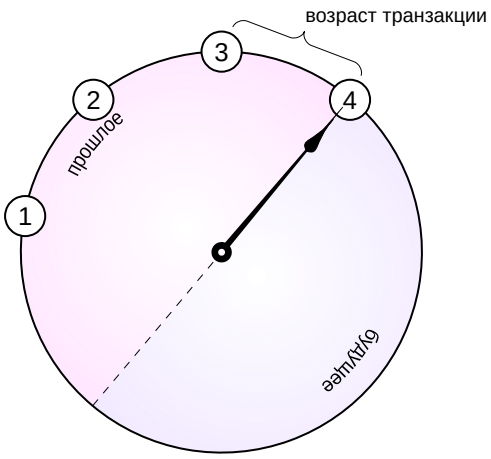


Под номер транзакции в PostgreSQL выделено 32 бита. Это довольно большое число (около 4 млрд номеров), но при активной работе сервера оно вполне может быть исчерпано. Например при нагрузке 1000 транзакций в секунду это произойдет всего через полтора месяца непрерывной работы.

Но мы говорили о том, что механизм многоверсионности полагается на последовательную нумерацию транзакций — из двух транзакций транзакция с меньшим номером считается начавшейся раньше. Понятно, что нельзя просто обнулить счетчик и продолжить нумерацию заново.

Почему под номер транзакции не выделено 64 бита — ведь это полностью исключило бы проблему? Дело в том, что (как рассматривалось в теме «Обзор внутреннего устройства») в заголовке каждой версии строки хранятся два номера транзакций — xmin и xmax. Заголовок и так достаточно большой, а увеличение разрядности привело бы к его увеличению еще на 8 байт.

Переполнение счетчика

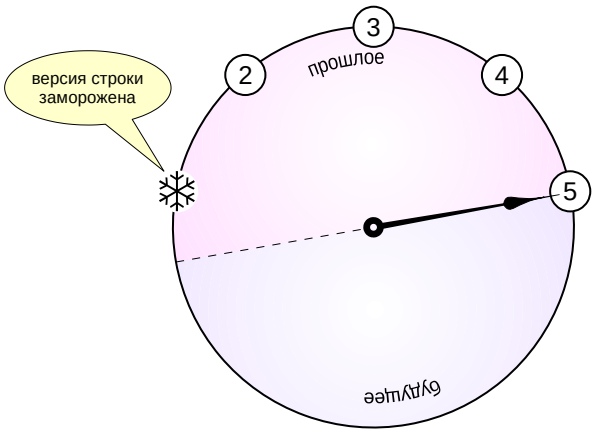


Поэтому вместо линейной схемы все номера транзакций закольцованы. Для любой транзакции половина номеров «против часовой стрелки» считается принадлежащей прошлому, а половина «по часовой стрелке» — будущему.

Возрастом транзакции называется число транзакций, прошедших с момента ее появления в системе (независимо от того, переходил ли счетчик через ноль или нет).

Переполнение счетчика

процесс очистки выполняет заморозку старых версий строк
после заморозки номер транзакции xmin может быть использован заново

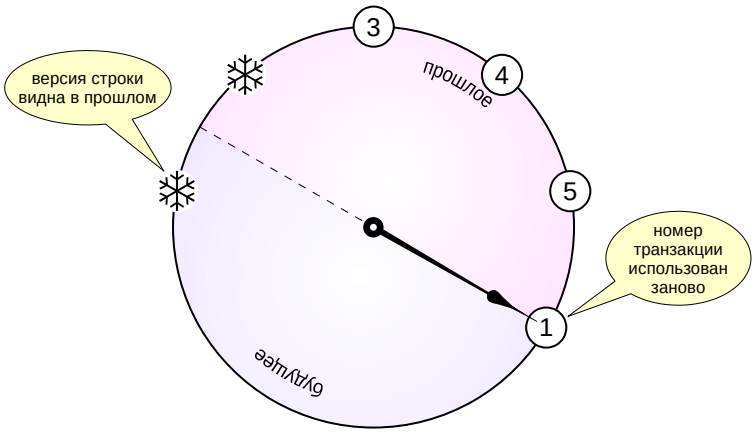


В такой закольцованной схеме возникает неприятная ситуация. Транзакция, находившаяся в далеком прошлом (транзакция 1), через некоторое время окажется в той половине круга, которая относится к будущему. Это, конечно, нарушило бы правила видимости для версий строк, созданных этой транзакцией, и привело бы к проблемам.

Чтобы не допустить путешествий из прошлого в будущее, процесс очистки выполняет еще одну задачу. Он находит достаточно старые и «холодные» версии строк (которые видны во всех снимках и изменение которых уже маловероятно) и специальным образом помечает — «замораживает» — их.

Переполнение счетчика

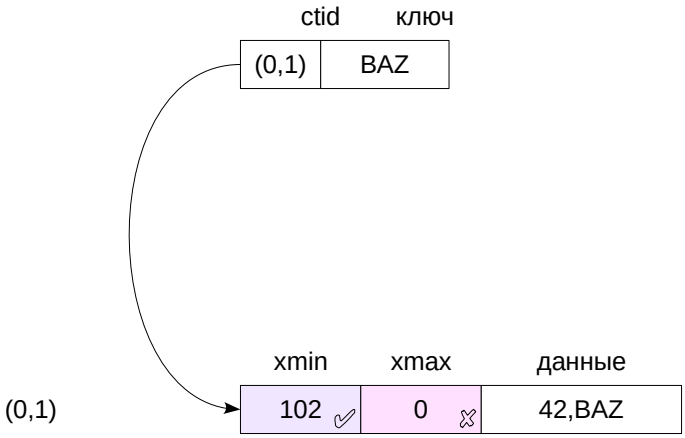
замороженные версии строк считаются «бесконечно старыми»



Замороженная версия строки считается старше любых обычных данных и всегда видна во всех снимках данных. При этом уже не требуется смотреть на номер транзакции xmin, и этот номер может быть безопасно использован заново. Таким образом, замороженные версии строк всегда остаются в прошлом.

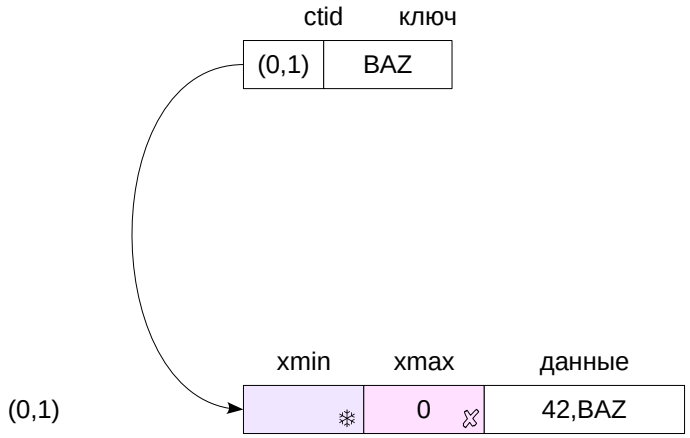
https://postgrespro.ru/docs/postgresql/12/routine-vacuuming#VACUUM-FOR-WRAPAROUND

Пример



На рисунке приведена ситуация до заморозки. В табличной странице после очистки осталась только одна актуальная версия строки.

После заморозки



замораживается только номер xmin

версии строк с xmax являются неактуальными и будут очищены

Для того чтобы пометить номер транзакции xmin как замороженный, выставляется специальная комбинация информационных битов в заголовке версии строки.

Заметим, что транзакцию xmax замораживать не нужно. Ее наличие означает, что данная версия строки больше не актуальна. После того, как она перестанет быть видимой в снимках данных, такая версия строки будет очищена.

Важно, чтобы версии строк замораживались вовремя. Если возникнет ситуация, при которой еще не замороженная транзакция рискует попасть в будущее, PostgreSQL аварийно остановится. Это возможно в двух случаях: либо транзакция не завершена и, следовательно, не может быть заморожена, либо не сработала очистка.

При запуске сервера после аварийного останова транзакция будет автоматически оборвана; дальше администратор должен вручную выполнить очистку и после этого система сможет продолжить работать дальше.

Заморозка вручную

VACUUM FREEZE

- принудительная заморозка версий строк с xmin любого возраста
- тот же эффект и при VACUUM FULL, CLUSTER

COPY … WITH FREEZE

- принудительная заморозка во время загрузки
- таблица должна быть создана или опустошена в той же транзакции
- могут нарушаться правила изоляции транзакции

Иногда бывает удобно управлять заморозкой вручную, а не дожидаться автоочистки.

Заморозку можно вызвать вручную командой VACUUM FREEZE — при этом будут заморожены все версии строк, без оглядки на возраст транзакций. При перестройке таблицы командами VACUUM FULL или CLUSTER все строки также замораживаются.

https://postgrespro.ru/docs/postgresql/12/sql-vacuum

Данные можно заморозить и при начальной загрузке с помощью команды COPY, указав параметр FREEZE. Для этого таблица должна быть создана (или опустошена командой TRUNCATE) в той же транзакции, что и COPY. Поскольку для замороженных строк действуют отдельные правила видимости, такие строки будут видны в снимках данных других транзакций в нарушение обычных правил изоляции (это касается транзакций с уровнем Repeatable Read или Serializable), хотя обычно это не представляет проблемы.

https://postgrespro.ru/docs/postgresql/12/sql-copy

Итоги

Многоверсионность требует очистки старых версий строк

- обычная очистка — как правило в автоматическом режиме
- полная очистка — в экстренных случаях

Переполнение счетчика транзакций требует заморозки

Долгие транзакции и одномоментное обновление большого объема данных приводят к проблемам

- таблицы и индексы увеличиваются в размерах
- избыточные сканирования индексов при очистке

Практика

1. В демонстрации было показано, что при обновлении всех строк таблицы одной командой UPDATE объем данных на диске увеличивается вдвое.
   Повторите эксперимент, обновляя таблицу небольшими пакетами строк, чередуя обновление с запуском очистки. Как сильно увеличилась таблица на этот раз?
2. Сравните время, за которое удаляются все строки таблицы при использовании команд DELETE и TRUNCATE.

1. С размером пакета можно экспериментировать; возьмите, например, 1 % от числа строк в таблице.

Для пакетной обработки нужно решить, как именно выбирать N строк из имеющихся в таблице.

Учтите, что обработка какого-то пакета может (в принципе) завершиться ошибкой. Важно, чтобы это не повлияло на возможность продолжить обработку и довести ее до конца. Например, если выбирать строки по диапазонам значений идентификатора, то как определить, какие пакеты успешно обработаны, а какие — нет?

2. Воспользуйтесь командой psql `\timing`.

1. Пакетное обновление

```
=> CREATE DATABASE mvcc_vacuum_overview;

CREATE DATABASE

=> \c mvcc_vacuum_overview

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

Создадим таблицу:

```
=> CREATE TABLE t(
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    n float,
    processed boolean DEFAULT false
);

CREATE TABLE

=> INSERT INTO t(n) SELECT random() FROM generate_series(1,100000);

INSERT 0 100000
```

Третий столбец пригодится нам для пакетного обновления.

Размер таблицы:

```
=> SELECT pg_size_pretty(pg_relation_size('t'));

 pg_size_pretty
----------------
 5096 kB
(1 row)
```

Еще раз проверим показанное в демонстрации. Обновляем все строки:

```
=> UPDATE t SET n = n + 1;

UPDATE 100000
```

Размер таблицы после обновления:

```
=> SELECT pg_size_pretty(pg_relation_size('t'));

 pg_size_pretty
----------------
 10192 kB
(1 row)
```

Таблица увеличилась в два раза.

Теперь нам надо очистить таблицу, поэтому заодно засечем время выполнения команды TRUNCATE:

```
=> \timing on

Timing is on.

=> TRUNCATE t;

TRUNCATE TABLE
Time: 8,863 ms

=> \timing off

Timing is off.
```

Заново вставляем те же строки:

```
=> INSERT INTO t(n) SELECT random() FROM generate_series(1,100000);

INSERT 0 100000

=> SELECT pg_size_pretty(pg_relation_size('t'));

 pg_size_pretty
----------------
 5096 kB
(1 row)
```

Для того чтобы контролировать обработанные строки, будем использовать столбец processed. Для удобства создадим функцию, выполняющую обновление пакета строк, причем размер пакета задается параметром:

```
=> CREATE FUNCTION do_update(batch_size integer) RETURNS void AS $$
    WITH batch AS (
        -- отбираем необработанные строки для пакета
        -- и блокируем их, пропуская уже заблокированные
        SELECT * FROM t
        WHERE NOT processed
        LIMIT batch_size
        FOR UPDATE SKIP LOCKED
    )
    UPDATE t
    SET n = n + 1,
        processed = true
    WHERE id IN (SELECT id FROM batch);
$$ LANGUAGE sql VOLATILE;
```

```
CREATE FUNCTION
```

Предложение FOR UPDATE SKIP LOCKED будет подробнее рассмотрено в теме «Обзор блокировок». Такой способ даст необходимое количество необработанных строк (из числа имеющихся в таблице), причем выполнение не будет задержано другими обновлениями, которые могут выполняться в это же время. Удобно также, что этот способ не требует вычисления каких-либо диапазонов. Обновление просто надо продолжать до тех пор, пока в таблице не останется необработанных строк.

Поскольку выполнение команды VACUUM не допускается в блоке транзакции, воспользуется средствами bash, чтобы выполнить нужные команды:

```
student$ for i in {1..100}
do
  psql -d mvcc_vacuum_overview -c 'SELECT do_update(1000);'
  psql -d mvcc_vacuum_overview -c 'VACUUM;'
done >/dev/null
```

Проверим, что все строки обработаны:

```
=> SELECT count(*) FROM t WHERE NOT processed;
```

```
 count
-------
     0
(1 row)
```

Размер таблицы после пакетного обновления:

```
=> SELECT pg_size_pretty(pg_relation_size('t'));
```

```
 pg_size_pretty
----------------
 5168 kB
(1 row)
```

Таблица увеличилась чуть больше, чем на 1%.

2. Удаление строк

```
=> \timing on
```

```
Timing is on.
```

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

```
DELETE 100000
Time: 118,617 ms
```

```
=> \timing off
```

```
Timing is off.
```

Удаление всех строк командой DELETE выполняется значительно дольше, чем TRUNCATE. Кроме того, после DELETE в таблице остаются удаленные версии строк, которые должны быть очищены.

По принципу работы команда TRUNCATE напоминает VACUUM FULL: она перезаписывает файл данных новым (пустым) файлом, и для этого полностью блокирует работу с таблицей для других транзакций.