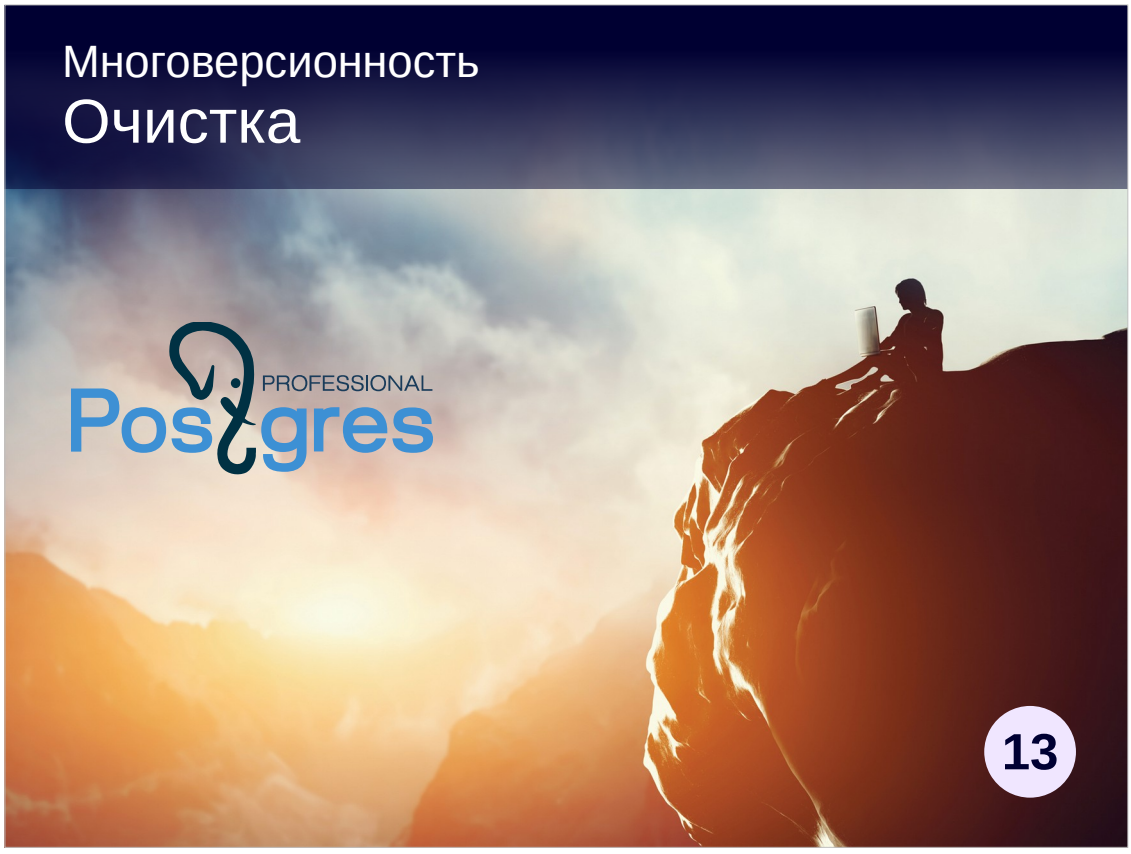


Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов

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

Темы

Обычная очистка и схема ее работы

Регулирование нагрузки

Анализ

Полная очистка и аналоги

Обычная очистка

Выполняется командой VACUUM

- не конфликтует с обычной активностью в системе

Обрабатывает таблицу и все ее индексы

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

Внутристраничная очистка выполняется быстро, но не решает всех задач. Она работает только в пределах одной табличной страницы и не затрагивает индексы.

Очистка, выполняемая командой VACUUM, обрабатывает таблицу полностью, включая все созданные на ней индексы, освобождая место за счет удаления и ненужных версий строк, и указателей.

Обработка происходит в фоновом режиме, 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. модуль «Блокировки»).

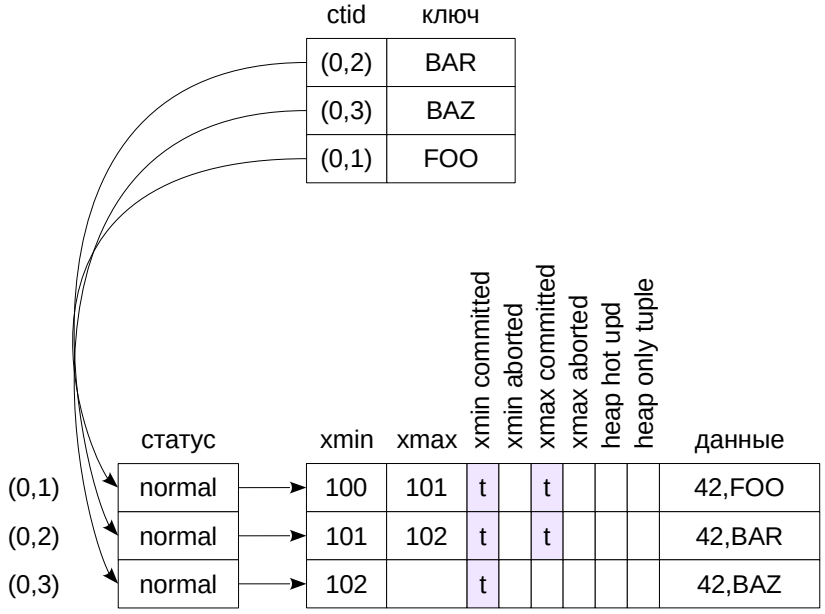
Из таблицы читаются только те страницы, в которых происходила какая-то активность. Для этого используется карта видимости (visibility map), в которой отмечены страницы, содержащие только достаточно старые версии строк, которые гарантированно видимы во всех снимках. Обрабатываются только страницы, не отмеченные в карте, а сама карта при этом обновляется.

В процессе работы обновляется и карта свободного пространства (free space map), в которой отражается наличие свободного места в страницах.

https://postgrespro.ru/docs/postgresql/13/sql-vacuum

https://postgrespro.ru/docs/postgresql/13/routine-vacuuming

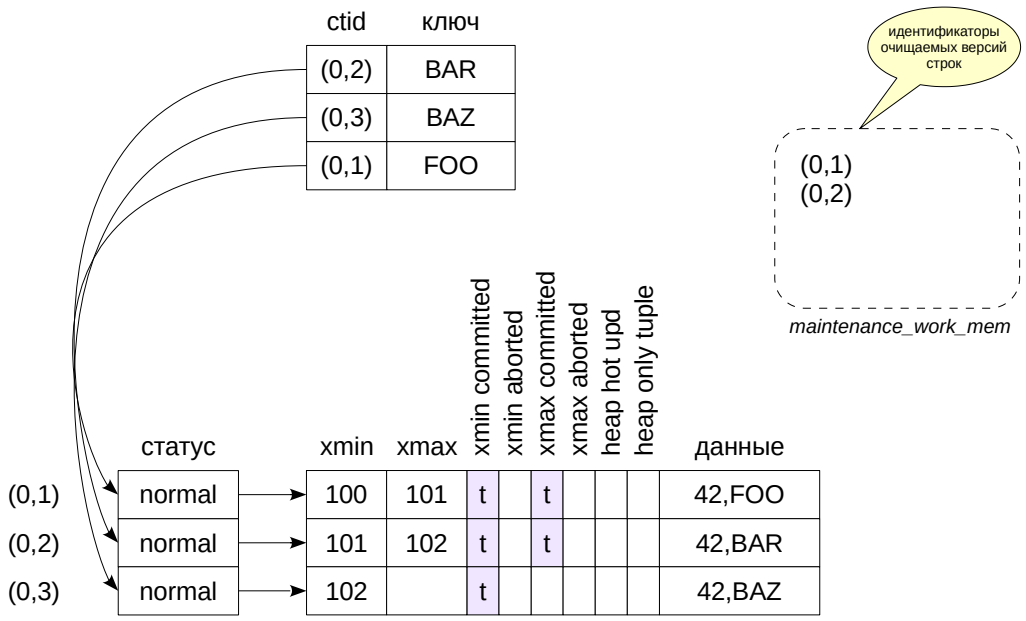
До очистки



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.

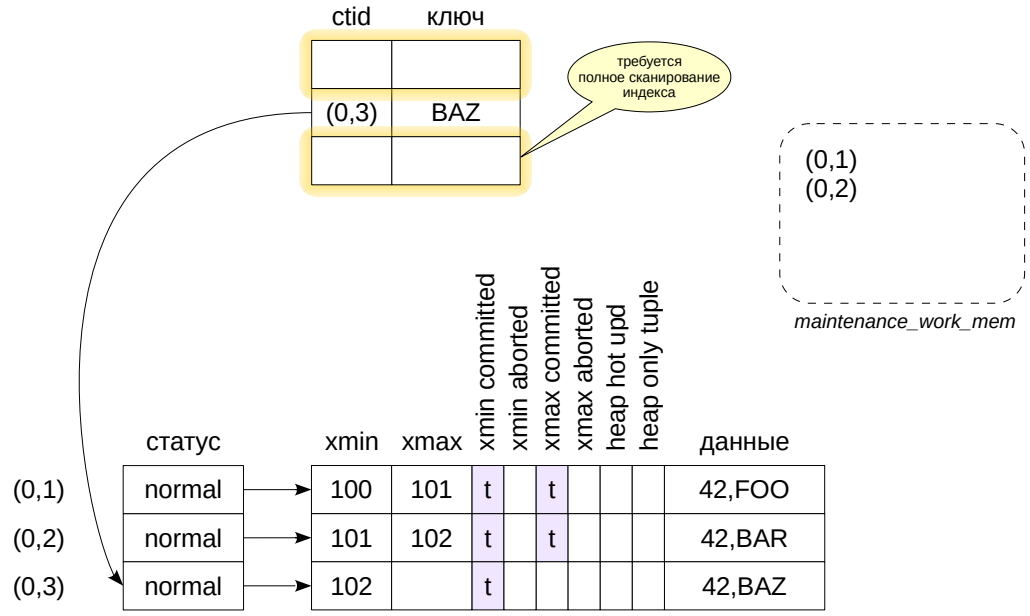
1. Сканирование таблицы



Сначала VACUUM выполняет сканирование таблицы (пропуская те страницы, которые помечены в карте видимости).

В прочитанных страницах процесс определяет ненужные версии строк и записывает их идентификаторы в специальный массив. Для этого в локальной памяти процесса VACUUM выделяется фрагмент размером *maintenance_work_mem* (или меньше, если таблица небольшая). Значение этого параметра по умолчанию — 64 Мбайт. Отметим, что эта память выделяется сразу в полном объеме, а не по мере необходимости.

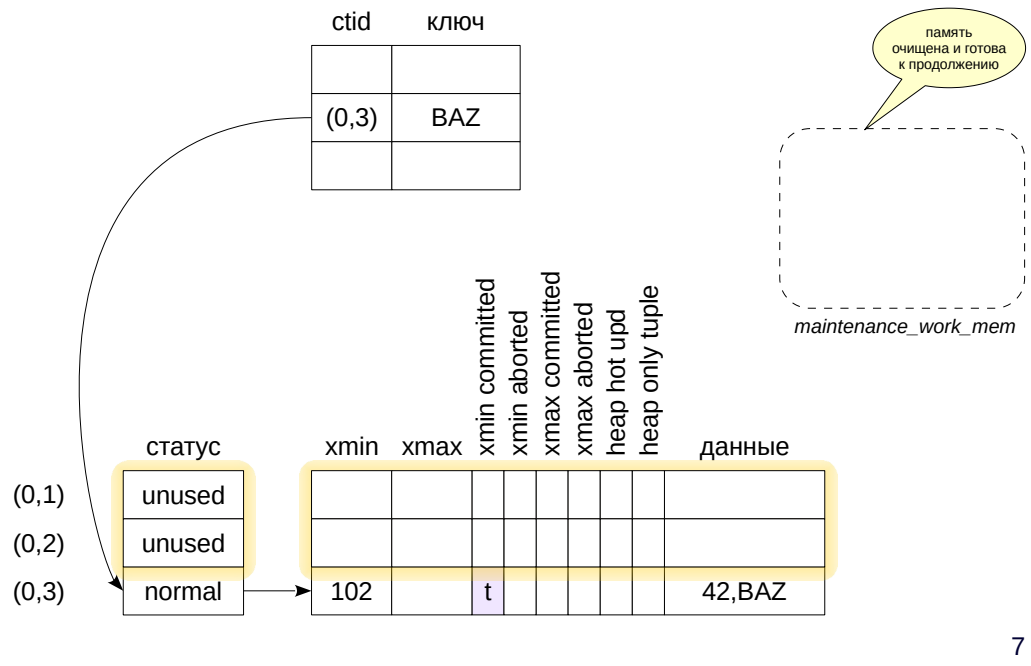
2. Очистка индексов



Когда выделенная под массив память заканчивается (или если мы уже дошли до конца таблицы), выполняется очистка индексов.

Для этого *каждый* из индексов, созданных на таблице, *полностью сканируется* в поисках записей, которые ссылаются на очищаемые версии строк. Найденные записи очищаются из индексных страниц.

3. Очистка таблицы



После этого необходимые табличные страницы повторно читаются, чтобы очистить в них ненужные версии строк и освободить указатели, на которые больше нет ссылок из индексов.

Если на первом проходе таблица не была просканирована полностью, VACUUM очищает массив в памяти и продолжает работу с этой точки.

Таким образом, если при очистке удаляется так много версий строк, что все они не помещаются в память размером *maintenance_work_mem,* все индексы будут полностью сканироваться несколько раз.
На больших таблицах это может занимать существенное время и создавать существенную нагрузку на систему. Чтобы ускорить процесс, имеет смысл либо вызывать очистку чаще (чтобы за каждый раз очищалось не очень большое количество версий строк), либо выделить больше памяти.

В конце работы очистки выполняется этап усечения таблицы. Если в конце файла образовалось достаточно много пустых страниц, они «откусываются» и освободившееся место возвращается операционной системе. Это действие требует установки исключительной блокировки. Если блокировка вызывает проблемы, этап усечения можно отключить параметром хранения *vacuum_truncate* или явно: VACUUM (TRUNCATE off). Подробнее см. модуль «Блокировки».

Мониторинг

VACUUM VERBOSE

Представление pg_stat_progress_vacuum

- полный размер таблицы
- число прочитанных страниц и число очищенных страниц
- количество уже завершенных циклов очистки индексов
- число идентификаторов версий строк, помещающихся в память, и текущее число идентификаторов в памяти
- текущая фаза очистки

Если очистка выполняется долго, может потребоваться узнать текущее состояние дел.

Для этого можно вызывать VACUUM с указанием VERBOSE — на консоль будет выводиться информация о ходе выполнения.

Кроме того, есть представление pg_stat_progress_vacuum, которое содержит всю информацию о выполняющейся в данный момент очистке. Ход очистки таблицы можно отслеживать, сравнивая число очищенных страниц (heap_blks_vacuumed) с общим количеством (heap_blks_total). Ход очистки индексов детально не отображается, но основное внимание надо обращать на количество циклов очистки (index_vacuum_count) — значение больше 1 означает, что памяти *maintenance_work_mem* не хватило для того, чтобы завершить очистку за один проход.

https://postgrespro.ru/docs/postgresql/13/progress-reporting

Обычная очистка

Создадим таблицу и индекс.

```
=> CREATE DATABASE mvcc_vacuum;
```

```
CREATE DATABASE
```

```
=> \c mvcc_vacuum
```

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

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

```
CREATE TABLE
```

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

```
CREATE INDEX
```

Как обычно, используем расширение pageinspect:

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

```
CREATE EXTENSION
```

А также представление для табличной страницы:

```
=> 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 || CASE
         WHEN (t_infomask & 256) > 0 THEN ' (c)'
         WHEN (t_infomask & 512) > 0 THEN ' (a)'
         ELSE ''
       END AS xmin,
       t_xmax || CASE
         WHEN (t_infomask & 1024) > 0 THEN ' (c)'
         WHEN (t_infomask & 2048) > 0 THEN ' (a)'
         ELSE ''
       END AS xmax,
       CASE WHEN (t_infomask2 & 16384) > 0 THEN 't' END AS hhu,
       CASE WHEN (t_infomask2 & 32768) > 0 THEN 't' END AS hot,
       t_ctid
FROM heap_page_items(get_raw_page('t',0))
ORDER BY lp;
```

```
CREATE VIEW
```

И представление для индекса:

```
=> CREATE VIEW t_id_v AS
SELECT itemoffset,
       ctid
FROM bt_page_items('t_id',1);
```

```
CREATE VIEW
```

Вставим строку в таблицу и обновим ее:

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

```
INSERT 0 1
```

```
=> UPDATE t SET id = 2;
```

```
UPDATE 1
```

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

```
=> \c mvcc_vacuum
```

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

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

```
BEGIN
```

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

```
 id
----
  2
(1 row)
```

Горизонт базы данных определяется этим снимком:

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

```
 backend_xmin
--------------
      1204316
(1 row)
```

Снова обновим строку.

```
=> UPDATE t SET id = 3;
```

```
UPDATE 1
```

Сейчас в таблице три версии строки:

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

```
 ctid  | state  |    xmin     |    xmax     | hhu | hot | t_ctid
-------+--------+-------------+-------------+-----+-----+--------
 (0,1) | normal | 1204314 (c) | 1204315 (c) |     |     | (0,2)
 (0,2) | normal | 1204315 (c) | 1204316     |     |     | (0,3)
 (0,3) | normal | 1204316     | 0 (a)       |     |     | (0,3)
(3 rows)
```

Выполним теперь очистку.

```
=> VACUUM t;
```

```
VACUUM
```

Как изменится табличная страница?

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

```
 ctid  | state  |    xmin     |    xmax     | hhu | hot | t_ctid
-------+--------+-------------+-------------+-----+-----+--------
 (0,1) | unused |             |             |     |     |
 (0,2) | normal | 1204315 (c) | 1204316 (c) |     |     | (0,3)
 (0,3) | normal | 1204316 (c) | 0 (a)       |     |     | (0,3)
(3 rows)
```

Очистка освободила одну версию строки, а вторая осталась без изменений, так как параллельная транзакция до сих пор не завершена и ее снимок активен.

В индексе — два указателя на оставшиеся версии:

```
=> SELECT * FROM t_id_v;
```

```
 itemoffset | ctid
------------+-------
          1 | (0,2)
          2 | (0,3)
(2 rows)
```

Можно попросить очистку рассказать о том, что происходит:

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

```
INFO:  vacuuming "public.t"
INFO:  "t": found 0 removable, 2 nonremovable row versions in 1 out of 1 pages
DETAIL:  1 dead row versions cannot be removed yet, oldest xmin: 1204316
There were 1 unused item identifiers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

Обратите внимание:

- 2 nonremovable row versions,

- 1 dead row version cannot be removed yet,
- oldest xmin показывает текущий горизонт.

Теперь завершим параллельную транзакцию и снова вызовем очистку.

```
=> COMMIT;
```

```
COMMIT
```

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

```
INFO:  vacuuming "public.t"
INFO:  scanned index "t_id" to remove 1 row versions
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  "t": removed 1 row versions in 1 pages
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  index "t_id" now contains 1 row versions in 2 pages
DETAIL:  1 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 1 removable, 1 nonremovable row versions in 1 out of 1 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 1204317
There were 1 unused item identifiers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

Теперь в странице осталась только последняя актуальная версия строки:

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

```
 ctid  | state  |    xmin     | xmax  | hhu | hot | t_ctid
-------+--------+-------------+-------+-----+-----+--------
 (0,1) | unused |             |       |     |     |
 (0,2) | unused |             |       |     |     |
 (0,3) | normal | 1204316 (c) | 0 (a) |     |     | (0,3)
(3 rows)
```

В индексе также только одна запись:

```
=> SELECT * FROM t_id_v;
```

```
 itemoffset | ctid
------------+-------
          1 | (0,3)
(1 row)
```

Регулирование нагрузки

Процесс чередует работу и ожидание

примерно *vacuum_cost_limit* условных единиц работы,
затем засыпает на *vacuum_cost_delay* мс

Настройки

vacuum_cost_limit = 200

vacuum_cost_delay = 0 ms

стоимость обработки:

vacuum_cost_page_hit = 1	— страница нашлась в кеше
vacuum_cost_page_miss = 10	— страница прочитана с диска
vacuum_cost_page_dirty = 20	— чистая страница стала грязной

Поскольку процесс очистки интенсивно работает с таблицами и индексами, может оказаться необходимым распределить действия во времени и тем самым сгладить пики нагрузки. Для этого существует возможность выполнять очистку порциями, чередуя работу и ожидание.

Размер порции *vacuum_cost_limit* указывается в условных единицах. Три других параметра, указанных на слайде, определяют стоимость обработки одной страницы в этих единицах в зависимости от необходимости разных действий. В лучшем случае стоимость будет равна *vacuum_cost_hit* (если страница нашлась в кеше и либо не изменилась, либо уже была грязной); в худшем случае — *vacuum_cost_page_miss* + *vacuum_cost_page_dirty* (если страница была прочитана с диска и изменилась в результате очистки).

Настройка по умолчанию фактически отключает механизм регулирования нагрузки, так как время ожидания выставлено в ноль. Это сделано из тех соображений, что если администратору пришлось запускать VACUUM вручную, он, скорее всего, хочет выполнить очистку как можно быстрее.

https://postgrespro.ru/docs/postgresql/13/runtime-config-resource#RUNTIME-CONFIG-RESOURCE-VACUUM-COST

Параллельная очистка

Сканирование и очистка таблицы — последовательно

Очистка индексов может выполняться параллельно

- каждый индекс — только одним рабочим процессом
- размер индекса должен превышать *min_parallel_index_scan_size*
- количество процессов ограничено *max_parallel_maintenance_workers* и, если указано, значением VACUUM (PARALLEL *n*)

Этапы сканирования и очистки таблицы всегда выполняются последовательно одним процессом.

Этап очистки индексов может выполняться в параллельном режиме. Это происходит, если на таблице создано несколько (больше одного) достаточно больших индексов: размер индекса должен превышать значение параметра *min_parallel_index_scan_size* (512 Кбайт по умолчанию). Тогда для каждого подходящего индекса запускается отдельный рабочий процесс.

Каждый индекс обрабатывается только одним рабочим процессом, то есть несколько процессов не могут очищать один и тот же индекс.

Количество процессов ограничено сверху значением параметра *max_parallel_maintenance_workers* и может быть дополнительно ограничено явным указанием степени параллелизма при вызове команды VACUUM (PARALLEL *n*).

Анализ

Выполняется при VACUUM ANALYZE или ANALYZE

- не конфликтует с обычной активностью в системе

Собирает статистику для планировщика

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.

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

Как и с обычной очисткой, обработка происходит в фоновом режиме и не мешает обычному использованию таблиц и индексов.

https://postgrespro.ru/docs/postgresql/13/sql-analyze

Анализ

Создадим таблицу с большим количество одинаковых строк и проиндексируем ее:

```
=> CREATE TABLE tt(s) AS SELECT 'FOO' FROM generate_series(1,1000000) AS g;

SELECT 1000000

=> CREATE INDEX ON tt(s);

CREATE INDEX
```

Планировщик ничего не знает про данные и выбирает индексный доступ, хотя читать придется всю таблицу:

```
=> EXPLAIN (costs off) SELECT * FROM tt WHERE s = 'FOO';

              QUERY PLAN
--------------------------------------
 Bitmap Heap Scan on tt
   Recheck Cond: (s = 'FOO'::text)
   ->  Bitmap Index Scan on tt_s_idx
         Index Cond: (s = 'FOO'::text)
(4 rows)
```

При анализе собирается статистика по случайной выборке строк:

```
=> ANALYZE VERBOSE tt;

INFO:  analyzing "public.tt"
INFO:  "tt": scanned 4425 of 4425 pages, containing 1000000 live rows and 0 dead rows; 30000 rows in sample, 1000000 estimated total rows
ANALYZE
```

Статистика сохраняется в системном каталоге. После этого планировщик знает, что во всех строках находится одно и то же значение, и перестанет использовать индекс:

```
=> EXPLAIN (costs off) SELECT * FROM tt WHERE s = 'FOO';

         QUERY PLAN
----------------------------
 Seq Scan on tt
   Filter: (s = 'FOO'::text)
(2 rows)
```

Полная очистка

Выполняется командой VACUUM FULL

- не совместима ни с какими операциями над таблицей, включая чтение

Полностью перестраивает таблицу и все ее индексы

- при работе потребуется дополнительное место для новых файлов
- освобожденное место возвращается операционной системе
- выполняется дольше, чем обычная очистка

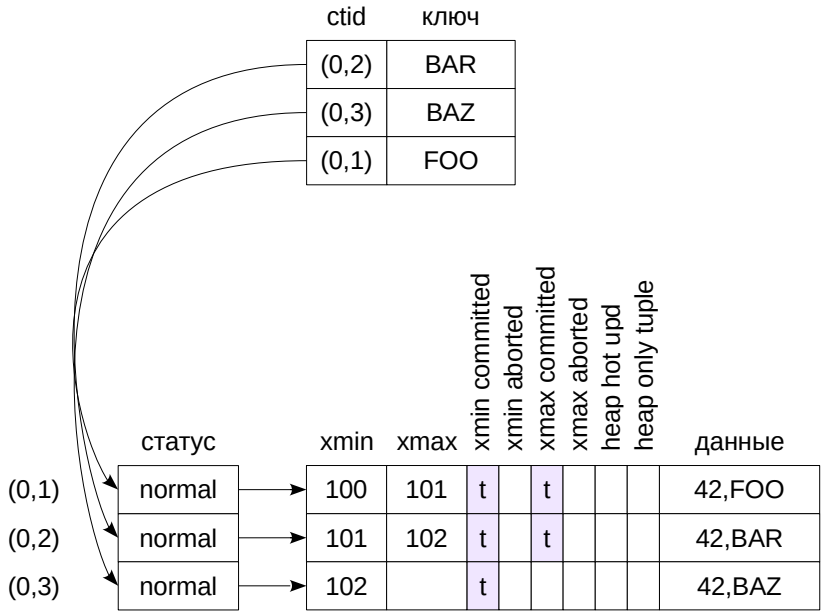
Обычная очистка не решает всех задач по освобождению места. Если таблица или индекс сильно выросли в размерах, то очистка не приведет к сокращению числа страниц (и к уменьшению файлов). Вместо этого внутри существующих страниц появятся «дыры», которые могут быть использованы для вставки новых строк или изменения существующих. Единственное 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 полную очистку. При этом таблица и все ее индексы перестраиваются полностью с нуля, а данные упаковываются максимально компактно (разумеется, с учетом *fillfactor*).

При перестройке PostgreSQL последовательно перестраивает сначала таблицу, а затем и каждый из индексов. Для них создаются новые файлы, а старые удаляются. Следует учитывать, что в процессе работы на диске потребуется дополнительное место.

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. На активно используемой системе это может быть неприемлемым; в таком случае может помочь расширение pg_repack (https://github.com/reorg/pg_repack), не входящее в поставку.

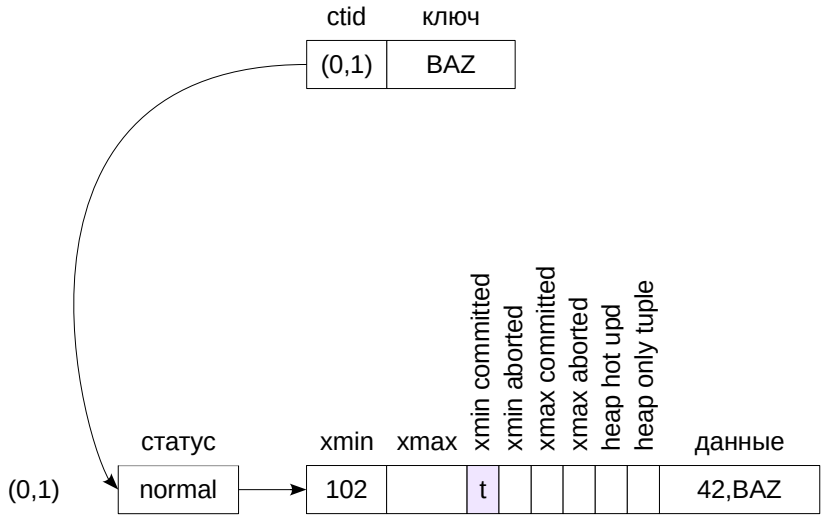
До полной очистки



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.

После полной очистки



После выполнения полной очистки содержимое таблицы и индекса полностью перестроено и физически находится в других файлах. При этом OID таблицы в системном каталоге сохраняется.

Полная очистка

Файлы, занимаемые таблицей и индексом:

```
=> SELECT pg_relation_filepath('t'), pg_relation_filepath('t_id');
```

```
 pg_relation_filepath | pg_relation_filepath
----------------------+----------------------
 base/103107/103108   | base/103107/103111
(1 row)
```

Вызываем полную очистку.

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

```
INFO:  vacuuming "public.t"
INFO:  "t": found 0 removable, 1 nonremovable row versions in 1 pages
DETAIL:  0 dead row versions cannot be removed yet.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

Таблица и индекс теперь полностью перестроены:

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

```
 ctid  | state  |    xmin     | xmax  | hhu | hot | t_ctid
-------+--------+-------------+-------+-----+-----+--------
 (0,1) | normal | 1204316 (c) | 0 (a) |     |     | (0,1)
(1 row)
```

```
=> SELECT * FROM t_id_v;
```

```
 itemoffset | ctid
------------+-------
          1 | (0,1)
(1 row)
```

Имена файлов поменялись:

```
=> SELECT pg_relation_filepath('t'), pg_relation_filepath('t_id');
```

```
 pg_relation_filepath | pg_relation_filepath
----------------------+----------------------
 base/103107/103160   | base/103107/103163
(1 row)
```

Аналоги полной очистки

CLUSTER

- полностью перестраивает таблицу и все ее индексы
- дополнительно физически упорядочивает версии строк в соответствии с одним из индексов

REINDEX

- полностью перестраивает отдельный индекс

TRUNCATE

- «опустошает» таблицу

Особенности

- все команды полностью блокируют работу с таблицей
- все команды создают новые файлы для данных

Есть несколько команд, которые работают, используя схожий механизм. Все они полностью блокируют работу с таблицей, все они удаляют старые файлы данных и создают новые.

Команда CLUSTER во всем аналогична VACUUM FULL, но дополнительно физически упорядочивает версии строк в соответствии с одним из индексов. Это дает планировщику возможность более эффективно использовать индексный доступ в некоторых случаях. Однако надо понимать, что кластеризация не поддерживается: при последующих изменениях таблицы физический порядок версий строк будет нарушаться.

Команда REINDEX перестраивает отдельный индекс на таблице. Фактически, VACUUM FULL и CLUSTER используют эту команду для того, чтобы перестроить индексы.

Команда TRUNCATE логически работает так же, как и DELETE — удаляет все табличные строки. Но DELETE, как уже было рассмотрено, помечает версии строк как удаленные, что требует дальнейшей очистки. TRUNCATE же просто создает новый, чистый файл. Это работает быстрее, но надо учитывать, что TRUNCATE заблокирует работу с таблицей на все время до конца транзакции.

Итоги

Обычная очистка освобождает место в страницах

- выполняется в фоновом режиме
- может потребоваться несколько раз сканировать индексы

Анализ собирает статистику для планировщика

- выполняется в фоновом режиме
- можно совместить с очисткой

Полная очистка перестраивает таблицу и индексы

- блокирует работу с таблицей на все время работы

Практика

1. Создайте большую таблицу с индексом. Временно уменьшите значение *maintenance_work_mem* так, чтобы потребовалось несколько проходов для очистки индекса. Проконтролируйте, запуская VACUUM VERBOSE.
2. Удалите из большой таблицы 90% случайных строк и проверьте, как изменился объем, который таблица занимает на диске, после выполнения обычной очистки.
3. Повторите п. 2 с полной очисткой.

1. Чтобы исключить срабатывание автоматической очистки (рассматривается в следующей теме), при создании таблицы укажите параметр хранения autovacuum_enabled:

```
CREATE TABLE ... WITH (autovacuum_enabled = off);
```

1. Сканирование индексов при очистке

Создадим таблицу с данными и индекс. Параметр autovacuum_enabled выключен, чтобы не срабатывала автоматическая очистка.

```
=> CREATE DATABASE mvcc_vacuum;

CREATE DATABASE

=> \c mvcc_vacuum

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

=> CREATE TABLE t(id integer) WITH (autovacuum_enabled = off);

CREATE TABLE

=> INSERT INTO t SELECT gen.id FROM generate_series(1,1000000) gen(id);

INSERT 0 1000000

=> CREATE INDEX t_id ON t(id);

CREATE INDEX
```

Уменьшаем размер памяти, выделяемой под массив идентификаторов:

```
=> ALTER SYSTEM SET maintenance_work_mem = '1MB';

ALTER SYSTEM

=> SELECT pg_reload_conf();

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

Обновляем все строки:

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

UPDATE 1000000
```

Запускаем очистку. Заодно через небольшое время в другом сеансе обратимся к pg_stat_progress_vacuum.

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

```
=> \c mvcc_vacuum

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

=> SELECT * FROM pg_stat_progress_vacuum \gx

-[ RECORD 1 ]------+------------------
pid                | 153899
datid              | 111833
datname            | mvcc_vacuum
relid              | 111834
phase              | vacuuming indexes
heap_blks_total    | 8850
heap_blks_scanned  | 3088
heap_blks_vacuumed | 2315
index_vacuum_count | 3
max_dead_tuples    | 174761
num_dead_tuples    | 174472
```

```
INFO:  vacuuming "public.t"
INFO:  scanned index "t_id" to remove 174472 row versions
DETAIL:  CPU: user: 0.13 s, system: 0.00 s, elapsed: 0.14 s
INFO:  "t": removed 174472 row versions in 772 pages
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  scanned index "t_id" to remove 174472 row versions
DETAIL:  CPU: user: 0.13 s, system: 0.00 s, elapsed: 0.13 s
INFO:  "t": removed 174472 row versions in 772 pages
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  scanned index "t_id" to remove 174472 row versions
DETAIL:  CPU: user: 0.12 s, system: 0.00 s, elapsed: 0.12 s
INFO:  "t": removed 174472 row versions in 772 pages
DETAIL:  CPU: user: 0.01 s, system: 0.00 s, elapsed: 0.05 s
INFO:  scanned index "t_id" to remove 174472 row versions
DETAIL:  CPU: user: 0.13 s, system: 0.00 s, elapsed: 0.25 s
INFO:  "t": removed 174472 row versions in 772 pages
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  scanned index "t_id" to remove 174472 row versions
DETAIL:  CPU: user: 0.11 s, system: 0.00 s, elapsed: 0.21 s
INFO:  "t": removed 174472 row versions in 772 pages
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  scanned index "t_id" to remove 127640 row versions
DETAIL:  CPU: user: 0.08 s, system: 0.00 s, elapsed: 0.09 s
INFO:  "t": removed 127640 row versions in 565 pages
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  index "t_id" now contains 1000000 row versions in 5486 pages
DETAIL:  1000000 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 1000000 removable, 1000000 nonremovable row versions in 8850 out of 8850 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 1259270
There were 0 unused item identifiers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.89 s, system: 0.00 s, elapsed: 1.25 s.
VACUUM
```

Восстановим значение измененного параметра.

```
=> ALTER SYSTEM RESET maintenance_work_mem;
```

```
ALTER SYSTEM
```

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

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

---

2. Очистка большого количества строк

Текущий размер файла данных:

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

```
 pg_size_pretty
----------------
 69 MB
(1 row)
```

Удалим 90% случайных строк. Случайность важна, чтобы в каждой странице остались какие-нибудь неудаленные строки — в противном случае очистка имеет шанс уменьшить размер файла.

```
=> DELETE FROM t WHERE random() < 0.9;
```

```
DELETE 900051
```

Объем после очистки:

```
=> VACUUM t;
```

```
VACUUM
```

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

```
 pg_size_pretty
----------------
 69 MB
(1 row)
```

Объем не изменился.

---

3. Полная очистка большого количества строк

Заново наполним таблицу.

```
=> TRUNCATE t;
```

```
TRUNCATE TABLE
```

```
=> INSERT INTO t SELECT gen.id FROM generate_series(1,1000000) gen(id);
```

```
INSERT 0 1000000
```

Текущий размер файла данных:

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

```
 pg_size_pretty
----------------
 35 MB
(1 row)
```

Обратите внимание, что в прошлый раз размер был примерно в два раза больше. Вторая половина была занята версиями строк, которые создала команда UPDATE.

Объем после удаления и полной очистки:

```
=> DELETE FROM t WHERE random() < 0.9;
```

```
DELETE 899973
```

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

```
VACUUM
```

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

```
 pg_size_pretty
----------------
 3544 kB
(1 row)
```

Объем уменьшился на 90%.