

# Многоверсионность Заморозка



## Авторские права

© Postgres Professional, 2016–2025

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.

Проблема переполнения счетчика транзакций

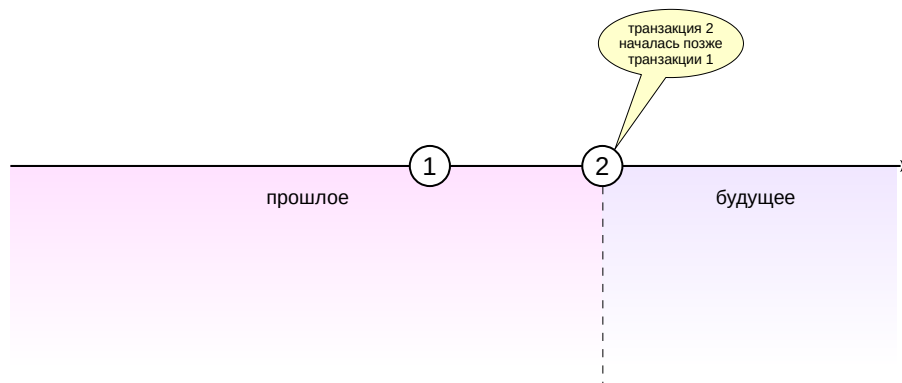
Заморозка версий строк и правила видимости

Настройка автоочистки для выполнения заморозки

Заморозка вручную

# Переполнение счетчика

меньшие номера — прошлое, бóльшие — будущее  
разрядность счетчика — 32 бита, что делать при переполнении?



3

Кроме освобождения места в страницах, очистка выполняет также задачу по предотвращению проблем, связанных с переполнением счетчика транзакций.

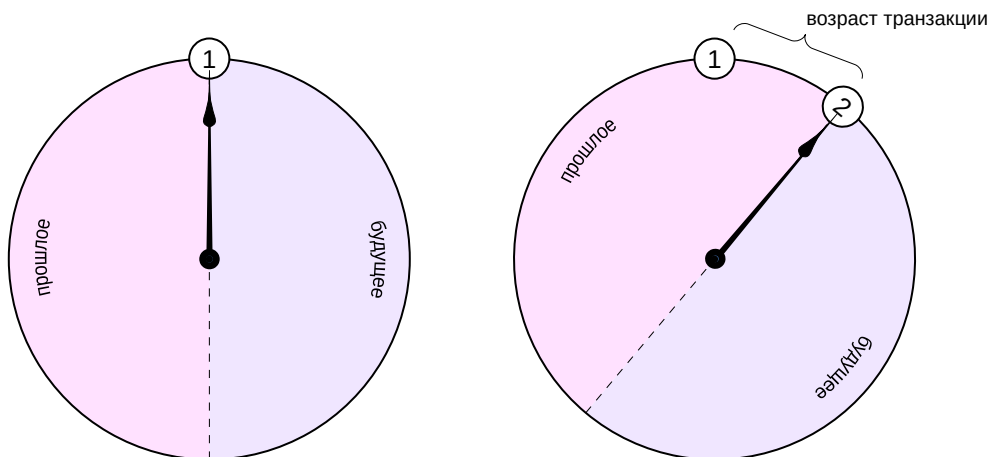
Под номер транзакции в PostgreSQL выделено 32 бита. Это довольно большое число (около 4 млрд номеров), но при активной работе сервера оно вполне может быть исчерпано. Например при нагрузке 1000 транзакций в секунду это произойдет всего через полтора месяца непрерывной работы.

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 «Страницы и версии строк») 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.

# Нумерация по кругу

пространство номеров транзакций закольцовано  
половина номеров — прошлое, половина — будущее

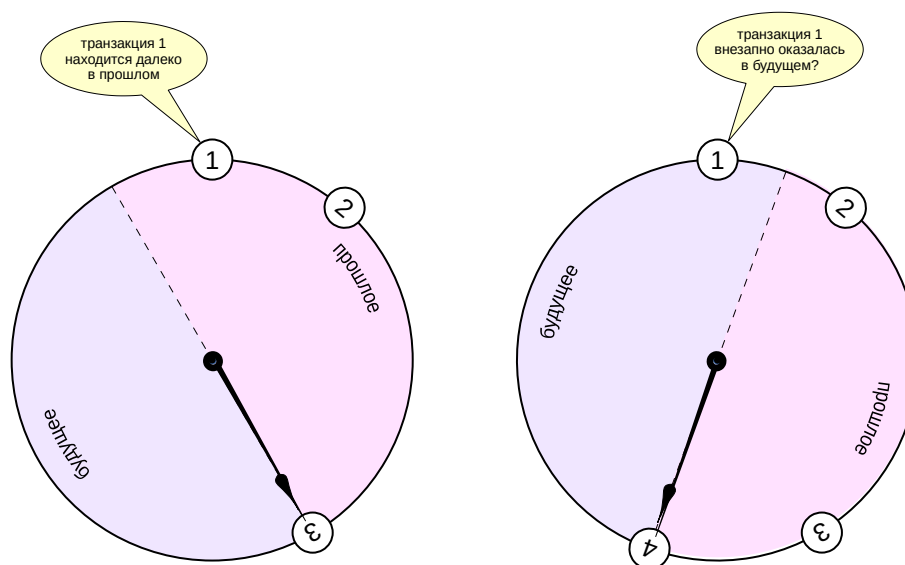


4

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).

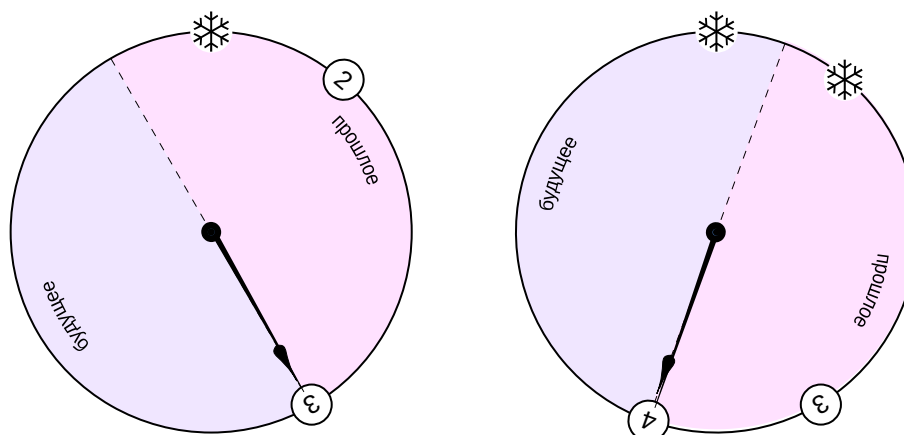
# Проблема видимости



В такой закольцованной схеме возникает неприятная ситуация. Транзакция, находившаяся в далеком прошлом (транзакция 1 на слайде), через некоторое время окажется в той половине круга, которая относится к будущему. Это, конечно, нарушит правила видимости и приведет к проблемам.

# Заморозка версий строк

замороженные версии строк считаются «бесконечно старыми»  
номер транзакции xmin может быть использован заново



6

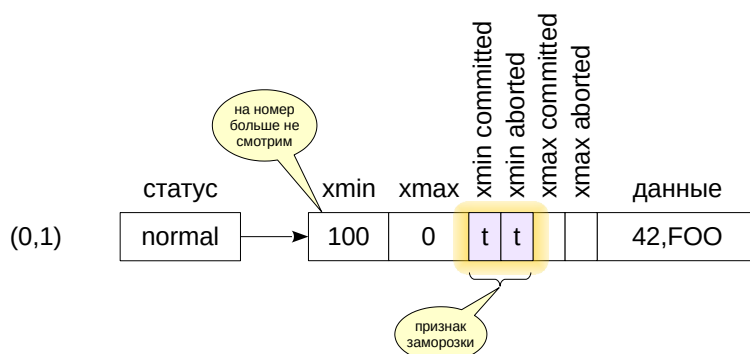
Чтобы не допустить путешествий из прошлого в будущее, процесс очистки выполняет еще одну задачу. Он находит достаточно старые и «холодные» версии строк (которые видны во всех снимках и изменение которых уже маловероятно) и специальным образом помечает — «замораживает» — их. Замороженная версия строки 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/16/routine-vacuuming#VACUUM-FOR-WRAPAROUND>

# Заморозка версий строк

## Еще одна задача процесса очистки

если вовремя не заморозить версии строк, они окажутся в будущем и сервер остановится для предотвращения ошибки



7

Для того чтобы пометить версию строки как замороженную, для транзакции xmin выставляются одновременно оба бита-подсказки — бит фиксации и бит отмены.

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.

Многие источники (включая документацию) упоминают специальный номер FrozenTransactionId = 2, который записывается на место xmin в замороженных версиях. Такая система действовала до версии 9.4, но сейчас заменена на биты-подсказки — это позволяет сохранить в версии строки исходный номер транзакции, что удобно для целей поддержки и отладки. Однако транзакции с номером 2 еще могут встретиться в старых системах, даже обновленных до последних версий.

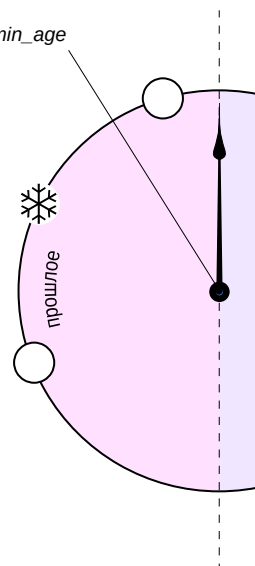
Важно, чтобы версии строк замораживались вовремя. Если возникнет ситуация, при которой еще не замороженный номер транзакции рискует попасть в будущее, PostgreSQL аварийно остановится. Это возможно в двух случаях: либо транзакция не завершена и, следовательно, не может быть заморожена, либо не сработала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

При запуске сервера транзакция будет автоматически отменена; дальше администратор должен вручную выполнить очистку, и после этого система сможет продолжить работу.

***vacuum\_freeze\_min\_age***

минимальный возраст,  
с которого начинается заморозка

*vacuum\_freeze\_min\_age*



8

Заморозкой управляют четыре основных параметра.

Параметр ***vacuum\_freeze\_min\_age*** определяет минимальный возраст транзакции *xmin*, с которого начинается заморозка.

Чем меньше это значение, тем больше может быть накладных расходов. Если строка «горячая» и активно меняется, заморозка ее версий будет пропадать без пользы: уже замороженные версии будут вычищаться, а новые версии придется снова замораживать.

Поэтому более молодые версии строк замораживаются только в тех случаях, когда это точно не добавляет работы, например, если в странице уже требуется заморозка других (более старых) строк или при полной очистке таблицы.

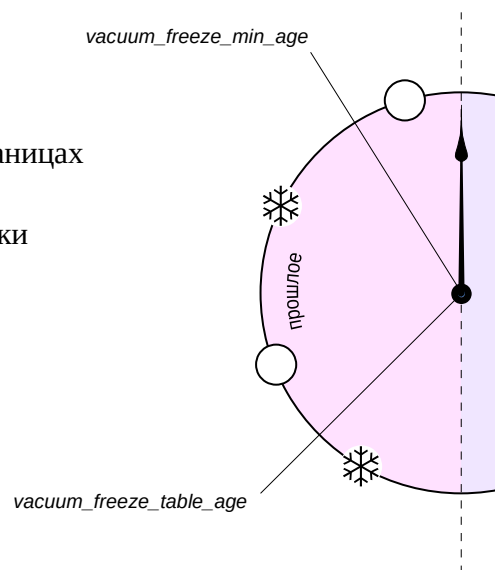
Заметим, что очистка просматривает только страницы, не отмеченные в карте видимости. Если на странице остались только актуальные версии, то очистка не придет в такую страницу и не заморозит их.

В заголовке табличной страницы также имеется признак видимости всех версий строк в ней; очистка использует его вместе с соответствующей отметкой в карте видимости.



## `vacuum_freeze_table_age`

при достижении такого возраста  
замораживаются версии строк на всех страницах  
(«агрессивная» заморозка)  
для ускорения используется карта заморозки



9

Параметр **`vacuum_freeze_table_age`** определяет возраст транзакции, при котором пора выполнять заморозку версий строк на всех страницах таблицы. Такая заморозка называется «агрессивной».

Для каждой таблицы хранится номер транзакции (`pg_class.relrozenxid`), для которого известно, что в версиях строк не осталось более старых незамороженных номеров транзакций. Возраст этой транзакции и сравнивается со значением параметра.

Чтобы не просматривать всю таблицу целиком, вместе с картой видимости ведется *карта заморозки*. В ней отмечены страницы, в которых заморожены все версии строк. Такие страницы при заморозке можно пропускать.

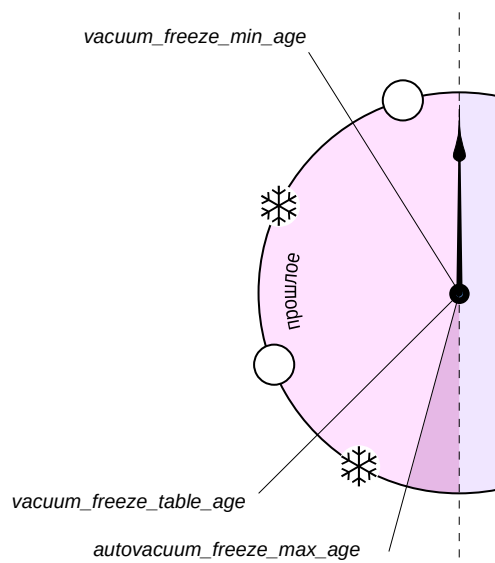
Даже в агрессивном режиме все версии строк с транзакциями младше `vacuum_freeze_min_age` не замораживаются, поэтому после заморозки новый возраст транзакции `relrozenxid` будет равен не нулю, а `vacuum_freeze_min_age`. Таким образом, заморозка всех страниц выполняется раз в  $(vacuum\_freeze\_table\_age - vacuum\_freeze\_min\_age)$  транзакций.

Мы уже говорили, что слишком маленькое значение параметра `vacuum_freeze_min_age` увеличивает накладные расходы на очистку. Но при больших значениях агрессивная заморозка будет выполняться слишком часто, что тоже плохо. Установка этого параметра требует компромисса.

## *autovacuum\_freeze\_max\_age*

при достижении такого возраста  
заморозка запускается принудительно  
определяет размер CLOG

VACUUM (index\_cleanup off)



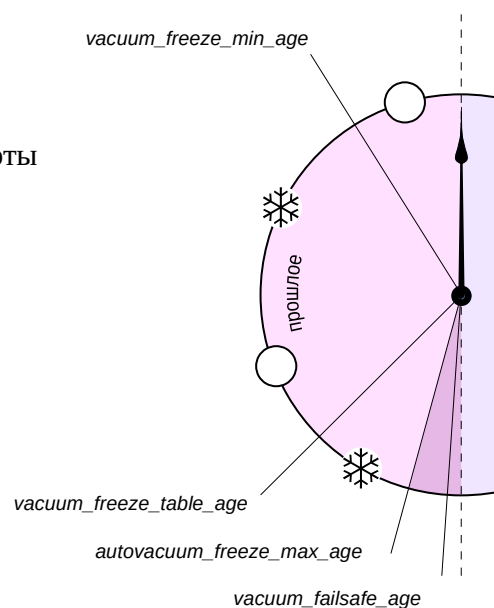
Параметр ***autovacuum\_freeze\_max\_age*** определяет возраст транзакции, при котором заморозка будет выполняться принудительно. Автоочистка для предотвращения последствий переполнения счетчика транзакций запустится, даже если она отключена параметрами.

Этот параметр также определяет размер структуры CLOG: данные о статусе более старых транзакций точно никогда не понадобятся, поэтому часть файлов из PGDATA/pg\_xact может быть удалена.

Если администратор понимает, что автоочистка не успеет заморозить версии строк до переполнения счетчика транзакций, можно воспользоваться ручной очисткой с параметром `index_cleanup off`. В этом случае индексы не будут очищаться, но за счет этого версии строк в таблицах будут заморожены быстрее.

vacuum\_failsafe\_age

при достижении такого возраста  
очистка переходит в защитный режим работы




11

Параметр ***vacuum\_failsafe\_age*** управляет включением *защитного* режима работы очистки, который служит для ускорения заморозки номеров транзакций.

В этом режиме будут отменены регламентные задержки *autovacuum\_vacuum\_cost\_delay* и *vacuum\_cost\_delay*. Также не будут выполняться некоторые необязательные работы (например очистка индексов). Такие меры позволяют очистке быстрее заморозить старые транзакции и перейти в обычный режим работы.

## Конфигурационные параметры

<code>vacuum_freeze_min_age</code>	=	50 000 000	
<code>vacuum_freeze_table_age</code>	=	150 000 000	} ↑?
 <code>autovacuum_freeze_max_age</code>	=	200 000 000	
<code>vacuum_failsafe_age</code>	=	1 600 000 000	

## Параметры хранения таблиц

<code>autovacuum_freeze_min_age</code>
<code>toast.autovacuum_freeze_min_age</code>
<code>autovacuum_freeze_table_age</code>
<code>toast.autovacuum_freeze_table_age</code>
<code>autovacuum_freeze_max_age</code>
<code>toast.autovacuum_freeze_max_age</code>

12

Значения по умолчанию довольно консервативны. Предел для `autovacuum_freeze_max_age` – порядка 2 млрд транзакций, а используется значение в 10 раз меньшее. Можно увеличить значения `vacuum_freeze_table_age` и `autovacuum_freeze_max_age` для уменьшения накладных расходов, но важно понимать, что если по каким-то причинам (например, из-за незавершенной транзакции) автоочистка вовремя не справится с заморозкой, у администратора останется мало времени для принятия мер. Заметьте, что изменение параметра `autovacuum_freeze_max_age` требует перезапуска сервера.

Значение по умолчанию `vacuum_failsafe_age` значительно больше, чем `autovacuum_freeze_max_age`, и если до исчерпания номеров останется мало времени, защитный режим ускорит заморозку.

Отметим, что сервер может скорректировать установленные значения параметров `vacuum_freeze_min_age`, `vacuum_freeze_table_age` и `vacuum_failsafe_age` исходя из значения `autovacuum_freeze_max_age`.

Ряд параметров также можно устанавливать на уровне отдельных таблиц с помощью параметров хранения. Это имеет смысл делать только в особенных случаях, когда таблица действительно требует особого обхождения. Заметьте, что имена параметров на уровне таблиц немного отличаются от имен конфигурационных параметров.

В модуле «Блокировки» рассматриваются т. н. мультитранзакции и дополнительные параметры настройки заморозки для них.

<https://www.postgresql.org/docs/16/runtime-config-client.html>

<https://www.postgresql.org/docs/16/runtime-config-autovacuum.html>

## Заморозка

Установим для демонстрации параметры заморозки.

Небольшой возраст транзакции:

```
=> ALTER SYSTEM SET vacuum_freeze_min_age = 1;
```

ALTER SYSTEM

Возраст, после которого будет выполняться заморозка всех страниц:

```
=> ALTER SYSTEM SET vacuum_freeze_table_age = 3;
```

ALTER SYSTEM

И отключим автоматическую очистку, чтобы запускать ее вручную в нужный момент.

```
=> ALTER SYSTEM SET autovacuum = off;
```

ALTER SYSTEM

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf
-----
t
(1 row)
```

Создадим таблицу с данными. Установим минимальный fillfactor: на каждой странице будет всего две строки.

```
=> CREATE DATABASE mvcc_freeze;
```

CREATE DATABASE

```
=> \c mvcc_freeze
```

You are now connected to database "mvcc\_freeze" as user "student".

```
=> CREATE TABLE t(id integer, s char(300)) WITH (fillfactor = 10);
```

CREATE TABLE

Создадим представление для наблюдения за битами-подсказками на первых двух страницах таблицы.

Сейчас нас интересует только xmin и биты, которые относятся к нему, поскольку версии строк с ненулевым xmax будут очищены. Кроме того, выведем и возраст транзакции xmin.

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

CREATE EXTENSION

```
=> CREATE VIEW t_v AS
SELECT '('||blkno||','||lp||')' as ctid,
       CASE lp_flags
         WHEN 0 THEN 'unused'
         WHEN 1 THEN 'normal'
         WHEN 2 THEN 'redirect to '||lp_off
         WHEN 3 THEN 'dead'
       END AS state,
       t_xmin AS xmin,
       age(t_xmin) AS xmin_age,
       CASE WHEN (t_infomask & 256) > 0 THEN 't' END AS xmin_c,
       CASE WHEN (t_infomask & 512) > 0 THEN 't' END AS xmin_a,
       t_xmax AS xmax
FROM (
  SELECT 0 blkno, * FROM heap_page_items(get_raw_page('t',0))
  UNION ALL
  SELECT 1 blkno, * FROM heap_page_items(get_raw_page('t',1))
) q
ORDER BY blkno, lp;
```

CREATE VIEW

Для того чтобы заглянуть в карту видимости и заморозки, воспользуемся еще одним расширением:

```
=> CREATE EXTENSION pg_visibility;
```

CREATE EXTENSION

Вставляем данные. Сразу выполним очистку, чтобы заполнить карту видимости.

```
=> INSERT INTO t(id, s) SELECT g.id, 'FOO' FROM generate_series(1,100) g(id);
```

```
INSERT 0 100
```

```
=> VACUUM t;
```

```
VACUUM
```

После очистки обе страницы отмечены в карте видимости (all\_visible):

```
=> SELECT * FROM generate_series(0,1) g(blkno), pg_visibility_map('t',g.blkno)
ORDER BY g.blkno;
```

blkno	all_visible	all_frozen
0	t	f
1	t	f

(2 rows)

Каков возраст транзакции, создавшей строки?

```
=> SELECT * FROM t_v;
```

ctid	state	xmin	xmin_age	xmin_c	xmin_a	xmax
(0,1)	normal	748	1	t		0
(0,2)	normal	748	1	t		0
(1,1)	normal	748	1	t		0
(1,2)	normal	748	1	t		0

(4 rows)

Возраст равен 1; версии строк с такой транзакцией еще не будут заморожены.

Обновим строку на нулевой странице. Новая версия попадет на ту же страницу благодаря небольшому значению fillfactor.

```
=> UPDATE t SET s = 'BAR' WHERE id = 1;
```

```
UPDATE 1
```

```
=> SELECT * FROM t_v;
```

ctid	state	xmin	xmin_age	xmin_c	xmin_a	xmax
(0,1)	normal	748	2	t		749
(0,2)	normal	748	2	t		0
(0,3)	normal	749	1			0
(1,1)	normal	748	2	t		0
(1,2)	normal	748	2	t		0

(5 rows)

Сейчас нулевая страница уже будет обработана заморозкой:

- возраст транзакции превышает значение, установленное в vacuum\_freeze\_min\_age;
- страница изменена и исключена из карты видимости.

```
=> SELECT * FROM generate_series(0,1) g(blkno), pg_visibility_map('t',g.blkno)
ORDER BY g.blkno;
```

blkno	all_visible	all_frozen
0	f	f
1	t	f

(2 rows)

Выполняем очистку.

```
=> VACUUM t;
```

```
VACUUM
```

Очистка обработала измененную страницу. У одной версии строки установлены оба бита — это признак заморозки. Другая версия строки слишком молода, однако тоже была заморожена при проходе страницы (это позволило отметить нулевую страницу в карте заморозки):

```
=> SELECT * FROM t_v;
```

ctid	state	xmin	xmin_age	xmin_c	xmin_a	xmax
(0,1)	redirect to 3					
(0,2)	normal	748	2	t	t	0
(0,3)	normal	749	1	t	t	0
(1,1)	normal	748	2	t		0
(1,2)	normal	748	2	t		0

(5 rows)

Теперь обе страницы отмечены в карте видимости (все версии строк на них актуальны). Очистка теперь не будет обрабатывать ни одну из этих страниц, и незамороженные версии строк на первой странице так и останутся незамороженными.

```
=> SELECT * FROM generate_series(0,1) g(blkno), pg_visibility_map('t',g.blkno)
ORDER BY g.blkno;
```

blkno	all_visible	all_frozen
0	t	t
1	t	f

(2 rows)

Именно для такого случая и требуется параметр `vacuum_freeze_table_age`, определяющий, в какой момент нужно просмотреть страницы, отмеченные в карте видимости, если они не отмечены в карте заморозки.

Для каждой таблицы сохраняется наибольший номер транзакции, для которого все версии строк с меньшими номерами `xmin` гарантированно заморожены. Ее возраст и сравнивается со значением параметра.

```
=> SELECT relfrozenxid, age(relfrozenxid) FROM pg_class WHERE relname = 't';
```

relfrozenxid	age
748	2

(1 row)

Сымитируем выполнение еще одной транзакции, чтобы возраст `relfrozenxid` таблицы достиг значения параметра `vacuum_freeze_table_age`.

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

pg_current_xact_id
750

(1 row)

```
=> SELECT relfrozenxid, age(relfrozenxid) FROM pg_class WHERE relname = 't';
```

relfrozenxid	age
748	3

(1 row)

```
=> VACUUM t;
```

VACUUM

Теперь, поскольку гарантированно была проверена вся таблица, номер замороженной транзакции можно увеличить — мы уверены, что в страницах не осталось более старой незамороженной транзакции.

```
=> SELECT relfrozenxid, age(relfrozenxid) FROM pg_class WHERE relname = 't';
```

relfrozenxid	age
751	0

(1 row)

Вот что получилось в страницах:

```
=> SELECT * FROM t_v;
```

ctid	state	xmin	xmin_age	xmin_c	xmin_a	xmax
(0,1)	redirect to 3					
(0,2)	normal	748	3	t	t	0
(0,3)	normal	749	2	t	t	0
(1,1)	normal	748	3	t	t	0
(1,2)	normal	748	3	t	t	0

(5 rows)

Обе страницы теперь отмечены в карте заморозки.

```
=> SELECT * FROM generate_series(0,1) g(blkno), pg_visibility_map('t',g.blkno)
ORDER BY g.blkno;
```

blkno	all_visible	all_frozen
0	t	t
1	t	t

(2 rows)

Номер последней замороженной транзакции есть и на уровне всей БД:

```
=> SELECT datname, datfrozenxid, age(datfrozenxid)
FROM pg_database;
```

datname	datfrozenxid	age
postgres	722	29
student	722	29
template1	722	29
template0	722	29
mvcc_freeze	722	29

(5 rows)

Он устанавливается в минимальное значение из relfrozenxid всех таблиц этой БД. Если возраст datfrozenxid превысит значение параметра autovacuum\_freeze\_max\_age, автоочистка будет запущена принудительно.



## VACUUM

заморозка версий строк по возрасту в соответствии с настройками

## VACUUM FREEZE

принудительная заморозка версий строк с xmin любого возраста

тот же эффект и при VACUUM FULL, CLUSTER

## COPY ... WITH FREEZE

принудительная заморозка сразу после загрузки

таблица должна быть создана или опустошена в той же транзакции

могут нарушиться правила изоляции транзакции

Несмотря на то, что во время работы автоочистки при необходимости выполняется и заморозка, иногда бывает удобно управлять заморозкой вручную.

Команда VACUUM, как и автоочистка, выполнит заморозку в соответствии с настройками.

Если выполнить команду VACUUM FREEZE, будут заморожены все версии строк без оглядки на возраст транзакций (как будто параметры *vacuum\_freeze\_min\_age* и *vacuum\_freeze\_table\_age* равны нулю).

При перестройке таблицы командами VACUUM FULL или CLUSTER все строки также замораживаются.

<https://postgrespro.ru/docs/postgresql/16/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 (для транзакций с уровнем Repeatable Read или Serializable), но обычно это не представляет проблемы. Подробнее такой случай рассматривается в практике.

<https://postgrespro.ru/docs/postgresql/16/sql-copy>

Текущая реализация такой заморозки для таблиц с TOAST полноценно обрабатывает только основную часть таблицы, а в карте видимости признак видимости всех строк не проставляется. А это означает дополнительный проход по всем страницам при последующей очистке.

Пространство номеров транзакций закольцовано  
Достаточно старые версии строк замораживаются  
процессом очистки  
Для оптимизации используется карта заморозки

1. Проверьте с помощью расширения `pageinspect`, что при использовании команды `COPY ... WITH FREEZE` версии строк действительно замораживаются.
2. Убедитесь, что даже на уровне изоляции `Repeatable Read` строки, загруженные командой `COPY ... WITH FREEZE`, оказываются видны в снимке данных.
3. Уменьшив значение параметра `autovacuum_freeze_max_age` и отключив автоочистку, воспроизведите ситуацию принудительного срабатывания автоочистки, выполнив соответствующее количество транзакций. Учтите, что срабатывание произойдет не сразу, а при выполнении ручной очистки какой-нибудь таблицы (или при перезапуске сервера).

16

3. Чтобы транзакциям выделялись настоящие (не виртуальные) номера, в транзакции нужно менять данные.

Можно организовать цикл в `bash`, в котором вызывать `psql` с командой обновления:

```
psql -c 'UPDATE ...'
```

Другой вариант — использовать для организации цикла `PL/pgSQL`.

Для этого можно создать процедуру, выполняющую фиксацию транзакции или использовать блок с обработкой исключений.

При перехвате исключения транзакция будет откатываться к неявной точке сохранения: фактически начнется новая вложенная транзакция (см. тему «Страницы и версии строк»).

Третий вариант — использовать утилиту `pgbench`:

<https://postgrespro.ru/docs/postgresql/16/pgbench>

## 1. Заморозка при COPY WITH FREEZE

Создаем таблицу и загружаем несколько строк в одной и той же транзакции:

```
=> CREATE DATABASE mvcc_freeze;
```

```
CREATE DATABASE
```

```
=> \c mvcc_freeze
```

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

```
=> BEGIN;
```

```
BEGIN
```

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

```
CREATE TABLE
```

```
=> COPY t FROM stdin WITH FREEZE;
```

```
=> 1
```

```
=> 2
```

```
=> 3
```

```
=> \.
```

```
COPY 3
```

```
=> COMMIT;
```

```
COMMIT
```

Проверяем версии строк:

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

```
CREATE EXTENSION
```

```
=> CREATE VIEW t_v AS
```

```
SELECT '(0,'||lp||')' as ctid,  
       CASE lp_flags  
         WHEN 0 THEN 'unused'  
         WHEN 1 THEN 'normal'  
         WHEN 2 THEN 'redirect to '||lp_off  
         WHEN 3 THEN 'dead'  
       END AS state,  
       t_xmin AS xmin,  
       age(t_xmin) AS xmin_age,  
       CASE WHEN (t_infomask & 256) > 0 THEN 't' END AS xmin_c,  
       CASE WHEN (t_infomask & 512) > 0 THEN 't' END AS xmin_a,  
       t_xmax AS xmax,  
       t_ctid
```

```
FROM heap_page_items(get_raw_page('t',0))
```

```
ORDER BY lp;
```

```
CREATE VIEW
```

```
=> SELECT * FROM t_v;
```

ctid	state	xmin	xmin_age	xmin_c	xmin_a	xmax	t_ctid
(0,1)	normal	744	3	t	t	0	(0,1)
(0,2)	normal	744	3	t	t	0	(0,2)
(0,3)	normal	744	3	t	t	0	(0,3)

(3 rows)

## 2. COPY WITH FREEZE и изоляция

В другом сеансе начнем транзакцию с уровнем изоляции Repeatable Read.

```
| => \c mvcc_freeze
```

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

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

```

| BEGIN
|
| => SELECT pg_current_xact_id();
|
|      pg_current_xact_id
|      -----
|                747
|
| (1 row)

```

Обратите внимание, что эта транзакция не должна обращаться к таблице t.

Теперь опустошим таблицу и загрузим в нее новые строки в одной транзакции. Если бы параллельная транзакция прочитала содержимое t, команда TRUNCATE ожидала бы ее завершения.

```

=> BEGIN;

BEGIN

=> TRUNCATE t;

TRUNCATE TABLE

=> COPY t FROM stdin WITH FREEZE;

=> 10
=> 20
=> 30
=> \.

COPY 3

=> COMMIT;

COMMIT

```

Теперь параллельная транзакция видит новые данные, хотя это и нарушает изоляцию:

```

| => SELECT * FROM t;
|
|      n
|      ----
|      10
|      20
|      30
| (3 rows)
|
| => COMMIT;
|
| COMMIT
|
| => \q

```

### 3. Аварийное срабатывание автоочистки

Предварительно заморозим все транзакции во всех базах. Для этого удобно воспользоваться командой vacuumdb:

```

student$ vacuumdb --all --freeze

vacuumdb: vacuuming database "mvcc_freeze"
vacuumdb: vacuuming database "postgres"
vacuumdb: vacuuming database "student"
vacuumdb: vacuuming database "template1"

```

Максимальный возраст незамороженных транзакций по всем БД:

```

=> SELECT datname, datfrozenxid, age(datfrozenxid) FROM pg_database;

   datname   | datfrozenxid | age
-----+-----+----
 postgres   |          749 |  0
 student    |          749 |  0
 template1   |          749 |  0
 template0   |          722 | 27
 mvcc_freeze |          749 |  0
(5 rows)

```

Отключаем автоочистку.

```

=> ALTER SYSTEM SET autovacuum = off;

```

```
ALTER SYSTEM
```

Уменьшаем значения параметров:

```
=> ALTER SYSTEM SET vacuum_freeze_min_age = 1_000;
```

```
ALTER SYSTEM
```

```
=> ALTER SYSTEM SET vacuum_freeze_table_age = 10_000;
```

```
ALTER SYSTEM
```

```
=> ALTER SYSTEM SET autovacuum_freeze_max_age = 100_000; # минимальное значение
```

```
ALTER SYSTEM
```

Требуется перезагрузка сервера.

```
student$ sudo pg_ctlcluster 16 main restart
```

```
student$ psql mvcc_freeze
```

Получить большое количество транзакций можно разными способами; например, можно воспользоваться утилитой `pgbench`. Попросим ее инициализировать свои таблицы и выполнить 100000 транзакций.

```
student$ pgbench -i mvcc_freeze
```

```
dropping old tables...
NOTICE: table "pgbench_accounts" does not exist, skipping
NOTICE: table "pgbench_branches" does not exist, skipping
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
100000 of 100000 tuples (100%) done (elapsed 0.16 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.35 s (drop tables 0.00 s, create tables 0.01 s, client-side generate 0.19 s,
vacuum 0.06 s, primary keys 0.09 s).
```

Выполнение ста тысяч транзакций может занять заметное время, ключ `--protocol=prepared` немного ускоряет работу за счет использования подготовленных операторов:

```
student$ pgbench -t 100000 -P 5 --protocol=prepared mvcc_freeze
```

```
pgbench (16.3 (Ubuntu 16.3-1.pgdg22.04+1))
starting vacuum...end.
progress: 5.0 s, 514.2 tps, lat 1.943 ms stddev 0.525, 0 failed
progress: 10.0 s, 562.0 tps, lat 1.778 ms stddev 0.482, 0 failed
progress: 15.0 s, 480.6 tps, lat 2.080 ms stddev 1.993, 0 failed
progress: 20.0 s, 607.2 tps, lat 1.646 ms stddev 1.238, 0 failed
progress: 25.0 s, 656.6 tps, lat 1.522 ms stddev 0.867, 0 failed
progress: 30.0 s, 667.4 tps, lat 1.498 ms stddev 0.467, 0 failed
progress: 35.0 s, 646.6 tps, lat 1.546 ms stddev 0.444, 0 failed
progress: 40.0 s, 440.0 tps, lat 2.268 ms stddev 1.575, 0 failed
progress: 45.0 s, 614.8 tps, lat 1.628 ms stddev 0.734, 0 failed
progress: 50.0 s, 653.6 tps, lat 1.530 ms stddev 0.420, 0 failed
progress: 55.0 s, 597.4 tps, lat 1.673 ms stddev 0.885, 0 failed
progress: 60.0 s, 602.6 tps, lat 1.659 ms stddev 0.522, 0 failed
progress: 65.0 s, 654.2 tps, lat 1.528 ms stddev 0.340, 0 failed
progress: 70.0 s, 630.2 tps, lat 1.586 ms stddev 0.828, 0 failed
progress: 75.0 s, 659.2 tps, lat 1.517 ms stddev 0.361, 0 failed
progress: 80.0 s, 647.2 tps, lat 1.544 ms stddev 1.255, 0 failed
progress: 85.0 s, 636.4 tps, lat 1.570 ms stddev 0.359, 0 failed
progress: 90.0 s, 627.6 tps, lat 1.594 ms stddev 0.372, 0 failed
progress: 95.0 s, 639.4 tps, lat 1.562 ms stddev 0.327, 0 failed
progress: 100.0 s, 634.4 tps, lat 1.577 ms stddev 0.312, 0 failed
progress: 105.0 s, 616.8 tps, lat 1.620 ms stddev 0.509, 0 failed
progress: 110.0 s, 604.8 tps, lat 1.653 ms stddev 0.482, 0 failed
progress: 115.0 s, 614.2 tps, lat 1.628 ms stddev 0.411, 0 failed
progress: 120.0 s, 638.0 tps, lat 1.567 ms stddev 0.302, 0 failed
progress: 125.0 s, 623.0 tps, lat 1.604 ms stddev 0.439, 0 failed
progress: 130.0 s, 597.6 tps, lat 1.672 ms stddev 0.465, 0 failed
progress: 135.0 s, 581.2 tps, lat 1.719 ms stddev 0.714, 0 failed
progress: 140.0 s, 611.6 tps, lat 1.635 ms stddev 0.462, 0 failed
progress: 145.0 s, 620.4 tps, lat 1.611 ms stddev 2.726, 0 failed
progress: 150.0 s, 649.0 tps, lat 1.539 ms stddev 0.319, 0 failed
progress: 155.0 s, 643.8 tps, lat 1.554 ms stddev 0.494, 0 failed
progress: 160.0 s, 647.8 tps, lat 1.543 ms stddev 0.416, 0 failed
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: prepared
number of clients: 1
number of threads: 1
```

maximum number of tries: 1  
number of transactions per client: 100000  
number of transactions actually processed: 100000/100000  
number of failed transactions: 0 (0.000%)  
latency average = 1.628 ms  
latency stddev = 0.852 ms  
initial connection time = 2.664 ms  
tps = 613.800346 (without initial connection time)

Видно, что возраст незамороженных транзакций превышает установленное пороговое значение (100000):

```
=> SELECT datname, datfrozenxid, age(datfrozenxid) FROM pg_database;
```

datname	datfrozenxid	age
postgres	749	100013
student	749	100013
template1	749	100013
template0	722	100040
mvcc_freeze	749	100013

(5 rows)

Теперь при выполнении команды VACUUM для любой таблицы будет запущен процесс автоочистки.

```
=> VACUUM t;
```

VACUUM

Среди процессов появился autovacuum worker:

```
student$ ps -o pid,command --ppid 115020
```

PID	COMMAND
115021	postgres: 16/main: checkpointer
115022	postgres: 16/main: background writer
115024	postgres: 16/main: walwriter
115025	postgres: 16/main: logical replication launcher
115067	postgres: 16/main: student mvcc_freeze [local] idle
115442	postgres: 16/main: autovacuum worker template0

И через некоторое время транзакции окажутся замороженными:

```
=> SELECT datname, datfrozenxid, age(datfrozenxid) FROM pg_database;
```

datname	datfrozenxid	age
postgres	100762	0
student	100762	0
template1	100762	0
template0	100762	0
mvcc_freeze	99829	933

(5 rows)