6-MAR-25

Tantor: Настройка производительности PostgreSQL 16

Практические задания

tantor

Создал: Олег Иванов

Оглавление

Авторские права

Создал: Олег Иванов

Создан: **4 марта 2025 г.**
По вопросам обучения обращайтесь: **edu@tantorlabs.ru**

Практика к главе 1

Часть 1. Стандартный тест pgbench

Образ виртуальной машины к курсу: https://disk.yandex.ru/d/APErrktFq-Gamg

1) При выполнении практик будет много раз перезапускаться экземпляр. Создайте файл с коротким именем для того, чтобы удобно перезапускать экземпляр:

```
astra@tantor:~$ su -
Password: root
root@tantor:~# echo "systemctl restart tantor-se-server-16" > /usr/local/bin/restart
root@tantor:~#  chmod 755 /usr/local/bin/restart
root@tantor:~# <ctrl+d>
logout
astra@tantor:~$ sudo restart
astra@tantor:~$
```

Вы создали командный файл и перезапустили с его помощью экземпляр.

2) Переключитесь в пользователя операционной системы postgres:

```
astra@tantor:~$ su - postgres
Password: postgres
postgres@tantor:~$
```

3) Утилита pgbench поставляется с постгрес и использует таблицы для встроенных тестов. Создайте таблицы для тестов pgbench:

```
postgres@tantor:~$ pgbench -i
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 0.10 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.43 s (drop tables 0.00 s, create tables 0.02 s, client-side generate 0.28 s, vacuum
0.07 s, primary keys 0.07 s).
```

Было создано четыре таблицы. В таблице `pgbench_accounts` 100тыс. строк, в остальных таблицах 1, 0 и 10 строк.

4) Запустите тест по умолчанию длительностью 30 секунд с выводом промежуточных результатов с интервалом в 5 секунд:

```
postgres@tantor:~$ pgbench -T 30 -P 5
starting vacuum...end.
progress: 5.0 s, 547.2 tps, lat 1.818 ms stddev 0.475, 0 failed
progress: 10.0 s, 551.6 tps, lat 1.805 ms stddev 0.230, 0 failed
progress: 15.0 s, 545.8 tps, lat 1.824 ms stddev 0.255, 0 failed
progress: 20.0 s, 554.2 tps, lat 1.797 ms stddev 0.204, 0 failed
progress: 25.0 s, 532.6 tps, lat 1.870 ms stddev 0.298, 0 failed
progress: 30.0 s, 502.6 tps, lat 1.982 ms stddev 2.024, 0 failed
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
duration: 30 s
```

```
number of transactions actually processed: 16171
number of failed transactions: 0 (0.000%)
latency average = 1.847 ms
latency stddev = 0.849 ms
initial connection time = 3.973 ms
tps = 539.088500 (without initial connection time)
```

В примере получен результат: 539 транзакций в секунду (transactions per second, tps)

5) Проверьте, как влияет удержание горизонта базы данных на TPS стандартного теста. Запустите тест, указав большой интервал времени:

```
postgres@tantor:~$ pgbench -T 10000 -P 15
```

6) Запустите второй терминал. В терминале запустите psl и начните транзакцию:

```
astra@tantor:~$ psql
postgres=# begin;
BEGIN
postgres=*#
```

7) В окне с запущенным тестом обратите внимание на показатель tps:

```
progress: 15.0 s, 541.5 tps, lat 1.838 ms stddev 0.242, 0 failed
progress: 30.0 s, 554.7 tps, lat 1.795 ms stddev 0.255, 0 failed
progress: 45.0 s, 545.9 tps, lat 1.824 ms stddev 0.279, 0 failed
```

8) Откройте транзакцию вызвав функцию, выдающую номер транзакции:

```
postgres=*# select pg_current_xact_id();
 pg_current_xact_id
--------------------
             974640
(1 row)
postgres=*#
```

9) Проверьте, что в окне с тестом tps постепенно начнет снижаться:

```
progress: 60.0 s, 484.6 tps, lat 2.055 ms stddev 0.603, 0 failed
progress: 75.0 s, 395.7 tps, lat 2.517 ms stddev 0.751, 0 failed
progress: 90.0 s, 349.1 tps, lat 2.853 ms stddev 0.793, 0 failed
progress: 105.0 s, 319.5 tps, lat 3.118 ms stddev 0.912, 0 failed
progress: 120.0 s, 308.1 tps, lat 3.237 ms stddev 0.942, 0 failed
progress: 135.0 s, 339.8 tps, lat 2.935 ms stddev 0.507, 0 failed
progress: 150.0 s, 306.3 tps, lat 3.257 ms stddev 0.478, 0 failed
progress: 165.0 s, 277.6 tps, lat 3.595 ms stddev 0.706, 0 failed
progress: 180.0 s, 264.2 tps, lat 3.777 ms stddev 0.673, 0 failed
```

За три минуты tps снизится в 1.5-2 раза. За час tps снизится в 100 раз. Мы не будем ждать час, часовой тест можно запустить в обеденное время.

10) Выполните запрос, показывающий PID активных процессов и длительность их транзакций:

```
  select age(backend_xmin), extract(epoch from (clock_timestamp()-xact_start))
secs, pid, datname database, state from pg_stat_activity where backend_xmin IS
NOT NULL OR backend_xid IS NOT NULL order by greatest(age(backend_xmin),
age(backend_xid)) desc;
    age   |    secs     |  pid   | database |        state
--------+-------------+--------+----------+--------------------
```

```
 175455 | 1425.651346 | 255554 | postgres | idle in transaction
      1 |    0.001878 | 255547 | postgres | active
      1 |    0.001213 | 255626 | postgres | active
```

Это запрос, который позволяет наблюдать за горизонтом баз данных. Запрос выдаёт число транзакций, прошедших с момента запуска транзакции серверным процессом, к которому относится строка; длительность транзакции, в первой строке самая долгая транзакция и время удержания горизонта; pid процесса, название базы данных, в которой работает транзакция; состояние транзакции. Если состояние транзакции "`idle in transaction`" это означает, что транзакция простаивает и ждёт команды от клиента. Простаивающая открытая транзакция нежелательна и таких транзакций в хорошо написанном приложении не должно быть.

В примере прошло 1425 секунд и за это время время tps в окне с тестом снизился в 7 раз:

```
progress: 1425.0 s, 71.6 tps, lat 13.964 ms stddev 1.215, 0 failed
```

11) Завершите транзакцию, удерживающую горизонт во втором окне терминала:

```
postgres=*# rollback;
ROLLBACK
postgres=#
```

12) повторите запрос в третьем окне терминала:

```
postgres=# select age(backend_xmin), extract(epoch from (clock_timestamp()-
xact_start)) secs, pid, datname database, state from pg_stat_activity where
backend_xmin IS NOT NULL OR backend_xid IS NOT NULL order by
greatest(age(backend_xmin), age(backend_xid)) desc;
 age |   secs   |  pid   | database | state
-----+----------+--------+----------+--------
   1 | 0.001050 | 255547 | postgres | active
   1 | 0.001280 | 255626 | postgres | active
```

Две транзакции это транзакции теста pgbench. Их "возраст" в числе транзакций 1. Они не удерживают горизонт.

13) Посмотрите в первом окне терминала с тестом, что tps достаточно быстро вернутся к исходным:

```
progress: 1905.0 s, 61.3 tps, lat 16.302 ms stddev 1.201, 0 failed
progress: 1920.0 s, 60.4 tps, lat 16.549 ms stddev 1.762, 0 failed
progress: 1935.0 s, 159.5 tps, lat 6.266 ms stddev 6.035, 0 failed
progress: 1950.0 s, 389.1 tps, lat 2.561 ms stddev 0.586, 0 failed
progress: 1965.0 s, 535.5 tps, lat 1.860 ms stddev 0.347, 0 failed
progress: 1980.0 s, 522.1 tps, lat 1.908 ms stddev 0.327, 0 failed
```

Вы наблюдали, что удержание горизонта в базе данных существенно снижает число транзакций, которые может обслужить СУБД. Одиночный запрос удерживает горизонт на время своей работы. Долгие запросы стоит переносить на реплики.

На уровне изоляции транзакций по умолчанию (`READ COMMITED`) транзакция начинает удерживать горизонт, как только ей будет назначен реальный номер транзакции (`xid`). Для получения реального номера использовалась функция **`pg_current_xact_id()`**. Реальный номер будет назначен при выполнении любой команды, меняющей данные. Например: INSERT, UPDATE, DELETE, CREATE, ALTER, DROP. Транзакции на уровне изоляции **`repeatable read`** удерживают горизонт с момента выполнения любой команды, в том числе SELECT и удерживают горизонт до завершения транзакции:

```
postgres=# begin transaction isolation level repeatable read;
BEGIN
```

```
postgres=*# select 1;
?column?
----------
        1
(1 row)
```

Часть 2. Использование pgbench с собственным скриптом

1) Создайте таблицу для теста выполнив команды в psql:

```
postgres=# drop table if exists t;
create table t(pk bigserial, c1 text default 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa');
insert into t select *, 'a' from generate_series(1, 100000);
alter table t add constraint pk primary key (pk);
DROP TABLE
CREATE TABLE
INSERT 0 100000
ALTER TABLE
```

2) Создайте три скрипта (командных файла):

```
postgres=# \q
postgres@tantor:~$ echo "select count(*) from t;" > count1.sql
echo "select count(1) from t;" > count2.sql
echo "select count(pk) from t;" > count3.sql
postgres@tantor:~$
```

3) Запустите по очереди три теста с созданными скриптами:

```
postgres@tantor:~$ pgbench -T 30 -P 5 -f count1.sql
pgbench (16.2)
starting vacuum...end.
progress: 5.0 s, 75.8 tps, lat 13.144 ms stddev 1.410, 0 failed
progress: 10.0 s, 74.2 tps, lat 13.495 ms stddev 1.454, 0 failed
progress: 15.0 s, 75.0 tps, lat 13.304 ms stddev 1.047, 0 failed
progress: 20.0 s, 75.2 tps, lat 13.313 ms stddev 1.267, 0 failed
progress: 25.0 s, 75.6 tps, lat 13.205 ms stddev 1.030, 0 failed
progress: 30.0 s, 76.2 tps, lat 13.125 ms stddev 0.896, 0 failed
transaction type: count1.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 2261
number of failed transactions: 0 (0.000%)
latency average = 13.263 ms
latency stddev = 1.207 ms
initial connection time = 3.788 ms
tps = 75.339419 (without initial connection time)
postgres@tantor:~$ pgbench -T 30 -P 5 -f count2.sql
pgbench (16.2)
starting vacuum...end.
progress: 5.0 s, 69.4 tps, lat 14.386 ms stddev 0.891, 0 failed
progress: 10.0 s, 67.8 tps, lat 14.707 ms stddev 1.009, 0 failed
progress: 15.0 s, 68.4 tps, lat 14.633 ms stddev 1.208, 0 failed
progress: 20.0 s, 68.4 tps, lat 14.615 ms stddev 0.909, 0 failed
progress: 25.0 s, 68.4 tps, lat 14.596 ms stddev 1.010, 0 failed
progress: 30.0 s, 67.0 tps, lat 14.940 ms stddev 0.720, 0 failed
```

```
transaction type: count2.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 2048
number of failed transactions: 0 (0.000%)
latency average = 14.645 ms
latency stddev = 0.983 ms
initial connection time = 4.314 ms
tps = 68.242454 (without initial connection time)
postgres@tantor:~$ pgbench -T 30 -P 5 -f count3.sql
pgbench (16.2)
starting vacuum...end.
progress: 5.0 s, 56.2 tps, lat 17.758 ms stddev 0.983, 0 failed
progress: 10.0 s, 55.2 tps, lat 18.095 ms stddev 0.698, 0 failed
progress: 15.0 s, 55.6 tps, lat 17.968 ms stddev 0.768, 0 failed
progress: 20.0 s, 55.4 tps, lat 18.016 ms stddev 0.765, 0 failed
progress: 25.0 s, 55.8 tps, lat 17.954 ms stddev 0.930, 0 failed
progress: 30.0 s, 55.0 tps, lat 18.125 ms stddev 1.161, 0 failed
transaction type: count3.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 1667
number of failed transactions: 0 (0.000%)
latency average = 17.985 ms
latency stddev = 0.906 ms
initial connection time = 4.129 ms
tps = 55.572129 (without initial connection time)
postgres@tantor:~$
```

4) Проанализируйте результат теста: определите какая команда работает быстрее

5) Выполните команды в psql:

```
postgres=# \timing
Timing is on.
postgres=# select count(pk) from t;
 count
--------
 100000
(1 row)

Time: 18.249 ms
postgres=# select count(1) from t;
 count
--------
 100000
(1 row)

Time: 15.223 ms
postgres=# select count(*) from t;
 count
--------
 100000
```

```
(1 row)

Time: 14.535 ms
```

Соответствует ли время выполнения команд результатам pgbench? Да, соответствует.
Соответствует ли время выполнения команд значению latency? Да, соответствует.
`latency average` измерено тестом pgbench с точностью, соответствующей `latency stddev`.

6) отключите измерение времени:

```
postgres=# \timing
Timing is off.
```

7) Выполните команды:

```
postgres=# explain analyze select count(pk) from t;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Aggregate  (cost=1791.00..1791.01 rows=1 width=8) (actual time=817.468..817.485 rows=1 loops=1)
   ->  Seq Scan on t  (cost=0.00..1541.00 rows=100000 width=8) (actual time=0.013..403.349 rows=100000
loops=>
 Planning Time: 0.047 ms
 Execution Time: 817.602 ms
(4 rows)

postgres=# explain analyze select count(1) from t;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Aggregate  (cost=1791.00..1791.01 rows=1 width=8) (actual time=800.453..800.470 rows=1 loops=1)
   ->  Seq Scan on t  (cost=0.00..1541.00 rows=100000 width=0) (actual time=0.015..399.821 rows=100000
loops=>
 Planning Time: 0.053 ms
 Execution Time: 800.589 ms
(4 rows)

postgres=# explain analyze select count(*) from t;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Aggregate  (cost=1791.00..1791.01 rows=1 width=8) (actual time=804.634..804.650 rows=1 loops=1)
   ->  Seq Scan on t  (cost=0.00..1541.00 rows=100000 width=0) (actual time=0.013..401.192 rows=100000
loops=>
 Planning Time: 0.047 ms
 Execution Time: 804.767 ms
(4 rows)

postgres=# explain analyze select count(*) from t;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Aggregate  (cost=1791.00..1791.01 rows=1 width=8) (actual time=813.472..813.489 rows=1 loops=1)
   ->  Seq Scan on t  (cost=0.00..1541.00 rows=100000 width=0) (actual time=0.015..410.861 rows=100000
loops=>
 Planning Time: 0.054 ms
 Execution Time: 813.610 ms
(4 rows)
```

Соответствует ли время выполнения команд explain analyze времени выполнения команд без explain analyze? Нет, не соответствует. Время выполнения команд ~15ms, время выполнения команд с explain analyze в этом примере ~800ms или меньше ~300ms (зависит от типа источника времени).

explain analyze с опциями по умолчанию показало время выполнения count(*) дольше, чем count(1), что расходится с предыдущими тестами. Накладные расходы на замеры времени существенно превышают время выполнения команд.

Два последних результата одной и той же команды `804.767 ms` и `813.610 ms` различаются из-за задержек в доступе к буферам в буферном кэше `401.192` и `410.861`.

Время выполнения узла Aggregate также расходится с результатами pgbench и выполнения команд без explain analyze:

```
817.485-403.349=414.136
800.470-399.821=400.649
804.650-401.192=403.458
813.489-410.861=402.628
```

Расхождение результатов в том, что count(*) медленнее, чем count(1).

8) Выполните команды, отключив в команде explain измерение времени:

```
postgres=# explain (analyze, buffers, timing off) select count(pk) from t;
                                        QUERY PLAN
-----------------------------------------------------------------------------
 Aggregate  (cost=1791.00..1791.01 rows=1 width=8) (actual rows=1 loops=1)
   Buffers: shared hit=541
   ->  Seq Scan on t  (cost=0.00..1541.00 rows=100000 width=8) (actual rows=100000 loops=1)
         Buffers: shared hit=541
 Planning:
   Buffers: shared hit=65
 Planning Time: 0.431 ms
 Execution Time: 21.388 ms
(8 rows)

postgres=# explain (analyze, buffers, timing off) select count(1) from t;
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=1791.00..1791.01 rows=1 width=8) (actual rows=1 loops=1)
   Buffers: shared hit=541
   ->  Seq Scan on t  (cost=0.00..1541.00 rows=100000 width=0) (actual rows=100000 loops=1)
         Buffers: shared hit=541
 Planning Time: 0.059 ms
 Execution Time: 18.952 ms
(6 rows)

postgres=# explain (analyze, buffers, timing off) select count(*) from t;
                                        QUERY PLAN
-----------------------------------------------------------------------------
 Aggregate  (cost=1791.00..1791.01 rows=1 width=8) (actual rows=1 loops=1)
   Buffers: shared hit=541
   ->  Seq Scan on t  (cost=0.00..1541.00 rows=100000 width=0) (actual rows=100000 loops=1)
         Buffers: shared hit=541
 Planning:
   Buffers: shared hit=3
 Planning Time: 0.058 ms
 Execution Time: 16.357 ms
(8 rows)
```

С опцией **`timing off`** результаты команды `explain` соответствуют результатам тестов pgbench и выполнению запросов без `explain`.

Параметр buffers на результат не влияет, он иллюстрирует то, что все блоки таблицы находились в буферном кэше (`shared hit=541`) при выполнении всех трёх команд.

У команды explain есть опция **`settings`**, которая выдаст строку со значениями параметров, которые:

a) влияют на планировщик

b) имеют значения, отличные от значений по умолчанию.

Если такие параметры отсутствуют, то параметр settings не влияет на вывод команды explain.

Особенности измерения времени, настройки источника времени, измерения его скорости будут рассмотрены в следующих главах курса.

Какие параметры “влияют на планировщик”? Это декларируется при добавлении параметра конфигурации разработчиками PostgreSQL или создателями библиотек расширений. Пример:

```
postgres=# select pg_settings_get_flags('work_mem');
 pg_settings_get_flags
-----------------------
 {EXPLAIN}
(1 row)
```

Если в результатах есть , то это означает, что параметр конфигурации задекларирован как “влияющий на планировщик”. Таких параметров около 60.

У параметров конфигурации могут быть следующие флаги:

EXPLAIN (примерно 60 параметров) параметры с этим флагом включаются в команды EXPLAIN (SETTINGS);
NO_SHOW_ALL (параметров с таким флагом нет) исключаются из команд SHOW ALL;
NO_RESET (3 параметра) не поддерживают команды RESET;
NO_RESET_ALL (3 параметра) исключаются из команд RESET ALL;
NOT_IN_SAMPLE (50 параметров) автоматически не включаются в файл postgresql.conf.sample, на основе которого утилитой initdb создаётся файл параметров кластера
RUNTIME_COMPUTED (5 параметров) значения параметра не устанавливаются, а вычисляются при запуске экземпляра.

Список параметров с флагами можно получить командой:

```
postgres=# select name, pg_settings_get_flags(name) from pg_settings where array_length(pg_settings_get_flags(name),1)>0;
```

Список вычисляемых параметров:

```
postgres=# select name, context, pg_settings_get_flags(name) from pg_settings where pg_settings_get_flags(name)::text like '%RUNTIME%';
                name                | context  |      pg_settings_get_flags
------------------------------------+----------+-------------------------------
 data_checksums                     | internal | {NOT_IN_SAMPLE,RUNTIME_COMPUTED}
 data_directory_mode                | internal | {NOT_IN_SAMPLE,RUNTIME_COMPUTED}
 shared_memory_size                 | internal | {NOT_IN_SAMPLE,RUNTIME_COMPUTED}
 shared_memory_size_in_huge_pages   | internal | {NOT_IN_SAMPLE,RUNTIME_COMPUTED}
 wal_segment_size                   | internal | {NOT_IN_SAMPLE,RUNTIME_COMPUTED}
(5 rows)
```

Список параметров, не сбрасывающихся в исходное значение:

```
postgres=# select name, context, pg_settings_get_flags(name) from pg_settings where pg_settings_get_flags(name)::text like '%RESET%';
           name           | context |          pg_settings_get_flags
--------------------------+---------+-------------------------------------
 transaction_deferrable   | user    | {NO_RESET,NO_RESET_ALL,NOT_IN_SAMPLE}
 transaction_isolation    | user    | {NO_RESET,NO_RESET_ALL,NOT_IN_SAMPLE}
 transaction_read_only    | user    | {NO_RESET,NO_RESET_ALL,NOT_IN_SAMPLE}
(3 rows)
```

9) Выполните команду, созданную на основе теста на языке plpgsql: https://gist.github.com/lukaseder/2611212b23ba40d5f828c69b79214a0e/ используемого в статье https://blog.jooq.org/whats-faster-count-or-count1/ :

```
DO $$
DECLARE
  v_ts TIMESTAMP;
  v_repeat CONSTANT INT := 100;
  rec RECORD;
  run INT[];
  stmt INT[];
  elapsed DECIMAL[];
  min_elapsed DECIMAL;
  i INT := 1;
BEGIN
  FOR r IN 1..5 LOOP
    v_ts := clock_timestamp();
    FOR i IN 1..v_repeat LOOP
      FOR rec IN (
        select count(*) from t
      ) LOOP
```

```
          NULL;
        END LOOP;
      END LOOP;
      run[i] := r;
      stmt[i] := 1;
      elapsed[i] := (EXTRACT(EPOCH FROM CAST(clock_timestamp() AS TIMESTAMP)) -
EXTRACT(EPOCH FROM v_ts));
      i := i + 1;
      v_ts := clock_timestamp();
      FOR i IN 1..v_repeat LOOP
        FOR rec IN (
          select count(1) from t
        ) LOOP
          NULL;
        END LOOP;
      END LOOP;
      run[i] := r;
      stmt[i] := 2;
      elapsed[i] := (EXTRACT(EPOCH FROM CAST(clock_timestamp() AS TIMESTAMP)) -
EXTRACT(EPOCH FROM v_ts));
      i := i + 1;
      v_ts := clock_timestamp();
      FOR i IN 1..v_repeat LOOP
        FOR rec IN (
          select count(pk) from t
        ) LOOP
          NULL;
        END LOOP;
      END LOOP;
      run[i] := r;
      stmt[i] := 3;
      elapsed[i] := (EXTRACT(EPOCH FROM CAST(clock_timestamp() AS TIMESTAMP)) -
EXTRACT(EPOCH FROM v_ts));
      i := i + 1;
    END LOOP;
    SELECT min(t.elapsed)
    INTO min_elapsed
    FROM unnest(elapsed) AS t(elapsed);
    FOR i IN 1..array_length(run, 1) LOOP
      RAISE INFO 'RUN %, Statement %: %', run[i], stmt[i], CAST(elapsed[i] /
min_elapsed AS DECIMAL(10, 5));
    END LOOP;
  END$$;

  INFO:  RUN 1, Statement 1: 1.00764
  INFO:  RUN 1, Statement 2: 1.12819
  INFO:  RUN 1, Statement 3: 1.38278
  INFO:  RUN 2, Statement 1: 1.01305
  INFO:  RUN 2, Statement 2: 1.12725
  INFO:  RUN 2, Statement 3: 1.37496
  INFO:  RUN 3, Statement 1: 1.00552
  INFO:  RUN 3, Statement 2: 1.13360
  INFO:  RUN 3, Statement 3: 1.38406
  INFO:  RUN 4, Statement 1: 1.00269
  INFO:  RUN 4, Statement 2: 1.13924
  INFO:  RUN 4, Statement 3: 1.38148
  INFO:  RUN 5, Statement 1: 1.00000
  INFO:  RUN 5, Statement 2: 1.14320
  INFO:  RUN 5, Statement 3: 1.41113
  DO
```

Результат соответствует результатам теста pgbench.

Это был пример использования программы на языке plpgsql для тестирования времени выполнения трёх команд. Использование pgbench более просто и даёт больше информации: времени выполнения команд (latency) и стандартного отклонения времени выполнения.

Часть 3. Использование утилиты sysbench

1) Создайте таблицы, которые использует sysbench выполнив команду:

```
postgres@tantor:~$ sysbench --db-driver=pgsql --pgsql-port=5432 --pgsql-db=postgres --pgsql-user=postgres --pgsql-password=postgres --tables=1 --table_size=100000 /usr/share/sysbench/oltp_read_only.lua prepare
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)

Creating table 'sbtest1'...
Inserting 100000 records into 'sbtest1'
Creating a secondary index on 'sbtest1'...
```

Создана одна таблица sbtest1 со 100тыс. строк.

2) Файлы тестов sysbench пишутся на языке lua. В директории `/usr/share/sysbench/` находятся стандартные тесты. Запустите тест только для чтения с названием `oltp_read_only.lua`:

```
postgres@tantor:~$ sysbench --db-driver=pgsql --pgsql-port=5432 --pgsql-db=postgres --pgsql-user=postgres --pgsql-password=postgres --threads=10 --time=15  --report-interval=5 /usr/share/sysbench/oltp_read_only.lua run
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 10
Report intermediate results every 5 second(s)
Initializing random number generator from current time

Initializing worker threads...
Threads started!

[ 5s ] thds: 10 tps: 584.03 qps: 9364.62 (r/w/o: 8194.56/0.00/1170.05) lat (ms,95%): 28.67 err/s: 0.00 reconn/s: 0.00
[ 10s ] thds: 10 tps: 592.77 qps: 9480.12 (r/w/o: 8294.58/0.00/1185.54) lat (ms,95%): 28.16 err/s: 0.00 reconn/s: 0.00
[ 15s ] thds: 10 tps: 477.21 qps: 7632.70 (r/w/o: 6678.48/0.00/954.21) lat (ms,95%): 51.02 err/s: 0.00 reconn/s: 0.00
SQL statistics:
    queries performed:
        read:                            115934
        write:                           0
        other:                           16562
        total:                           132496
    transactions:                        8281   (550.22 per sec.)
    queries:                             132496 (8803.46 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          15.0492s
    total number of events:              8281

Latency (ms):
         min:                                    3.70
         avg:                                   18.12
         max:                                  178.98
         95th percentile:                       30.81
         sum:                               150092.73
```

```
Threads fairness:
    events (avg/stddev):           828.1000/63.26
    execution time (avg/stddev):   15.0093/0.01
```

Число транзакций в секунду 550.22.

Остальные показатели не удобны. Число транзакций 8281 и **events:** зависят от длительности теста. Другие показатели с трудом поддаются анализу. Например, Latency 95th percentile более точно соответствует реальности, чем avg, который вряд ли имеет смысл, так как измерения сильно отклоняющиеся от среднего (178.98) стоило исключить из расчёта. Задержки могут быть связаны с планировщиком или активностью операционной системы. В примере использовалось 10 потоков, в виртуальной машине меньше ядер и задержки связаны с планировщиком операционной системы. Измерения, выполненные сразу после запуска утилиты, авторам утилиты стоило исключать из подсчета.

Часть 4. Использование HammerDB

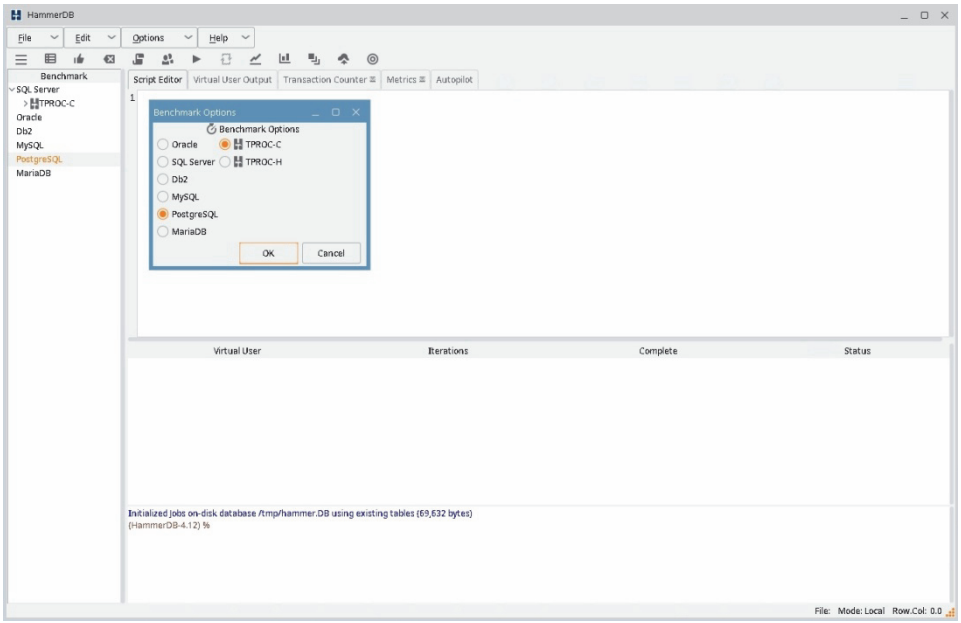
1) Запустите терминал под пользователем astra и установите HammerDB:

```
astra@tantor:~$ wget https://github.com/TPC-Council/HammerDB/releases/download/v4.12/HammerDB-4.12-Linux-x64-installer.run
- 'HammerDB-4.12-Linux-x64-installer.run' saved [12665959/12665959]
astra@tantor:~$ chmod +x HammerDB-4.12-Linux-x64-installer.run
astra@tantor:~$ ./HammerDB-4.12-Linux-x64-installer.run
```

Установите в директорию по умолчанию /home/astra/HammerDB-4.12
После инсталляции утилита запустится. Утилиту можно запустить повторно командой:

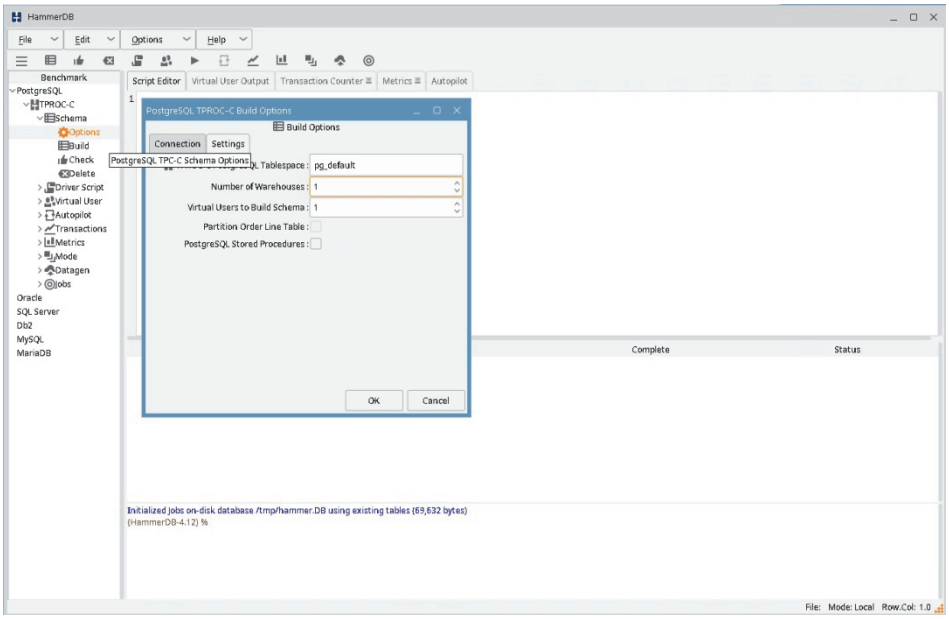
```
astra@tantor:~$ cd /home/astra/HammerDB-4.12
astra@tantor:~/HammerDB-4.12$ ./hammerdb &
```

2) После запуска утилиты в её окне кликните на PostgreSQL. В появившемся окне "Benchmark Options" оставьте значения PostgreSQL и TPROC-C. Нажмите в окне кнопку OK. В окне подтверждения также нажмите кнопку OK.

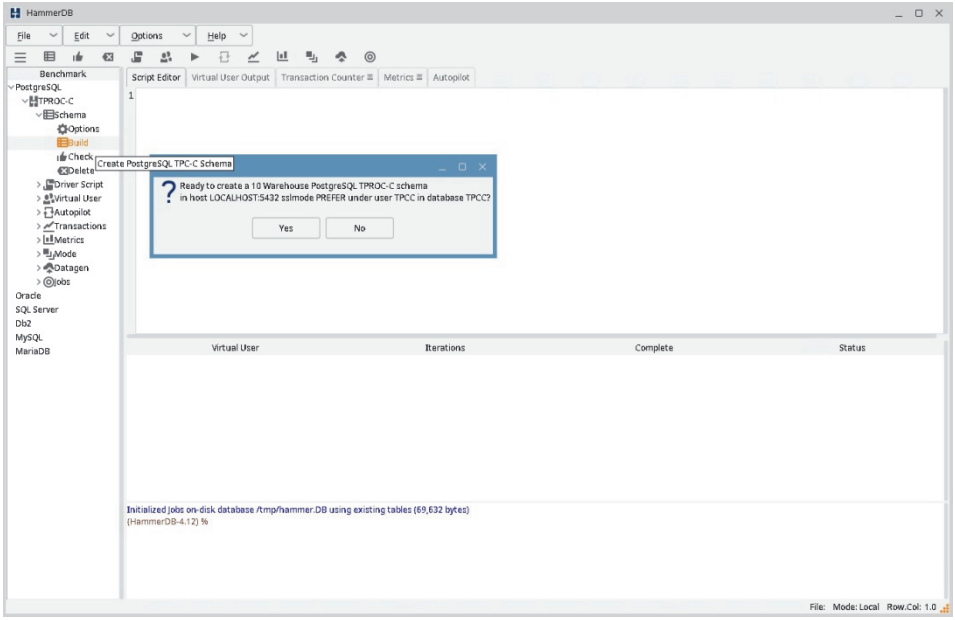


3) Раскрыв меню PostgreSQL->TPROC-C->Schema и кликнув на Options перейдите в закладку Settings и установите Number of warehouses (число складов) = 2. По умолчанию установлено значение 1. Для 10 складов потребуется 1Гб. Установите число Virtual Users to build schema в

двойное число ядер процессоров минус один, например 7 при числе ядер 4. Нажмите OK, настройка сохранится. При реальном тестировании потребуется значительно большее число складов. Однако, небольшое число складов позволит симулировать конкуренцию за блокировки между сессиями.

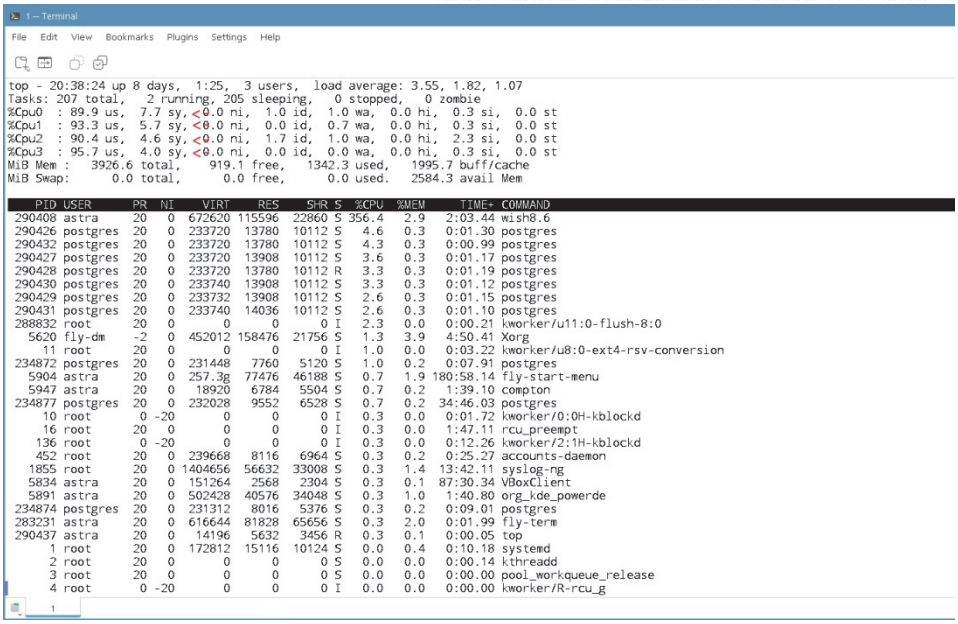


4) Кликните в меню PostgreSQL->TPROC-C->Schema на Build. Появится окно с предупреждением, что будет создана база данных tpcc и в ней таблицы с данными для теста TPC-C.



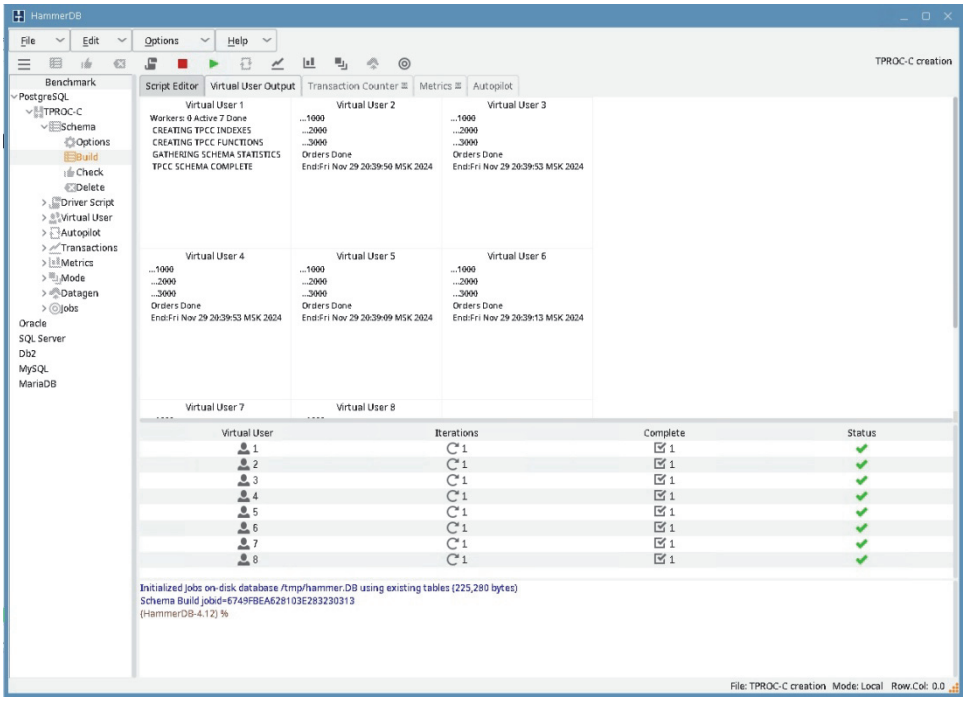
Время на создание несколько минут. Можно посмотреть нагрузку на процессора пока идёт создание таблиц командой top. Чтобы показывалась нагрузка по ядрам нужно нажать 1. Для выхода из утилиты top нужно нажать клавишу q.

```
astra@tantor:~/HammerDB-4.12$ top
```

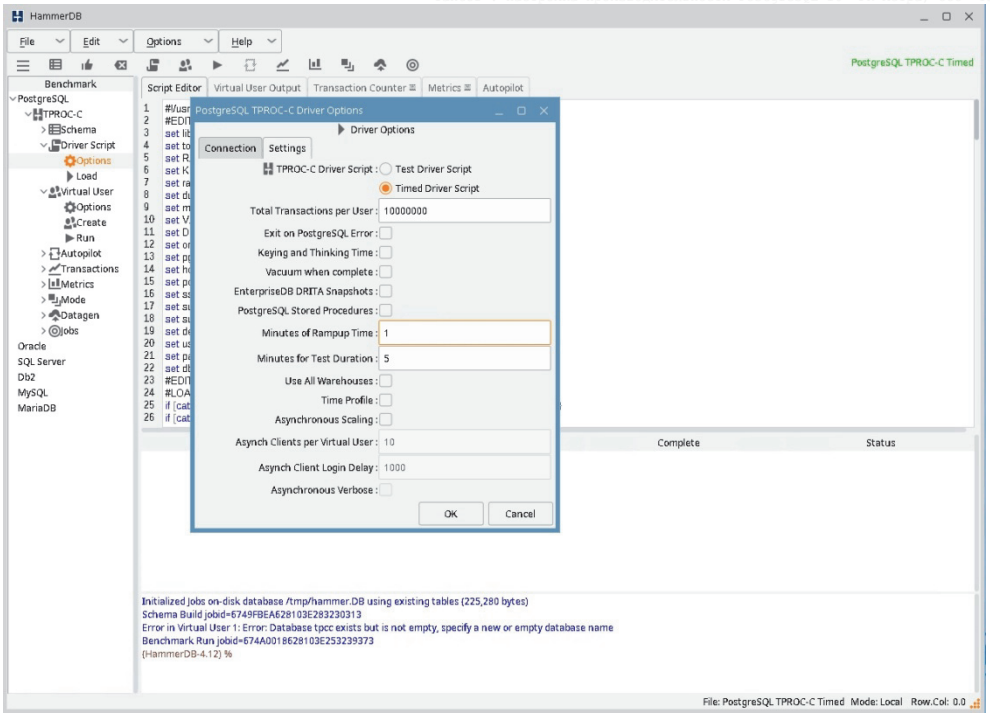


В окне утилиты top видно, что нагрузка на четыре ядра близка к 100%. Пропорция USR/SYS: `%Cpu0 89.9us + 7.7 sy`.

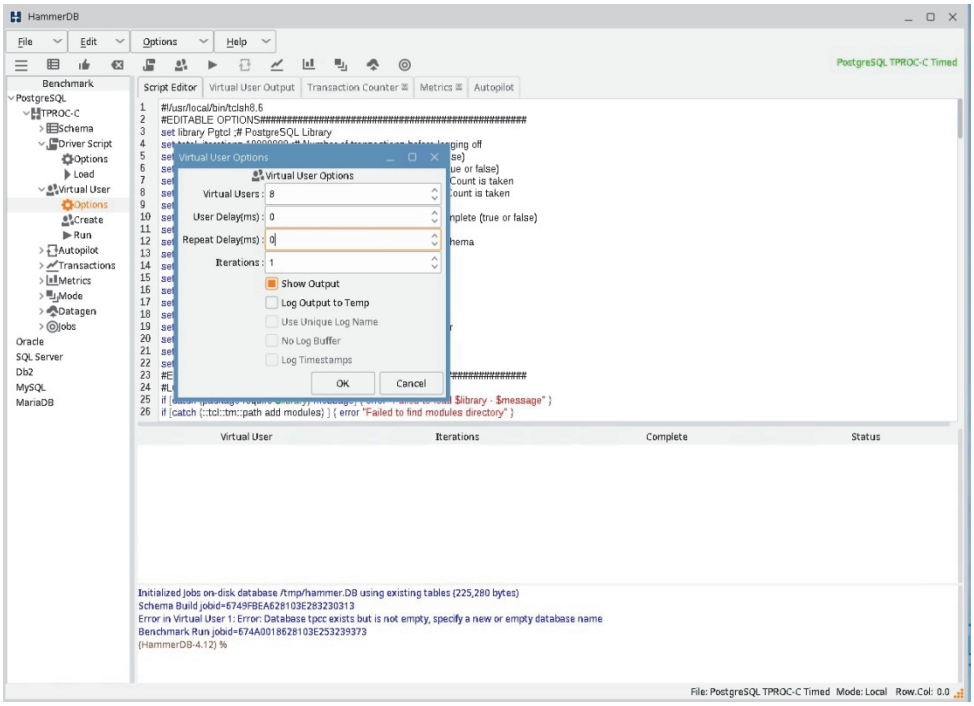
На то, что таблицы созданы укажет сообщение TPCC SCHEMA COMPLETE в окне Virtual User 1.



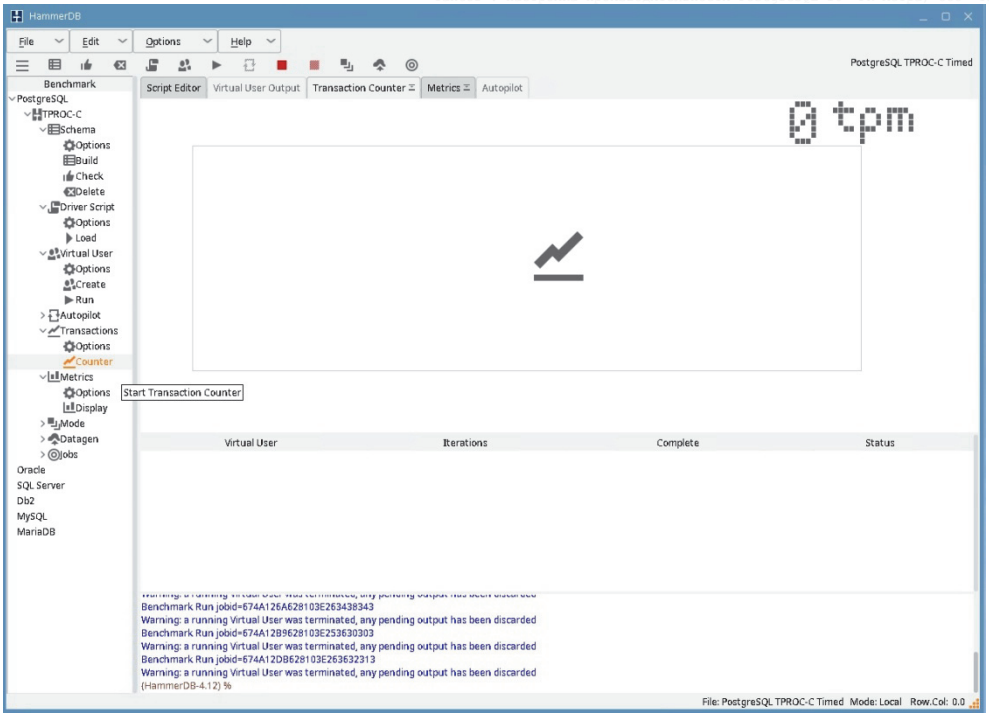
5) Нажмите на красный квадрат на toolbar утилиты. Выберите в меню PostgreSQL->TPROC-C-> Driver Script -> Options. В открывшемся окне перейдите в закладку Settings и укажите Minutes of Rampup Time 0. Это время "разогрева", то есть постепенного увеличения нагрузки на СУБД. Мы установили небольшое число клиентов, которые не нагрузят СУБД и задержка не нужна. Minutes of Test Duration 5, это длительность теста, если тест не будет остановлен раньше.



6) В окне PostgreSQL->TPROC-C-> Virtual User -> Options установите User Delay(ms)=0, Repeat Delay(ms)=0. При реальном тестировании рекомендуют устанавливать число пользователей в 10 раз меньше, чем число складов.

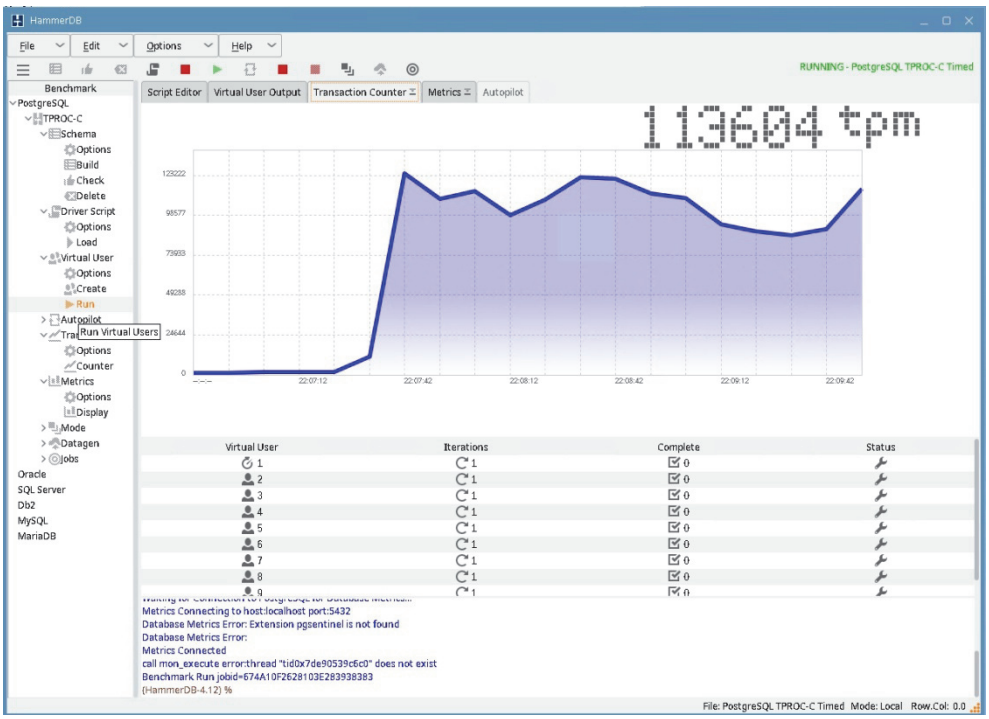


7) Перед запуском теста для наблюдения за tpm нужно запустить Transaction Counter. Для этого кликните в меню на PostgreSQL->TPROC-C-> Transactions -> Counter. Запустится счетчик транзакций.



7) Кликните на PostgreSQL->TPROC-C-> Virtual User -> Create и затем на PostgreSQL->TPROC-C-> Virtual User -> Run

Кликните на закладку Transaction Counter. Появится окно с метрикой теста, называющейся tpm.



В примере tpm=113604. График метрики tpm колеблется, это не неточности, а следствие динамической нагрузки, соответствующей правилам теста. Уменьшение tpm на десятки процентов возникает из-за удержания горизонта базы данных автоанализом. Через несколько минут после запуска теста tpm может снизиться на треть. Это происходит из-за запуска автоанализа по таблице `order_line` и продолжающегося до двух минут. На снижение tpm также влияет выполнение контрольной точки.

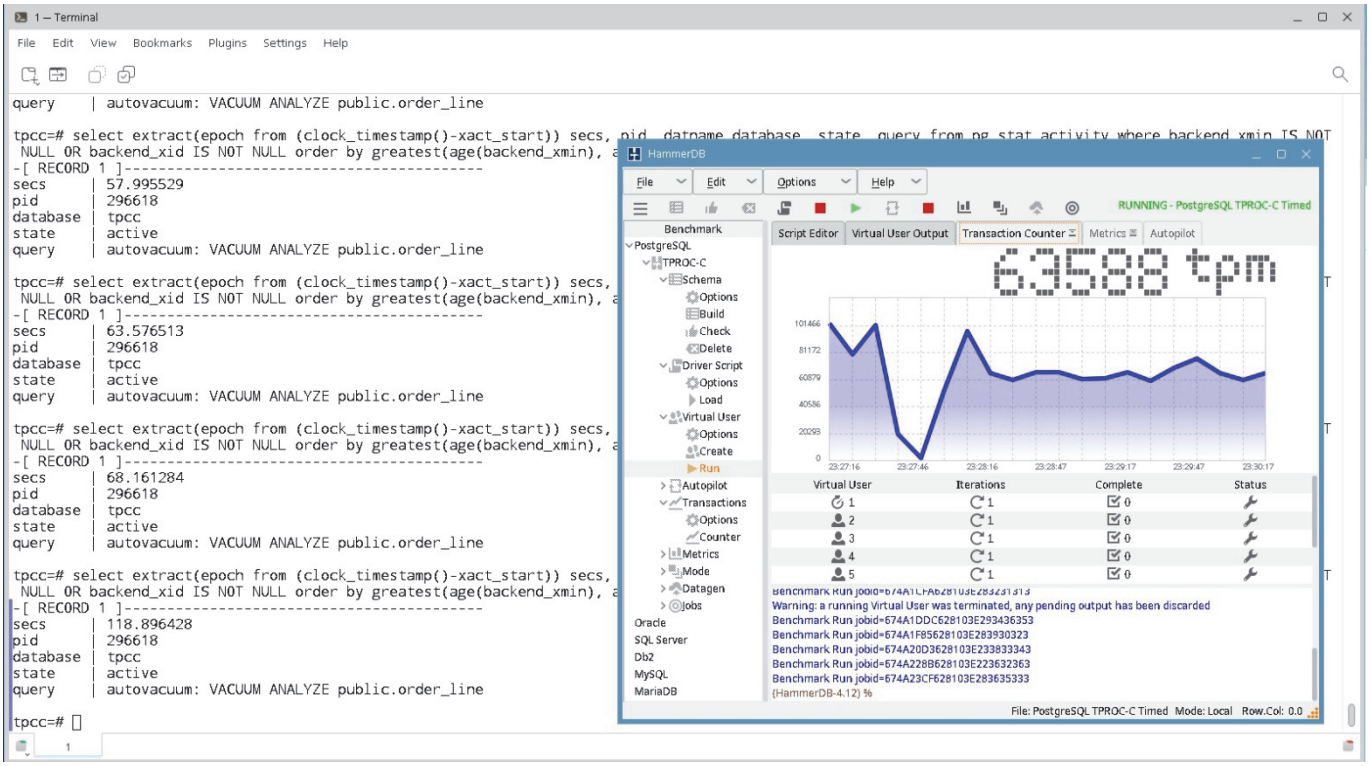
Это пример, когда периодический сбор статистики не то, что бесполезен, а вреден. Если данные статистики после пересборов не меняются, статистику бессмысленно "актуализировать".

```
postgres=# \c tpcc
You are now connected to database "tpcc" as user "postgres".
tpcc=# select extract(epoch from (clock_timestamp()-xact_start)) secs, pid,
datname database, state, query from pg_stat_activity where backend_xmin IS NOT
NULL OR backend_xid IS NOT NULL order by greatest(age(backend_xmin),
age(backend_xid)) desc limit 1 \gx
-[ RECORD 1 ]----------------------------------------
secs     | 118.896428
pid      | 296618
database | tpcc
state    | active
query    | autovacuum: VACUUM ANALYZE public.order_line
```

В отсутствие нагрузки анализ таблицы длится (на СУБД с параметрами конфигурации по умолчанию) 27 секунд:

```
tpcc=# VACUUM (ANALYZE, verbose)  public.order_line;
INFO:  vacuuming "tpcc.public.order_line"
INFO:  finished vacuuming "tpcc.public.order_line": index scans: 1
pages: 0 removed, 186118 remain, 166506 scanned (89.46% of total)
tuples: 446332 removed, 15377120 remain, 0 are dead but not yet removable, oldest xmin: 10827114
removable cutoff: 10827114, which was 0 XIDs old when operation ended
frozen: 79603 pages from table (42.77% of total) had 1825159 tuples frozen
index scan needed: 51544 pages from table (27.69% of total) had 597993 dead item identifiers removed
index "order_line_i1": pages: 81640 in total, 0 newly deleted, 0 currently deleted, 0 reusable
avg read rate: 83.907 MB/s, avg write rate: 48.399 MB/s
buffer usage: 170193 hits, 296131 misses, 170812 dirtied
WAL usage: 384641 records, 164101 full page images, 1262224255 bytes
system usage: CPU: user: 6.17 s, system: 4.05 s, elapsed: 27.57 s
INFO:  analyzing "public.order_line"
INFO:  "order_line": scanned 30000 of 186118 pages, containing 2492677 live rows and 0 dead rows; 30000 rows
in sample, 15464402 estimated total rows
VACUUM
```

tpm снизился от 101000 до 63588 из-за того, что автоанализ удерживал горизонт базы данных.



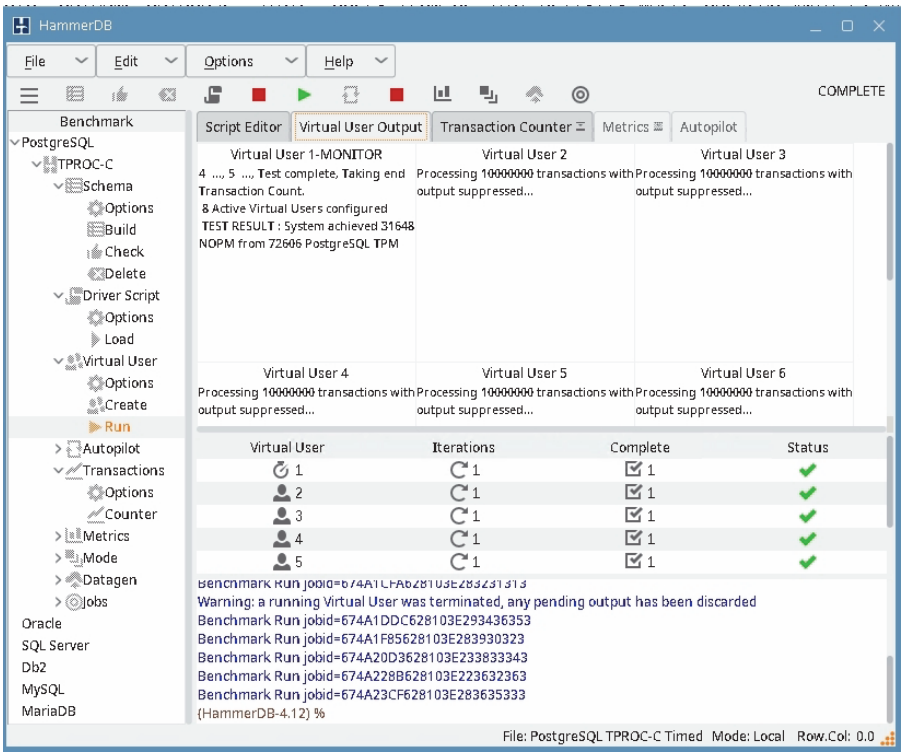
8) Ждать остановки теста по времени не нужно. Остановите тест, нажав на левый красный квадрат (Destroy Virtual users) на toolbar. На правый красный квадрат (Stop Transaction Counter) нажимать не нужно, иначе остановится счётчик транзакций. В окне PostgreSQL->TPROC-C-> Virtual User -> Options установите User Delay(ms)=500, Repeat Delay(ms)=500. Эти параметры не

сильно влияют на tpm, они влияют на колебания графика tpm. На tpm влияет число клиентов (сессий с СУБД), называемых Virtual Users.

Меняя параметры в Virtual User -> Options, нажав на Virtual User -> Run и перейдя в закладку Transaction Counter можно наблюдать как изменится tpm. Максимальный tpm достигается при числе клиентов равному числу ядер процессоров.

9) После завершения тестирования закройте окно HammerDB и удалите базу данных tpcc, которая после недолгого тестирования разрастётся в несколько раз. В таблицы тест выставляет строки и они увеличиваются.

По завершении теста в окне Virtual User 1 будет показан результат теста:



В примере результат tpms (PostgreSQL TPM)=72606 и NOPM (New Orders per minute)=31648.

Со временем tpcm при повторных тестах будут уменьшаться. Из-за этого тест не подходит для использования при проверке того как повлияют изменения конфигурации кластера на производительность, так как tpm не только не стабилен, но и уменьшается со временем. Для стабильности результата придется удалять базу данных tpcc и создавать заново. Тест может использоваться для долговременного однократного тестирования в целях сравнения с другими СУБД того же или другого типа. При тестировании все параметры должны быть одинаковы, особенно размер базы данных (число складов), длительность теста, число клиентов.

Тест TPC-C удобен тем, что при его выполнении можно проводить анализ работы экземпляра доступными инструментами (расширениями, обращаясь к представлениям со статистиками) и выявляя наиболее эффективные инструменты и метрики. Это следствие более сложных команд, чем у простых нагрузочных тестов pgbench TPC-B.

Графическое приложение HammerDB удобно для таких целей.

```
postgres=# \c tpcc
You are now connected to database "tpcc" as user "postgres".
tpcc=# select pg_size_pretty(pg_database_size('tpcc'));
 pg_size_pretty
----------------
 2682 MB
(1 row)
```

После создания таблиц база данных занимала 1Гб.

```
tpcc=# vacuum full;
VACUUM
tpcc=# select pg_size_pretty(pg_database_size('tpcc'));
 pg_size_pretty
----------------
 2331 MB
(1 row)

postgres=# drop database tpcc;
DROP DATABASE
postgres=#
```

Часть 5. Использование приложения Go-TPC

1) Установите go-tpc:

```
postgres@tantor:~$ mkdir gotpc
mkdir: cannot create directory ‘gotpc’: File exists
postgres@tantor:~$ cd gotpc
postgres@tantor:~/gotpc$ wget https://raw.githubusercontent.com/pingcap/go-tpc/master/install.sh
- ‘install.sh’ saved [2020/2020]
postgres@tantor:~/gotpc$ chmod +x install.sh
postgres@tantor:~/gotpc$ ./install.sh
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100 4984k  100 4984k    0     0  4370k      0  0:00:01  0:00:01 --:--:-- 23.2M
Detected shell: bash
Shell profile:  /var/lib/postgresql/.bash_profile
/var/lib/postgresql/.bash_profile has been modified to to add go-tpc to PATH
open a new terminal or source /var/lib/postgresql/.bash_profile to use it
Installed path: /var/lib/postgresql/.go-tpc/bin/go-tpc
================================================
Have a try:     go-tpc tpcc
================================================
postgres@tantor:~/gotpc$
```

2) Перейдите в директорию утилиты и запустите утилиту с опцией **tpcc prepare**, которая создаст базу данных gotpcc и создаст в этой базе данных объекты для теста типа **TPC-C**:

```
postgres@tantor:~/gotpc$ cd $HOME/.go-tpc/bin
postgres@tantor:~/.go-tpc/bin$ ./go-tpc tpcc prepare -d postgres -U postgres -p 'postgres' -D gotpcc -H
127.0.0.1 -P 5432 --conn-params sslmode=disable
maxprocs: Leaving GOMAXPROCS=4: CPU quota undefined
creating table warehouse
creating table district
creating table customer
creating index idx_customer
creating table history
creating index idx_h_w_id
creating index idx_h_c_w_id
creating table new_order
creating table orders
creating index idx_order
creating table order_line
creating table stock
creating table item
load to item
load to warehouse in warehouse 1
load to stock in warehouse 1
load to district in warehouse 1
load to warehouse in warehouse 2
load to stock in warehouse 2
load to district in warehouse 2
load to warehouse in warehouse 3
load to stock in warehouse 3
load to district in warehouse 3
load to warehouse in warehouse 4
load to stock in warehouse 4
load to district in warehouse 4
load to warehouse in warehouse 5
load to stock in warehouse 5
load to district in warehouse 5
load to warehouse in warehouse 6
load to stock in warehouse 6
```

```
load to district in warehouse 6
load to warehouse in warehouse 7
load to stock in warehouse 7
load to district in warehouse 7
load to warehouse in warehouse 8
load to stock in warehouse 8
load to district in warehouse 8
load to warehouse in warehouse 9
load to stock in warehouse 9
load to district in warehouse 9
load to warehouse in warehouse 10
load to stock in warehouse 10
load to district in warehouse 10
load to customer in warehouse 1 district 1
load to history in warehouse 1 district 1
..
```

Для каждого типа теста этой утилиты лучше создавать таблицы в отдельной базе данных, так как у части таблиц названия пересекаются. В драйвере утилиты нет локальных подсоединений, подсоединение должно быть через сетевой интерфейс, поэтому указываются все параметры соединения.

По умолчанию утилита создаёт 10 warehouses. Они создаются в течение 4 минут. Число складов можно поменять, указав параметр `--warehouses 4`.

3) Примерно через 3-4 минуты вы увидите в логе утилиты строки "**`begin to check`** `warehouse 1 at condition 3.3.2.4`". Утилита начала проверять данные после загрузки. Проверка идёт долго и не имеет смысла. Она нужна для тех СУБД, которые теряют данные, PostgreSQL не теряет данные. Нажмите на клавиатуре <ctrl+c>, чтобы прервать проверку:

```
load to new_order in warehouse 10 district 10
load to order_line in warehouse 10 district 10
begin to check warehouse 1 at condition 3.3.2.4
begin to check warehouse 1 at condition 3.3.2.5
begin to check warehouse 1 at condition 3.3.2.6
begin to check warehouse 1 at condition 3.3.2.7
begin to check warehouse 1 at condition 3.3.2.8
begin to check warehouse 1 at condition 3.3.2.9
begin to check warehouse 1 at condition 3.3.2.10
begin to check warehouse 1 at condition 3.3.2.10
^C
Got signal [interrupt] to exit.
check prepare failed, err check warehouse 1 at condition 3.3.2.10 failed exec SELECT count(*)
      FROM (  SELECT  c.c_id, c.c_d_id, c.c_w_id, c.c_balance c1,
                    (SELECT sum(ol_amount) FROM orders, order_line
                        WHERE OL_W_ID=O_W_ID
                          AND OL_D_ID = O_D_ID
                          AND OL_O_ID = O_ID
                          AND OL_DELIVERY_D IS NOT NULL
                          AND O_W_ID=?
                          AND O_D_ID=c.C_D_ID
                          AND O_C_ID=c.C_ID) sm, (SELECT  sum(h_amount)  from  history
                                                    WHERE H_C_W_ID=?
                                                      AND H_C_D_ID=c.C_D_ID
                                                      AND H_C_ID=c.C_ID) smh
                   FROM customer c
                  WHERE  c.c_w_id = ? ) t
   WHERE c1<>sm-smh failed pq: canceling statement due to user request
Finished
```

Размер базы данных, которую создала утилита 1Гб:

```
postgres=# select pg_size_pretty(pg_database_size('gotpcc'));
 pg_size_pretty
----------------
 1028 MB
(1 row)
```

4) Запустите тест tpcc:

```
postgres@tantor:~/.go-tpc/bin$ ./go-tpc tpcc run -d postgres -U postgres -p
'postgres' -D gotpcc -H 127.0.0.1 -P 5432 --conn-params sslmode=disable
maxprocs: Leaving GOMAXPROCS=4: CPU quota undefined
```

```
[Current] DELIVERY - Takes(s): 9.9, Count: 61, TPM: 369.4, Sum(ms): 879.3, Avg(ms): 14.4, 50th(ms): 14.2,
90th(ms): 16.8, 95th(ms): 17.8, 99th(ms): 21.0,
99.9th(ms): 27.3, Max(ms): 27.3
[Current] NEW_ORDER - Takes(s): 9.9, Count: 714, TPM: 4310.5, Sum(ms): 5931.1, Avg(ms): 8.3, 50th(ms): 8.4,
90th(ms): 11.0, 95th(ms): 12.6, 99th(ms): 17.8
, 99.9th(ms): 26.2, Max(ms): 29.4
[Current] ORDER_STATUS - Takes(s): 9.9, Count: 61, TPM: 369.7, Sum(ms): 104.9, Avg(ms): 1.7, 50th(ms): 2.1,
90th(ms): 2.1, 95th(ms): 2.6, 99th(ms): 3.1, 9
9.9th(ms): 5.2, Max(ms): 5.2
[Current] PAYMENT - Takes(s): 10.0, Count: 697, TPM: 4197.6, Sum(ms): 2448.8, Avg(ms): 3.5, 50th(ms): 3.7,
90th(ms): 4.2, 95th(ms): 4.7, 99th(ms): 7.9, 99
.9th(ms): 8.4, Max(ms): 13.1
[Current] STOCK_LEVEL - Takes(s): 9.9, Count: 65, TPM: 395.9, Sum(ms): 277.1, Avg(ms): 4.3, 50th(ms): 3.7,
90th(ms): 4.7, 95th(ms): 5.8, 99th(ms): 24.1, 9
9.9th(ms): 41.9, Max(ms): 41.9
```

На консоль будет выводиться текущая статистика TPM.

5) Примерно через минуту нажмите на клавиатуре <ctrl+c>, чтобы прервать тест. Утилита выдаст статистику с результатом **tpmC**:

```
^C
Got signal [interrupt] to exit.
Finished
[Summary] DELIVERY - Takes(s): 182.8, Count: 1329, TPM: 436.3, Sum(ms): 17163.3, Avg(ms): 12.9, 50th(ms):
12.6, 90th(ms): 14.7, 95th(ms): 16.8, 99th(ms): 22.0, 99.9th(ms): 29.4, Max(ms): 30.4
[Summary] NEW_ORDER - Takes(s): 182.8, Count: 14194, TPM: 4659.0, Sum(ms): 106451.6, Avg(ms): 7.5, 50th(ms):
7.3, 90th(ms): 9.4, 95th(ms): 10.5, 99th(ms): 15.7, 99.9th(ms): 22.0, Max(ms): 41.9
[Summary] ORDER_STATUS - Takes(s): 182.8, Count: 1288, TPM: 422.9, Sum(ms): 2306.0, Avg(ms): 1.8, 50th(ms):
2.1, 90th(ms): 2.6, 95th(ms): 2.6, 99th(ms): 4.2, 99.9th(ms): 8.9, Max(ms): 10.0
[Summary] PAYMENT - Takes(s): 182.8, Count: 13431, TPM: 4407.9, Sum(ms): 46553.2, Avg(ms): 3.5, 50th(ms):
3.7, 90th(ms): 4.2, 95th(ms): 4.7, 99th(ms): 7.9, 99.9th(ms): 14.2, Max(ms): 19.9
[Summary] STOCK_LEVEL - Takes(s): 182.7, Count: 1262, TPM: 414.4, Sum(ms): 3707.7, Avg(ms): 2.9, 50th(ms):
3.1, 90th(ms): 3.7, 95th(ms): 4.2, 99th(ms): 8.9, 99.9th(ms): 24.1, Max(ms): 41.9
tpmC: 4659.0, tpmTotal: 10340.5, efficiency: 3622.8%
```

6) Удалите базу данных, созданную утилитой:

```
postgres@tantor:~/.go-tpc/bin$ psql -c "drop database gotpcc;"
DROP DATABASE
```

7) В отличие от HammerDB утилита go-tpc работает не на полной скорости, а выполняет правила теста TPC-C, работая с задержками и постепенно увеличивая нагрузку. В утилите есть ещё два теста: TPC-H и собственный тест утилиты tpc-ch (CH-benCHmark смешанная нагрузка из тестов TPC-C и TPC-H). Тест подходит для СУБД, которые используются в общих целях со смешанной нагрузкой. PostgreSQL редко используется со смешанной нагрузкой, так как для долгих(аналитических) запросов используют физические реплики.

Тест TPC-H для хранилищ данных и состоит только из читающих команд. Результат теста TPC-H более стабилен, позволяет проводить сравнение. Недостаток в том, что размер баз для официальных тестов довольно большой.

Создание базы данных занимает примерно 10 минут. Размер базы данных 1660Мб.

Выполнять следующие пункты этой практики не обязательно. Команды и их результаты приведены для ознакомления. Вы можете использовать эти команды как справочник или если хотите закрепить навыки использования утилиты.

```
postgres@tantor:~/.go-tpc/bin$ ./go-tpc tpch prepare -d postgres -U postgres -p
'postgres' -D gotpch -H 127.0.0.1 -P 5432 --conn-params sslmode=disable
 maxprocs: Leaving GOMAXPROCS=4: CPU quota undefined
creating nation
creating region
creating part
creating supplier
creating partsupp
creating customer
creating orders
creating lineitem
generating nation table
generate nation table done
generating region table
generate region table done
generating customers table
```

```
generate customers table done
generating suppliers table
generate suppliers table done
generating part/partsupplier tables
generate part/partsupplier tables done
generating orders/lineitem tables
generate orders/lineitem tables done
Finished
```

Запуск теста TPC-H:

```
postgres@tantor:~/.go-tpc/bin$ ./go-tpc tpch run -d postgres -U postgres -p 'postgres' -D gotpch -H 127.0.0.1
-P 5432 --conn-params sslmode=disable
maxprocs: Leaving GOMAXPROCS=4: CPU quota undefined
[Current] Q1: 6.21s
[Current] Q2: 0.91s
[Current] Q3: 1.31s
[Current] Q4: 0.44s
[Current] Q5: 0.44s
[Current] Q10: 1.24s
[Current] Q11: 0.30s
[Current] Q12: 1.17s
[Current] Q13: 1.04s
[Current] Q14: 1.17s
[Current] Q6: 0.77s
[Current] Q7: 0.64s
[Current] Q8: 0.97s
[Current] Q9: 3.05s
[Current] Q15: 0.84s
[Current] Q16: 0.57s
[Current] Q17: 1199.91s
[Current] Q18: 5.27s
[Current] Q19: 1.04s
```

Запрос Q17 в тесте долгий, он выполняется 3 часа(11999.91s). Утилита показывает максимальное значение `1199.91s` для длинных запросов, в утилите программная ошибка.

Пример мониторинга запроса Q17:

```
postgres=# select extract(epoch from (clock_timestamp()-xact_start)) secs, pid,
datname database, state, query from pg_stat_activity where backend_xmin IS NOT
NULL OR backend_xid IS NOT NULL order by greatest(age(backend_xmin),
age(backend_xid)) desc limit 1 \gx
-[ RECORD 1 ]----------------------------------------------
secs     | 6434.701066
pid      | 316066
database | gotpch
state    | active
query    |                                                  +
         | /*PLACEHOLDER*/ select                           +
         |         sum(l_extendedprice) / 7.0 as avg_yearly+
         | from                                             +
         |         lineitem,                                +
         |         part                                     +
         | where                                            +
         |         p_partkey = l_partkey                    +
         |         and p_brand = 'Brand#43'                 +
         |         and p_container = 'LG PACK'              +
         |         and l_quantity < (                       +
         |                 select                           +
         |                         0.2 * avg(l_quantity)    +
         |                 from                             +
         |                         lineitem                 +
         |                 where                            +
         |                         l_partkey = p_partkey    +
         |         );                                       +
         |
```

В запросе есть коррелированный подзапрос. Коррелированные подзапросы тяжелы для всех СУБД. Например, на специализированной под OLAP СУБД clickhouse запросы теста TPC-H Q17 и Q20 не работают "As of October 2024, **the query does not work out-of-the box due to correlated subqueries**. Corresponding issue: https://github.com/ClickHouse/ClickHouse/issues/6697" (https://clickhouse.com/docs/en/getting-started/example-datasets/tpch).

Запрос Q20 работает на PostgreSQL с параметрами по умолчанию 7,6 часов, запрос Q17 работет около 4 часов. Долгие запросы интересны для оптимизации выполнения запросов:

выяснения причин долгого выполнения. К оптимизации запросов стоит переходить после настройки работы экземпляра PostgreSQL. План выполнения этого запроса:

```
postgres=# \c gotpch
You are now connected to database "gotpch" as user "postgres".
gotpch=# explain select sum(l_extendedprice) / 7.0 as avg_yearly from lineitem,
part where p_partkey = l_partkey and p_brand = 'Brand#43' and p_container = 'LG
PACK' and l_quantity < ( select 0.2 * avg(l_quantity) from lineitem where
l_partkey = p_partkey);
                                    QUERY PLAN
-------------------------------------------------------------------------------------
Aggregate  (cost=2204613.84..2204613.86 rows=1 width=32)
 -> Hash Join  (cost=6625.16..2204608.92 rows=1970 width=8)
      Hash Cond: (lineitem.l_partkey = part.p_partkey)
      Join Filter: (lineitem.l_quantity < (SubPlan 1))
      ->  Seq Scan on lineitem  (cost=0.00..184745.33 rows=6001233 width=21)
      ->  Hash  (cost=6622.70..6622.70 rows=197 width=8)
          ->  Gather  (cost=1000.00..6622.70 rows=197 width=8)
                Workers Planned: 2
                ->  Parallel Seq Scan on part  (cost=0.00..5603.00 rows=82 width=8)
                      Filter:((p_brand = 'Brand#43'::bpchar) AND (p_container = 'LG PACK'::bpchar))
      SubPlan 1
        ->  Aggregate  (cost=199748.49..199748.51 rows=1 width=32)
              ->  Seq Scan on lineitem lineitem_1  (cost=0.00..199748.41 rows=31 width=5)
                    Filter: (l_partkey = part.p_partkey)
JIT:
  Functions: 24
  Options: Inlining true, Optimization true, Expressions true, Deforming true
(17 rows)
```

В таблицах 6млн. и 200т. строк, что не так много:

```
gotpch=# select count(*) from lineitem;
  count
---------
 6001215
(1 row)

gotpch=# select count(*) from part;
 count
--------
 200000
(1 row)
```

Долгий запрос нужно прервать, набрав на клавиатуре <ctrl+c> и затем удалить базу данных **gotpch**:

```
postgres@tantor:~/.go-tpc/bin$ psql -c "drop database gotpch;"
```

В составе приложения go-tpc есть три теста. Для создания базы данных для третьего теста (смешанного) используются параметр "**ch**", вместо "tpcc" и "tpch":

```
go-tpc ch prepare -d postgres -U postgres -p 'postgres' -D gotpcch -H 127.0.0.1
-P 5432 --conn-params sslmode=disable
```

Практика к главе 2

Часть 1. Запуск экземпляра с огромными страницами

1) Найдите список экземпляров PostgreSQL:

```
postgres@tantor:~$ ps -ef | grep /postgres
postgres    1423       1  0 Nov27 ? 00:00:06 /opt/tantor/db/16/bin/postgres -D /var/lib/postgresql/tantor-se-16/data
postgres    1432       1  0 Nov27 ? 00:00:03 /usr/lib/postgresql/15/bin/postgres -D /var/lib/postgresql/15/main -c config_file=/etc/postgresql/15/main/postgresql.conf
```

В примере два экземпляра: СУБД Тантор и PostgreSQL Astralinux.

2) Посмотрите, сколько памяти использует экземпляр PostgreSQL и какой размер у огромных страниц:

```
postgres@tantor:~$ cat /proc/1432/status | grep VmPeak
VmPeak:   222268 kB
postgres@tantor:~$ cat /proc/meminfo | grep Huge
AnonHugePages:      2048 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:               0 kB
```

Максимальное потребление памяти экземпляром: `222268 kB` / `2048 kB`=109 страниц по 2Мб (2048). Огромные страницы могут использоваться разделяемым пулом (параметр **`shared_buffers`**) и параллельными процессами (параметр **`min_dynamic_shared_memory`**). Форки могут использовать и под другие структуры памяти. Пиковое потребление памяти на загруженном экземпляре показывает использование памяти всеми структурами.

Значение в строке `AnonHugePages:` отличное от нуля укажет на то, что какой-то процесс явно запросил использование THP системным вызовом `madvise()`. PostgreSQL не использует такой системный вызов.

Выполните команду free:

```
postgres@tantor:~$ free
               total        used        free      shared  buff/cache   available
Mem:         4020796     1137792      764984      158692     2556828     2883004
Swap:              0           0           0
```

3) Остановите экземпляр и проверьте сколько огромных страниц он может максимально выделить в соответсвии с его параметрами конфигурации:

```
postgres@tantor:~$ sudo systemctl stop postgresql
postgres@tantor:~$ /usr/lib/postgresql/15/bin/postgres -c config_file=/etc/postgresql/15/main/postgresql.conf -D /var/lib/postgresql/15/main -C shared_memory_size_in_huge_pages
70
```

4) Выделите память под 100 страниц и проверьте, что они выделены:

```
postgres@tantor:~$ sudo sysctl -w vm.nr_hugepages=100
vm.nr_hugepages = 100
```

```
postgres@tantor:~$ cat /proc/meminfo | grep Huge
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
HugePages_Total:     100
HugePages_Free:       27
HugePages_Rsvd:        1
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:          204800 kB
```

Значение `HugePages_Free: = 27` указывает, что свободных страниц недостаточно под разделяемый пул размера 128Мб. Если у вас страниц больше 70, то такого количества достаточно. Если их меньше 80, то выделите память под 300 страниц и проверьте что свободных страниц больше 80:

```
postgres@tantor:~$ sudo sysctl -w vm.nr_hugepages=300
vm.nr_hugepages = 300

postgres@tantor:~$ cat /proc/meminfo | grep Huge
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
HugePages_Total:     218
HugePages_Free:      145
HugePages_Rsvd:        1
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:          446464 kB
```

5) Выполните команду free:

```
postgres@tantor:~$ free
               total        used        free      shared  buff/cache   available
Mem:         4020796     1331220      571396      158696     2556992     2689576
Swap:              0           0           0
```

В выводе команды free показатели: used увеличился на 193428 байт, free уменьшился на 193 588 байт, available уменьшился на 193 428 байт.

6) Перезапустите службу postgresql:

```
postgres@tantor:~$ sudo systemctl restart postgresql
postgres@tantor:~$ cat /proc/meminfo | grep Huge
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
HugePages_Total:     218
HugePages_Free:      126
HugePages_Rsvd:       73
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:          446464 kB
```

7) Получите список процессов, которые используют огромные страницы размером 2Мб:

```
postgres@tantor:~$ sudo grep "KernelPageSize:     2048 kB"
/proc/[[:digit:]]*/smaps | awk {'print $1'} | cut -d "/" -f3 | sort | uniq
112647
112649
112650
```

```
112652
112653
112654
```

8) Сравните с номерами (PID) процессов экземпляра Astralinux PostgresSQL:

```
postgres@tantor:~$ ps -ef | grep 15/
postgres  112647       1  0 17:55 ?        00:00:00 /usr/lib/postgresql/15/bin/postgres -D
/var/lib/postgresql/15/main -c config_file=/etc/postgresql/15/main/postgresql.conf
postgres  112648  112647  0 17:55 ?        00:00:00 postgres: 15/main: logger
postgres  112649  112647  0 17:55 ?        00:00:00 postgres: 15/main: checkpointer
postgres  112650  112647  0 17:55 ?        00:00:00 postgres: 15/main: background writer
postgres  112652  112647  0 17:55 ?        00:00:00 postgres: 15/main: walwriter
postgres  112653  112647  0 17:55 ?        00:00:00 postgres: 15/main: autovacuum launcher
postgres  112654  112647  0 17:55 ?        00:00:00 postgres: 15/main: logical replication
launcher
```

Номера процессов совпадают. Экземпляр использует огромные страницы.

На то, что экземпляр стал использовать огромные страницы также укажет увеличение значения **HugePages_Rsvd** или уменьшение **HugePages_Free**. Из-за неудобства проверки по этим метрикам операционной системы то ли страницы зарезервированы экземпляром, то ли уже используются и экземпляром ли или другими процессами операционной системы, в 17 версии PostgreSQL добавили параметр `huge_pages_status`, который показывает, используются ли огромные страницы экземпляром или не используются.

Если хочется быть уверенным что экземпляр использует огромные страницы, можно установить значение параметра `huge_pages=on`. При таком значении параметра экземпляр не запустится, если не выделит огромные страницы.

Огромные страницы под использование параллельными процессами резервируются только при запуске экземпляра и только параметром `min_dynamic_shared_memory`. Если огромные страницы не зарезервированы или их будет недостаточно, то параллельные процессы станут выделять и использовать обычные страницы.

9) Посмотрите размер и тип огромных страниц, которые использует процесс postgres:

```
postgres@tantor:~$ cat /proc/112647/smaps_rollup | grep tlb
Shared_Hugetlb:    12288 kB
Private_Hugetlb:       0 kB
```

Процесс postgres использует разделяемые огромные страницы.

Часть 2. Изменение значения `oom_score`

1) Выполните команду:

```
postgres@tantor:~$ for PID in $(pgrep "postgres"); do awk '/Pss/ {PSS+=$2} END
{ getline cmd < "/proc/'$PID'/cmdline"; sub("\0", " ", cmd); getline oom <
"/proc/'$PID'/oom_score"; printf "%.0f -%s-> %s (PID %s) \n", PSS, oom, cmd,
'$PID'}' /proc/$PID/smaps; done|sort -n -r
28579 -0-> /opt/tantor/db/16/bin/postgres -D/var/lib/postgresql/tantor-se-16/data (PID 112360)
5629 -668-> postgres: walwriter   (PID 112364)
5166 -70-> /usr/lib/postgresql/15/bin/postgres -D/var/lib/postgresql/15/main-
cconfig_file=/etc/postgresql/15/main/postgresql.conf (PID 112647)
3207 -668-> postgres: autovacuum launcher   (PID 112365)
2945 -668-> postgres: 15/main: autovacuum launcher   (PID 112653)
2933 -668-> postgres: autoprewarm leader   (PID 112366)
2685 -668-> postgres: 15/main: logical replication launcher   (PID 112654)
2644 -668-> postgres: checkpointer   (PID 112361)
2618 -668-> postgres: logical replication launcher   (PID 112368)
2366 -668-> postgres: pg_wait_sampling collector   (PID 112367)
2157 -668-> postgres: background writer   (PID 112362)
1497 -667-> postgres: 15/main: logger   (PID 112648)
1479 -667-> postgres: 15/main: checkpointer   (PID 112649)
```

```
1468 -667-> postgres: 15/main: walwriter   (PID 112652)
1438 -667-> postgres: 15/main: background writer   (PID 112650)
```

Команда показывает значение **oom_score** процессов экземпляров PostgreSQL.
У основного процесса экземпляра СУБД Тантор **oom_score=0**. У экземпляра Astralinux PostgreSQL **oom_score=70**.

Уменьшение oom_score было выполнено путем редактирования файла службы. Посмотрите содержимое файла:

```
postgres@tantor:~$ cat /usr/lib/systemd/system/tantor-se-server-16.service
[Unit]
Description=Tantor Special Edition database server 16
Documentation=https://www.postgresql.org/docs/16/static/
After=syslog.target
After=network.target

[Service]
Type=forking

User=postgres
Group=postgres

LimitNOFILE=infinity
LimitNOFILESoft=infinity

# Where to send early-startup messages from the server (before the logging options of postgresql.conf take
effect)
# This is normally controlled by the global default set by systemd
# StandardOutput=syslog

# Disable OOM kill on the postmaster
OOMScoreAdjust=-1000
# ... but allow it still to be effective for child processes
# (note that these settings are ignored by Postgres releases before 9.5)
Environment=PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
Environment=PG_OOM_ADJUST_VALUE=0

# Maximum number of seconds pg_ctl will wait for postgres to start.  Note that PGSTARTTIMEOUT should be less
than
# TimeoutSec value.
Environment=PGSTARTTIMEOUT=270
Environment=PGDATA=/var/lib/postgresql/tantor-se-16/data

ExecStartPre=/opt/tantor/db/16/bin/postgresql-check-db-dir ${PGDATA}
ExecStart=/opt/tantor/db/16/bin/pg_ctl start -D ${PGDATA} -s -w -t ${PGSTARTTIMEOUT}
ExecStop=/opt/tantor/db/16/bin/pg_ctl stop -D ${PGDATA} -s -m fast
ExecReload=/opt/tantor/db/16/bin/pg_ctl reload -D ${PGDATA} -s

# Give a reasonable amount of time for the server to start up/shut down. Ideally, the timeout for starting
# PostgreSQL server should be handled more nicely by pg_ctl in ExecStart, so keep its timeout smaller than
this
# value.
TimeoutSec=300

[Install]
WantedBy=multi-user.target
```

В файле написано, что OOM kill запрещается для процесса postmaster, но работает для остальных процессов экземпляра. Если нужно уменьшить значение oom_score, то можно изменить в этом файле значение параметра **PG_OOM_ADJUST_VALUE**.

3) Пункт опционален, его можно не выполнять. Понизьте oom_score на 200. Для этого отредактируйте файл службы и установите значение **PG_OOM_ADJUST_VALUE=-300**:

```
postgres@tantor:~$ su -
Password: root
root@tantor:~# cd /usr/lib/systemd/system/
root@tantor:~# mcedit tantor-se-server-16.service
отредактируйте и затем сохраните файл клавишей F2 ENTER закройте редактор клавишей F10
root@tantor:/usr/lib/systemd/system# cat tantor-se-server-16.service | grep OOM
# Disable OOM kill on the postmaster
OOMScoreAdjust=-1000
Environment=PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
```

```
   Environment=PG_OOM_ADJUST_VALUE=-300
   root@tantor:/usr/lib/systemd/system# systemctl daemon-reload
   root@tantor:/usr/lib/systemd/system# restart
   root@tantor:/usr/lib/systemd/system# for PID in $(pgrep "postgres"); do awk '/Pss/ {PSS+=$2} END
{ getline cmd < "/proc/'$PID'/cmdline"; sub("\0", " ", cmd); getline oom <
"/proc/'$PID'/oom_score"; printf "%.0f -%s-> %s (PID %s) \n", PSS, oom, cmd, '$PID'}'
/proc/$PID/smaps; done|sort -n -r
   33392 -0-> /opt/tantor/db/16/bin/postgres -D/var/lib/postgresql/tantor-se-16/data (PID 119419)
   4993 -70-> /usr/lib/postgresql/15/bin/postgres -D/var/lib/postgresql/15/main-
cconfig_file=/etc/postgresql/15/main/postgresql.conf (PID 112647)
   3232 -468-> postgres: autovacuum launcher   (PID 119424)
   2865 -668-> postgres: 15/main: autovacuum launcher   (PID 112653)
   2770 -468-> postgres: logical replication launcher   (PID 119427)
   2648 -668-> postgres: 15/main: logical replication launcher   (PID 112654)
   2533 -468-> postgres: background writer   (PID 119421)
   2523 -468-> postgres: pg_wait_sampling collector   (PID 119426)
   2019 -667-> postgres: 15/main: checkpointer   (PID 112649)
   1777 -468-> postgres: autoprewarm leader   (PID 119425)
   1562 -468-> postgres: checkpointer   (PID 119420)
   1517 -667-> postgres: 15/main: walwriter   (PID 112652)
   1480 -667-> postgres: 15/main: background writer   (PID 112650)
   1475 -667-> postgres: 15/main: logger   (PID 112648)
   1463 -468-> postgres: walwriter   (PID 119423)
```

У всех процессов экземпляра tantor oom_score уменьшился на 200. После редактирования файла службы перечитали конфигурацию командой **`systemctl daemon-reload`** и рестартовали экземпляр.

4) Проверьте, что ksm отключён:

```
postgres@tantor:~$ cat /proc/vmstat | grep ksm
ksm_swpin_copy 0
cow_ksm 0
postgres@tantor:~$ cat /sys/kernel/mm/ksm/run
0
postgres@tantor:~$ cat /sys/kernel/mm/ksm/pages_scanned
0
```

Эту проверку делают на всякий случай. ksm полезен и может быть включён, если в операционной системе запускаются виртуальные машины из одного файла образа виртуальной машины.

5) Проверьте значения параметров, влияющих на выделение памяти большего размера, чем есть:

```
postgres@tantor:~$ sudo sysctl -a | grep vm.over
vm.overcommit_kbytes = 0
vm.overcommit_memory = 0
vm.overcommit_ratio = 50
postgres@tantor:~$ sudo sysctl -a | grep vm.swap
vm.swappiness = 60
```

Так как в виртуальной машине отключён раздел подкачки, то `vm.overcommit_memory` должен быть равен нулю. Так как подкачка отключена, то значение параметра `vm.swappiness` не играет роли.

Часть 3. Выгрузка длинных строк утилитой `pg_dump`

1) Создайте базу данных с названием ela и таблицу следующими командами:

```
postgres@tantor:~$ psql
postgres=# create database ela;
CREATE DATABASE
```

```
postgres=# \c ela
You are now connected to database "ela" as user "postgres".NOTICE:  table "t2"
does not exist, skipping
postgres=# drop table if exists t2;
create table t2 (c1 text, c2 text);
insert into t2 (c1) VALUES (repeat('a', 1024*1024*512));
update t2 set c2 = c1;
\q
DROP TABLE
CREATE TABLE
INSERT 0 1
UPDATE 1
```

2) Остановите экземпляр postgresql

```
postgres@tantor:~$ sudo systemctl stop postgresql
```

3) Установите в 0 число огромных страниц:

```
postgres@tantor:~$ sudo sysctl -w vm.nr_hugepages=0
vm.nr_hugepages = 0
```

4) Попробуйте выгрузить базу данных ela утилитой `pg_dump`:

```
postgres@tantor:~$ pg_dump -c -C -d ela -f ela.sql
pg_dump: error: Dumping the contents of table "t2" failed: PQgetResult() failed.
pg_dump: detail: Error message from server: ERROR:  out of memory
DETAIL:  Cannot enlarge string buffer containing 536870913 bytes by 536870912 more bytes.
pg_dump: detail: Command was: COPY public.t2 (c1, c2) TO stdout;
```

Возникла ошибка невозможности выгрузить содержимое таблицы из-за превышения размера памяти под `string buffer`. Максимальный размер буфера 1Гб.

5) Начиная с версии 16.2 в СУБД Тантор есть параметр `enable_large_allocations`, который увеличивает размер строкового буфера до 2Гб. Параметр можно установить и в сессиях утилиты pg_dump, используя параметр утилиты. Выгрузите базу данных ela, используя этот параметр:

```
postgres@tantor:~$ time pg_dump -c -d ela -f ela.sql --enable-large-allocations

real    0m12.556s
user    0m1.614s
sys     0m0.699s
```

выгрузка базы данных ela заняла 12.556 секунд.

6) Был создан файл размером 1Гб:

```
postgres@tantor:~$ ls -al ela.sql
-rw-r--r-- 1 postgres postgres 1073742755 ela.sql
```

Часть 4. Нехватка памяти

1) В окне терминала проверьте сколько памяти свободно:

```
postgres@tantor:~$ free
               total        used        free      shared  buff/cache   available
Mem:         4020792      993520     2888632      185788      546988     3027272
Swap:              0           0           0
```

Свободно 2.8Гб, доступно 3Гб
2) Запустите утилиту psql:

```
postgres@tantor:~$ psql -d ela
Type "help" for help.

ela=# \c ela
```

3) В другом терминале посмотрите список процессов экземпляров PostgreSQL:

```
root@tantor:~$ for PID in $(pgrep "postgres"); do awk '/Pss/ {PSS+=$2} END { getline cmd < "/proc/'$PID'/cmdline"; sub("\0", " ", cmd); getline oom < "/proc/'$PID'/oom_score"; printf "%.0f -%s-> %s (PID %s) \n", PSS, oom, cmd, '$PID'}' /proc/$PID/smaps; done|sort -n -r
3039132 -922-> postgres: postgres ela [local] COPY  (PID 12518)
28619 -0-> /opt/tantor/db/16/bin/postgres -D /var/lib/postgresql/tantor-se-16/data (PID 12499)
5502 -668-> postgres: walwriter   (PID 12503)
3031 -668-> postgres: autovacuum launcher   (PID 12504)
2753 -668-> postgres: checkpointer   (PID 12500)
2507 -668-> postgres: logical replication launcher   (PID 12507)
2499 -668-> postgres: pg_wait_sampling collector   (PID 12506)
2069 -668-> postgres: background writer   (PID 12501)
1697 -668-> postgres: autoprewarm leader   (PID 12505)
```

В первой строке номер серверного процесса, подсоединенного к базе данных ela.

3) В окне psql выполните скрипт, полученный утилитой pg_dump:

```
ela=# \i ela.sql
SET
CREATE TABLE
ALTER TABLE
psql:ela.sql:44: server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
psql:ela.sql:44: error: connection to server was lost
```

Соединение было потеряно.

4) Посмотрите последние сообщения ядра linux:

```
postgres@tantor:~$ sudo dmesg | tail -n 120
```

```
[16387.760734] Mem-Info:
[16387.760737] active_anon:342614 inactive_anon:609810 isolated_anon:0
                active_file:22 inactive_file:71 isolated_file:0
                unevictable:0 dirty:8 writeback:0
                slab_reclaimable:5664 slab_unreclaimable:13043
                mapped:15337 shmem:46435 pagetables:5001
                sec_pagetables:0 bounce:0
                kernel_misc_reclaimable:0
                free:20934 free_pcp:1 free_cma:0
[16387.760743] Node 0 active_anon:1370456kB inactive_anon:2439240kB active_file:88kB inactive_file:284kB unevictable:0kB isolated(anon):0kB isolated(file):0kB mapped:61348kB dirty:32kB writeback:0kB shmem:185740kB shmem_thp:0kB shmem_pmdmapped:0kB anon_thp:0kB writeback_tmp:0kB kernel_stack:5696kB pagetables:20004kB sec_pagetables:0kB all_unreclaimable? no
[16387.760749] Node 0 DMA free:14848kB boost:0kB min:260kB low:324kB high:388kB reserved_highatomic:0KB active_anon:0kB inactive_anon:0kB active_file:0kB inactive_file:0kB unevictable:0kB writepending:0kB present:15992kB managed:15360kB mlocked:0kB bounce:0kB free_pcp:0kB local_pcp:0kB free_cma:0kB
[16387.760756] lowmem_reserve[]: 0 3385 3825 3825 3825
[16387.760770] Node 0 DMA32 free:61272kB boost:0kB min:59580kB low:74472kB high:89364kB reserved_highatomic:0KB active_anon:1077032kB inactive_anon:2365224kB active_file:0kB inactive_file:0kB unevictable:0kB writepending:32kB present:3653568kB managed:3555048kB mlocked:0kB bounce:0kB free_pcp:4kB local_pcp:0kB free_cma:0kB
[16387.760778] lowmem_reserve[]: 0 0 439 439 439
[16387.760791] Node 0 Normal free:7616kB boost:0kB min:7736kB low:9668kB high:11600kB reserved_highatomic:0KB active_anon:105944kB inactive_anon:261496kB active_file:416kB inactive_file:0kB unevictable:0kB writepending:0kB present:524288kB managed:450384kB mlocked:0kB bounce:0kB free_pcp:0kB local_pcp:0kB free_cma:0kB
[16387.760798] lowmem_reserve[]: 0 0 0 0 0
[16387.760811] Node 0 DMA: 0*4kB 0*8kB 0*16kB 0*32kB 0*64kB 0*128kB 0*256kB 1*512kB (U) 0*1024kB 1*2048kB (M) 3*4096kB (M) = 14848kB
```

```
   [16387.760846] Node 0 DMA32: 849*4kB (UME) 615*8kB (UME) 229*16kB (UME) 278*32kB (UME) 177*64kB (UE) 67*128kB
(UME) 34*256kB (UME) 15*512kB (UME) 1*1024kB (U) 0*2048kB 1*4096kB (M) = 62284kB
   [16387.760893] Node 0 Normal: 230*4kB (UME) 153*8kB (UME) 107*16kB (UME) 71*32kB (UME) 25*64kB (UME) 5*128kB
(M) 0*256kB 0*512kB 0*1024kB 0*2048kB 0*4096kB = 8368kB
   [16387.760934] Node 0 hugepages_total=0 hugepages_free=0 hugepages_surp=0 hugepages_size=2048kB
   [16387.760937] 46530 total pagecache pages
   [16387.760939] 0 pages in swap cache
   [16387.760941] Free swap  = 0kB
   [16387.760943] Total swap = 0kB
   [16387.760946] 1048462 pages RAM
   [16387.760948] 0 pages HighMem/MovableOnly
   [16387.760950] 43264 pages reserved
   [16387.760952] 0 pages hwpoisoned
   [16387.760954] Tasks state (memory values in pages):
   [16387.760956] [  pid  ]   uid  tgid total_vm      rss pgtables_bytes swapents oom_score_adj name
   [16387.760961] [    270]     0   270    13053      288   118784        0          -250 systemd-journal
   [16387.760966] [    299]     0   299     8116      608    86016        0         -1000 systemd-udevd
   [16387.760971] [    426]     0   426     3359       66    49152        0         -1000 auditd
   [16387.760975] [    451]     0   451    59920      269   102400        0             0 accounts-daemon
   [16387.760978] [    452]     0   452      658       32    45056        0             0 acpid
   [16387.760982] [    455]   109   455     2246       96    61440        0             0 avahi-daemon
   [16387.760986] [    456]     0   456     1814       64    53248        0             0 cron
   [16387.760989] [    457]   100   457     2605      448    61440        0          -900 dbus-daemon
   [16387.760993] [    461]   109   461     2202       98    57344        0             0 avahi-daemon
   [16387.760996] [    467]   995   467    60688      895   110592        0             0 polkitd
   [16387.761000] [    473]     0   473     6953      256    81920        0             0 systemd-logind
   [16387.761003] [    562]     0   562    67078      832   163840        0             0 NetworkManager
   [16387.761007] [    578]     0   578     4923      256    77824        0             0 wpa_supplicant
   [16387.761010] [    585]     0   585    80121      589   122880        0             0 ModemManager
   [16387.761013] [    703]   114   703   423624     1552   208896        0             0 docker-registry
   [16387.761017] [    721]     0   721   340017     3498   299008        0          -999 containerd
   [16387.761020] [    786]     0   786     4954      448    81920        0         -1000 sshd
   [16387.761024] [    865]     0   865      620       96    45056        0             0 fly-getexe
   [16387.761027] [    891]     0   891     1630       64    49152        0             0 agetty
   [16387.761031] [   1744]     0  1744     9731     4013   118784        0             0 astra-event-dia
   [16387.761034] [   1768]     0  1768   384560     7030   425984        0          -500 dockerd
   [16387.761038] [   1774]     0  1774   345665     5100   466944        0             0 syslog-ng
   [16387.761041] [   5571]     0  5571     3670      192    73728        0             0 fly-dm
   [16387.761044] [   5578]   102  5578   100486    24100   581632        0             0 Xorg
   [16387.761047] [   5587]     0  5587   106865      136   106496        0             0 VBoxService
   [16387.761051] [   5642]   102  5642     5567      512    81920        0           100 systemd
   [16387.761054] [   5645]   102  5645    28019     1385   102400        0           100 (sd-pam)
   [16387.761058] [   5670]     0  5670     6821      400   102400        0             0 fly-dm
   [16387.761061] [   5682]  1000  5682     5610      576    81920        0           100 systemd
   [16387.761064] [   5683]  1000  5683    28019     1385   102400        0           100 (sd-pam)
   [16387.761068] [   5698]  1000  5698    13047     4141   147456        0             0 fly-wm
   [16387.761071] [   5758]  1000  5758     2317      256    57344        0           200 dbus-daemon
   [16387.761074] [   5791]  1000  5791    60058      832   192512        0           200 kglobalaccel5
   [16387.761078] [   5798]  1000  5798     4628       98    57344        0             0 VBoxClient
   [16387.761081] [   5799]  1000  5799    37695      130    69632        0             0 VBoxClient
   [16387.761085] [   5811]  1000  5811     4628      130    61440        0             0 VBoxClient
   [16387.761088] [   5812]  1000  5812    37687       98    73728        0             0 VBoxClient
   [16387.761091] [   5817]  1000  5817     4628      130    57344        0             0 VBoxClient
   [16387.761095] [   5819]  1000  5819    37816       66    69632        0             0 VBoxClient
   [16387.761098] [   5822]  1000  5822     4628       98    57344        0             0 VBoxClient
   [16387.761101] [   5823]  1000  5823    37748      194    73728        0             0 VBoxClient
   [16387.761104] [   5837]     0  5837    58952      256    98304        0             0 upowerd
   [16387.761108] [   5844]  1000  5844     2867      202    49152        0             0 ssh-agent
   [16387.761111] [   5870]  1000  5870   103544     1216   258048        0             0 polkit-kde-auth
   [16387.761115] [   5871]  1000  5871   125625     1600   290816        0             0 org_kde_powerde
   [16387.761118] [   5873]  1000  5873 67186318      736   196608        0             0 baloo_file
   [16387.761121] [   5876]  1000  5876   188404     3439   364544        0             0 fly-notificatio
   [16387.761124] [   5877]  1000  5877    39425      576   163840        0             0 astra-event-wat
   [16387.761128] [   5880]  1000  5880    36132      384   139264        0             0 fly-cups-watch
   [16387.761131] [   5881]  1000  5881    63043     1056   208896        0             0 kscreend
   [16387.761135] [   5882]  1000  5882 67439028     7809   712704        0             0 fly-start-menu
   [16387.761138] [   5883]  1000  5883    67294     1184   241664        0             0 fly-shutdown-sc
   [16387.761142] [   5886]  1000  5886    69346     1391   262144        0             0 fly-touchpadd
   [16387.761145] [   5892]  1000  5892    77765      288   102400        0             0 at-spi-bus-laun
   [16387.761148] [   5896]  1000  5896    58767      160    94208        0             0 agent
   [16387.761152] [   5900]  1000  5900    90195     1472   282624        0             0 fly-notify-osd-
   [16387.761155] [   5911]  1000  5911   393645     8394   720896        0             0 fly-sound-apple
   [16387.761158] [   5916]  1000  5916    87538     1600   266240        0             0 fly-reflex-serv
   [16387.761171] [   5921]  1000  5921     2208      160    61440        0             0 dbus-daemon
   [16387.761176] [   5922]  1000  5922    14788     3917   163840        0             0 applet.py
   [16387.761179] [   5929]  1000  5929     4633      256    73728        0             0 compton
   [16387.761182] [   5944]  1000  5944   123154     1376   278528        0           200 kactivitymanage
   [16387.761186] [   5974]  1000  5974   106825      418   106496        0             0 pulseaudio
   [16387.761190] [   5976]     0  5976    99686      565   139264        0             0 udisksd
   [16387.761193] [   5985]  1000  5985    60389      384   106496        0             0 gsettings-helpe
   [16387.761196] [   6002]  1000  6002    41071      224    86016        0             0 at-spi2-registr
   [16387.761200] [   6003]  1000  6003   161984     4706   495616        0             0 nm-applet
   [16387.761203] [   6017]  1000  6017    59383      832   188416        0           200 kscreen_backend
   [16387.761207] [   6067]  1000  6067    59604      224   102400        0           200 gvfsd
```

```
[16387.761211] [   6450]  1000  6450   154277     5493   385024        0             0 fly-term
[16387.761214] [   6459]  1000  6459     2225      416    57344        0             0 bash
[16387.761218] [   6835]  1000  6835     5043      320    81920        0             0 su
[16387.761221] [   6836]   113  6836     2322      448    53248        0             0 bash
[16387.761225] [   8280]     0  8280     8723     1216   102400        0             0 cupsd
[16387.761229] [  10858]  1000 10858   154156     5320   376832        0             0 fly-term
[16387.761232] [  10865]  1000 10865     2225      384    57344        0             0 bash
[16387.761236] [  12288]   113 12288    66375     8928   229376        0          -900 postgres
[16387.761239] [  12289]   113 12289    19967      637   122880        0             0 postgres
[16387.761243] [  12290]   113 12290    66442     1313   155648        0             0 postgres
[16387.761246] [  12291]   113 12291    66409     1185   151552        0             0 postgres
[16387.761249] [  12295]   113 12295    66375     1729   143360        0             0 postgres
[16387.761252] [  12296]   113 12296    66878      993   172032        0             0 postgres
[16387.761256] [  12297]   113 12297    66854      865   159744        0             0 postgres
[16387.761260] [  12499]   113 12499    57491     4352   184320        0         -1000 postgres
[16387.761263] [  12500]   113 12500    57528     1104   151552        0             0 postgres
[16387.761267] [  12501]   113 12501    57524     1008   147456        0             0 postgres
[16387.761271] [  12503]   113 12503    57524     1744   143360        0             0 postgres
[16387.761274] [  12504]   113 12504    57923     1008   163840        0             0 postgres
[16387.761277] [  12505]   113 12505    57524      742   147456        0             0 postgres
[16387.761280] [  12506]   113 12506    57635      880   143360        0             0 postgres
[16387.761283] [  12507]   113 12507    57891      912   151552        0             0 postgres
[16387.761287] [  12517]   113 12517     6495      448    94208        0             0 psql
[16387.761290] [  12518]   113 12518  1630962   789553  6533120        0             0 postgres
[16387.761293] [  12540]  1000 12540     5043      256    86016        0             0 su
[16387.761297] [  12541]   113 12541     2221      384    57344        0             0 bash
[16387.761300] oom-kill:constraint=CONSTRAINT_NONE,nodemask=(null),
cpuset=tantor-se-server-16.service,mems_allowed=0,global_oom,
task_memcg=/system.slice/tantor-se-server-16.service,task=postgres,pid=12518,
uid=113
[16387.761329] Out of memory: Killed process 12518 (postgres) total-
vm:6523848kB, anon-rss:3151300kB, file-rss:0kB, shmem-rss:6912kB, UID:113
pgtables:6380kB oom_score_adj:0
```

Процесс OOM kill остановил серверный процесс, который выполнял скрипт.

При срабатывании OOM-kill значение `oom_score_adj` никакой роли не сыграло, так как в операционной системе не было потребителей памяти, кроме процессов PostgreSQL.

Процесс OOM kill показывает размер **виртуальной** памяти `total-vm:6523848kB`. `6523848kB/4kB` (размер страницы)=`1630962` (в страницах) это размер памяти серверного процесса на момент его остановки.

Распределение памяти в диагностических сообщениях отображается в страницах размером 4Кб, кроме столбца **pgtables_bytes**:

```
[16387.760954] Tasks state (memory values in pages):
[16387.760956] [  pid  ]  uid  tgid total_vm      rss pgtables_bytes swapents oom_score_adj name
[16387.761290] [  12518]  113 12518  1630962   789553        6533120        0             0 postgres
```

===== ==== примечание ==========

Серверный процесс читал файл в свою локальную память размером 3Гб.

В виртуальной машине 4Гб и нет подкачки. По данным утилиты `free` доступной памяти (`available`) было `3027272kibi`. **При этом операционная система выделила `6523848kB` виртуальной памяти**. Если бы значение `vm.overcommit_memory=2` при `vm.overcommit_ratio=50`, то процесс смог бы выделить только 1,5Гб памяти и получил бы отказ на попытку выделения большего количества, несмотря на то, что доступной памяти было 3Гб. При этом бы процесс OOM-kill себя бы не проявлял. Пример:

```
postgres=# \i ela.sql
SET
CREATE TABLE
ALTER TABLE
psql:ela.sql:44: ERROR:  out of memory
DETAIL:  Failed on request of size 2147483646 in memory context "COPY".
CONTEXT:  COPY t2, line 1
```

Не пытайтесь установить на виртуальной машине **vm.overcommit_memory=2** без включенного swap, либо увеличения:

```
root@tantor:~# sysctl -w vm.overcommit_ratio=100
```

иначе вы не сможете дать ни одной команды:

```
root@tantor:~# free
               total        used        free      shared  buff/cache   available
Mem:         4020796      948788     1958532      188700     1533332     3072008
Swap:              0           0           0
root@tantor:~# sysctl -w vm.overcommit_memory=2
vm.overcommit_memory = 2
root@tantor:~# free
-bash: fork: Cannot allocate memory
root@tantor:~# reboot
-bash: fork: Cannot allocate memory
```

После включения swap в следующей части практики можно будет проверить, что при `vm.overcommit_memory=2` OOM kill не срабатывает, вместо срабатывания процессу, выделяющему память, выдаётся ошибка.

==========

5) Посмотрите список процессов PostgreSQL:

```
postgres@tantor:~$ for PID in $(pgrep "postgres"); do awk '/Pss/ {PSS+=$2} END { getline cmd < "/proc/'$PID'/cmdline"; sub("\0", " ", cmd); getline oom < "/proc/'$PID'/oom_score"; printf "%.0f -%s-> %s (PID %s) \n", PSS, oom, cmd, '$PID'}' /proc/$PID/smaps; done|sort -n -r
```

Список пуст, либо работают только процессы экземпляра Astralinux PostgreSQL. После того как OOM kill остановил серверный процесс, экземпляр был перезапущен в соответствии со значением параметра конфигурации `restart_after_crash`:

```
postgres=# select name, setting, context, max_val, min_val from pg_settings where name ~ 'restart';
         name          | setting | context | max_val | min_val
-----------------------+---------+---------+---------+---------
 restart_after_crash | on      | sighup  |         |
(1 row)
```

По умолчанию параметр включён и после сбоя серверного процесса процесс postgres аварийно остановит все дочерние процессы (эквивалент остановки в режиме immediate без контрольной точки) и запустит процессы заново (эквивалент "crash recovery" - запуска экземпляра после сбоя). Если отключить параметр, то все процессы экземпляра будут принудительно остановлены, в том числе процесс postgres. Состояние кластера после остановки будет таким же как после принудительной остановки командой `pg_ctl stop -m immediate`:

```
postgres@tantor:~$ pg_controldata | grep state
Database cluster state:               in production
```

6) Запустите экземпляр tantor:

```
postgres@tantor:~$ sudo systemctl start tantor-se-server-16
```

Если экземпляр не запускается, можно подождать пока он остановится или дать команду **`pg_ctl stop -m immediate`** и повторить команду **`sudo systemctl start tantor-se-server-16`**

7) Этот пункт можно не выполнять. Вызвать нехватку памяти и остановку psql может простой запрос:

```
postgres@tantor:~$ psql

postgres=# select repeat('a', 100000000) from generate_series(1, 100);
Killed
```

```
postgres@tantor:~$ sudo dmesg | tail -5
[ 1002.311864] [   6421]   113  6421   747298   718592  5861376     0     0 psql
[ 1002.311867] [   6422]   113  6422   139618    75066   770048     0     0 postgres
[ 1002.311870] oom-kill:constraint=CONSTRAINT_NONE,nodemask=(null),
cpuset=containerd.service,mems_allowed=0,global_oom,task_memcg=/user.slice/user-
1000.slice/session-3.scope, task=psql,pid=6421,uid=113
[ 1002.311894] Out of memory: Killed process 6421 (psql) total-vm:2989192kB, anon-
rss:2874368kB, file-rss:0kB, shmem-rss:0kB, UID:113 pgtables:5724kB oom_score_adj:0
```

В случае, если `vm.overcommit_memory=2` то результат будет:

```
postgres=# select repeat('a', 100000000) from generate_series(1, 100);
out of memory for query result
```

Установить `vm.overcommit_memory=2` можно только после установки `vm.overcommit_ratio=100` или включения swap, иначе в виртуальной машине сразу возникнет нехватка памяти и вы не сможете дать ни одной команды.

Обычно OOM kill останавливает процесс psql, но может остановить и процессы экземпляра, что приведёт к аварийной остановке экземпляра:

```
postgres=# select repeat('a', 1000000000) from generate_series(1, 1000);
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
The connection to the server was lost. Attempting reset: Failed.
The connection to the server was lost. Attempting reset: Failed.
!?>

[1211447.321264] [  pid  ]   uid  tgid  total_vm      rss pgtables_bytes swapents oom_score_adj name
[1211447.321658] [ 524388]   113 524388   756951   205344  4251648   313792             0 psql
[1211447.321661] [ 524389]   113 524389   808432   716332  6062080    17383             0 postgres
[1211447.321664] oom-kill:constraint=CONSTRAINT_NONE,nodemask=(null),cpuset=user.slice,mems_allowed=0,
global_oom,task_memcg=/system.slice/tantor-se-server-16.service,task=postgres,pid=524389,uid=113
[1211447.321735] Out of memory: Killed process 524389 (postgres) total-vm:3233728kB, anon-rss:2863536kB,
file-rss:1664kB, shmem-rss:128kB, UID:113 pgtables:5920kB oom_score_adj:0
```

При этом экземпяр может не запуститься и его надо будет запускать командой:
`sudo systemctl start tantor-se-server-16`

Часть 5. Включение подкачки (swap)

1) Добавьте файл для swap размером 2Гб и включите swap:

```
root@tantor:~# dd if=/dev/zero of=/swap_file bs=1M count=2048
2048+0 records in
2048+0 records out
2147483648 bytes (2.1 GB, 2.0 GiB) copied, 22.3827 s, 95.9 MB/s
root@tantor:~# chmod 600 /swap_file
root@tantor:~# mkswap /swap_file
Setting up swapspace version 1, size = 2 GiB (2147479552 bytes)
no label, UUID=4b56ea3b-ac66-46ef-801b-2aa473f27ef6
root@tantor:~# swapon /swap_file
root@tantor:~# free -m
               total        used        free      shared  buff/cache   available
Mem:            3926         881         672          19        2680        3044
Swap:           2047           0        2047
```

При перезагрузке виртуальной машины подкачка не будет включена автоматически.

2) Выполните скрипт дампа:

```
postgres=# \timing
Timing is on.
```

```
ela=# \i ela.sql
...
SET
CREATE TABLE
ALTER TABLE
COPY 1
Time: 18979.619 ms (00:18.980)
```

Команда COPY загружала строку 18 секунд.

3) В том же окне psql, в котором выполнялся скрипт загрузки посмотрите значение параметра `enable_large_allocations`:

```
ela=# show enable_large_allocations;
 enable_large_allocations
--------------------------
 on
(1 row)
```

Параметр был включён при выполнении файла ela.sql командой:
`SET enable_large_allocations TO on;` из этого файла.
По умолчанию параметр выключен.

4) В другом окне терминала посмотрите начало файла `ela.sql`:

```
postgres@tantor:~$ head -n 20 ela.sql
--
-- PostgreSQL database dump
SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET transaction_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;
SET enable_large_allocations TO on;
```

Значение параметра `enable_large_allocations` было выгружено утилитой `pg_dump`.

5) посмотрите характеристики параметра `enable_large_allocations`:

```
ela=# select * from pg_settings where name like '%large%' \gx
-[ RECORD 1 ]---+--------------------------------------------
name            | enable_large_allocations
setting         | on
unit            |
category        | Resource Usage / Memory
short_desc      | Sets whether to use large memory buffers - greater than 1 Gb, up to 2 Gb
extra_desc      |
context         | superuser
vartype         | bool
source          | session
min_val         |
max_val         |
enumvals        |
boot_val        | off
reset_val       | off
sourcefile      |
sourceline      |
pending_restart | f
```

Параметр может устанавливаться на разных уровнях, в том числе на уровне сессии.
Параметр увеличивает размер строкового буфера с 1Гб до 2Гб. Могут выгружаться строки размером до 2Гб. Строки больше 2Гб выгружаться не будут.

Часть 6. Страничный кэш

1) Откройте или переключитесь в терминал с правами root:

```
postgres@tantor:~$ su -
Password: root
```

2) Выполните команды:

```
root@tantor:~# cat /proc/meminfo | grep Cached
Cached:           1367440 kB
SwapCached:         49328 kB
root@tantor:~# free
               total        used        free      shared  buff/cache   available
Mem:         4020792      605576     2250740       36796     1452148     3415216
Swap:        2097148      508696     1588452
```

По умолчанию команда free показывает значения в kibibytes (число байт поделенных на 1024 1024). Для вывода в килобайтах нужно использовать параметр --kylo или --si (число байт поделенных на 1000). Разница заметная, во втором разряде слева (36820 37703):

```
root@tantor:~# free
               total        used        free      shared  buff/cache   available
Mem:         4020792      575192     3493796       36820      215152     3445600
Swap:        2097148      507160     1589988
root@tantor:~# free --si
               total        used        free      shared  buff/cache   available
Mem:         4117291      589750     3576889       37703      220323     3527540
Swap:        2147479      519331     1628147
root@tantor:~# free --bytes
               total        used        free      shared  buff/cache   available
Mem:      4117291008   588890112  3577745408    37703680   220340224  3528400896
Swap:     2147479552   519331840  1628147712
```

Cached входит в buff/cache , разница (с учётом разных единиц измерения) это значение buff.

3) Сделайте чистые блоки страничного кэша свободными:

```
root@tantor:~# echo 1 > /proc/sys/vm/drop_caches
root@tantor:~# cat /proc/meminfo | grep Cached
Cached:           156684 kB
SwapCached:         49332 kB
root@tantor:~# free
               total        used        free      shared  buff/cache   available
Mem:         4020792      536520     3530520       36800      184272     3484272
Swap:        2097148      508696     1588452
```

Значения Cached и buff/cache уменьшились на 1,2Гб. Значение free увеличилось.

4) Посмотрите число страниц разного размера:

```
root@tantor:~# cat /proc/buddyinfo
Node 0, zone      DMA      0      0      0      0      0      0      0      1      0      1      3
Node 0, zone    DMA32   1670   1466    868    707    734    503    294    166    111    111    648
Node 0, zone   Normal    444    283    262     98     31     10      3     39     32      2      1
```

Node 0 - номер физического процессора. Зоны:

zone DMA - виртуальная память со смещением от нуля до 16Мб

zone DMA32 - от 16Мб до 4Гб

zone Normal - от 4Гб и до 2^48 (2 в степени 48)

Каждая зона делится на части адресного пространства памяти размером (4096 байт *2^n):

4Кб 8Кб 16Кб 32Кб 64Кб 128Кб 256Кб 512Кб 1Мб 2Мб 4Мб

5) Посмотрите индекс фрагментации наборов страниц:

```
root@tantor:~# cat /sys/kernel/debug/extfrag/extfrag_index
Node 0, zone    DMA -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
Node 0, zone  DMA32 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
Node 0, zone Normal -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000  0.964  0.982  0.991  0.996
```

Дефрагментация запустится автоматически при уменьшении свободной памяти ниже vm.min_free_kbytes=67584 или значения vm.watermark_scale_factor. Значение по умолчанию vm.watermark_scale_factor=10, что означает 0.1% от размера **свободной** физической памяти. Эти значения слишком малы и дефрагментация запускается в крайнем случае. Рекомендации по установке значений даны в теоретической части главы.

6) Запустите дефрагментацию вручную:

```
root@tantor:~# echo 1 > /proc/sys/vm/compact_memory
```

7) Проверьте результат дефрагментации (как изменился индекс фрагментации):

```
root@tantor:~# cat /sys/kernel/debug/extfrag/extfrag_index
Node 0, zone    DMA -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
Node 0, zone  DMA32 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
Node 0, zone Normal -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000  0.991
```

8) Посмотрите сколько грязных 4-килобайтных страниц в страничном кэше linux:

```
root@tantor:~# cat /proc/vmstat | grep dirty
nr_dirty 126
nr_dirty_threshold 176446
nr_dirty_background_threshold 88115
```

9) Выполните команду sync, которая сбрасывает грязные страницы на диск:

```
root@tantor:~# sync
root@tantor:~# cat /proc/vmstat | grep dirty
nr_dirty 0
nr_dirty_threshold 174962
nr_dirty_background_threshold 87374
root@tantor:~# cat /proc/vmstat | grep dirty
nr_dirty 2
nr_dirty_threshold 175071
nr_dirty_background_threshold 87428
```

По команде sync сбрасываются все грязные страницы и nr_dirty в какой-то момент становится равен 0, но при работе процессов станицы постоянно грязнятся и значение почти сразу увеличивается.

10) Посмотрите значения параметров, относящихся к работе с грязными страницами:

```
root@tantor:~# sysctl -a | grep dirty
vm.dirty_background_bytes = 0
vm.dirty_background_ratio = 10
vm.dirty_bytes = 0
vm.dirty_expire_centisecs = 3000
vm.dirty_ratio = 20
vm.dirty_writeback_centisecs = 500
vm.dirtytime_expire_seconds = 43200
```

Длительность удержания страницы с момента изменения определяется параметром: `vm.dirty_expire_centisecs` - сколько буфер может быть грязным, прежде чем будет помечен для записи, по умолчанию 3000 (30 секунд), можно поменять на 500 (5 секунд).
Также на удержание влияет параметр: `vm.dirty_writeback_centisecs` - период ожидания между записями на диск, по умолчанию 500 (5 секунд), можно уменьшить до 250 (2,5 секунды). Рекомендуемые значения остальных параметров:

```
vm.dirty_ratio = 10
vm.dirty_background_ratio = 3
```

Эти параметры можно не менять, так как страницы кэша linux на запись посылают процессы экземпляра вызовом fdatasync по WAL, fsync по файлам табличных пространств в конце контрольной точки и делают это для отказоустойчивости (защиты от пропадания питания). Этими параметрами можно сгладить пиковую нагрузку, которая возникает в конце выполнения контрольной точки. Но даже для сглаживания нагрузки имеются параметры PostgreSQL `checkpoint_flush_after` и `bgwriter_flush_after`, которые также установлены в оптимальные для обычной нагрузки и обычного оборудования (HDD и SSD) значений.

11) Посмотрите параметры конфигурации PostgreSQL, влияющие на удержание грязных страниц страничного кэша:

```
postgres=# \dconfig *flush*
List of configuration parameters
       Parameter        | Value
------------------------+-------
 backend_flush_after    | 0
 bgwriter_flush_after   | 512kB
 checkpoint_flush_after | 256kB
 wal_writer_flush_after | 1MB
(4 rows)
```

Эти параметры ограничивают объём грязных страниц в страничном кэше linux и уменьшает вероятность затормаживания при выполнении fsync в конце контрольной точки или когда linux сбрасывает грязные страницы на диск в соответствии с параметрами из предыдущего пункта. Если параметры предыдущего пункта допускают накопление большого количества грязных страниц, могут возникать относительно длительные задержки, что делает работу экземпляра менее "гладкой".

12) Выполните команду:

```
postgres=# alter system set shared_preload_libraries = pg_stat_statements,
pg_wait_sampling, pg_stat_kcache, pg_qualstats, pg_store_plans, pg_prewarm;
```

13) перегрузите экземпляр, так как изменение параметра применится только после перезапуска экземпляра:

```
root@tantor:~# systemctl restart tantor-se-server-16
```

Эта команда загрузит при запуске экземпляра разделяемые библиотеки в память процессов. Загруженная библиотека `pg_wait_sampling` понадобится в следующей практике.

Практика к главе 3

Часть 1. Стандартный тест pgbench

1) В обсуждении https://www.postgresql.org/message-id/flat/53FD5D6C.40105%40catalyst.net.nz приводился пример теста pgbench с включенным HyperThread и выключенным на 4-ядерном процессоре. Посмотрим на каком числе клиентов достигается максимальный tps на вашей виртуальной машине. Создайте тестовые таблицы со scale factor 300 как в тесте по ссылке:

```
postgres@tantor:~$ pgbench -i -s 300
dropping old tables...
creating tables...
generating data (client-side)...
30000000 of 30000000 tuples (100%) done (elapsed 50.12 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 78.79 s (drop tables 0.02 s, create tables 0.01 s, client-side generate
50.79 s, vacuum 2.46 s, primary keys 25.52 s).
```

Таблицы создавались 78 секунд.

2) Выполните 15-секундные тесты с разным числом соединений:

```
postgres@tantor:~$ pgbench -c 64 -T 15 2> /dev/null | grep tps
tps = 1773.570100 (without initial connection time)
postgres@tantor:~$ pgbench -c 100 -T 15 2> /dev/null | grep tps
tps = 1646.101354 (without initial connection time)
postgres@tantor:~$ pgbench -c 32 -T 15 2> /dev/null | grep tps
tps = 1685.528291 (without initial connection time)
postgres@tantor:~$ pgbench -c 16 -T 15 2> /dev/null | grep tps
tps = 1550.934797 (without initial connection time)
```

Максимальный tps достигается при 64 соединениях.

3) Повторите тест:

```
postgres@tantor:~$ pgbench -c 64 -T 15 2> /dev/null | grep tps
tps = 1676.523228 (without initial connection time)
postgres@tantor:~$ pgbench -c 100 -T 15 2> /dev/null | grep tps
tps = 1572.848543 (without initial connection time)
postgres@tantor:~$ pgbench -c 32 -T 15 2> /dev/null | grep tps
tps = 1694.453483 (without initial connection time)
postgres@tantor:~$ pgbench -c 16 -T 15 2> /dev/null | grep tps
tps = 1477.982431 (without initial connection time)
```

Значения tps немного понизились во всех тестах. Тест по умолчанию вносит изменения в строки таблиц и индексные записи. Повторные запуски тестов дают стабильно худшие результаты. Тест с изменением таблиц нельзя использовать без пересоздания таблиц.
В обсуждении, на которое была приведена ссылка, в зависимости от процессора максимальный tps достигался на 256, 96, 48 соединениях на процессорах с разным числом ядер. Точность до ближайшей степени двойки: приведено значение 256, а реальный максимум может быть в диапазоне от 140 до 350.

4) Можно использовать select-only тест, который не меняет данные. Этот тест зависит от заполнения кэша буферов и первые выполнения теста имеют меньший tps, чем последующие. Максимальный tps стал достигаться на 16 соединениях.

```
postgres@tantor:~$ pgbench -b select-only -c 64 -T 15  2> /dev/null | grep tps
tps = 8687.455945 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 32 -T 15  2> /dev/null | grep tps
tps = 9112.963307 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 16 -T 15  2> /dev/null | grep tps
tps = 9611.093353 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 8 -T 15  2> /dev/null | grep tps
tps = 9548.481330 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 4 -T 15  2> /dev/null | grep tps
tps = 7841.653129 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 100 -T 15  2> /dev/null | grep tps
tps = 8780.049265 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 64 -T 15  2> /dev/null | grep tps
tps = 9200.191422 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 32 -T 15  2> /dev/null | grep tps
tps = 9499.272270 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 16 -T 15  2> /dev/null | grep tps
tps = 9820.530795 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 8 -T 15  2> /dev/null | grep tps
tps = 9637.473378 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 4 -T 15  2> /dev/null | grep tps
tps = 7825.917563 (without initial connection time)
```

Часть 2. Привязка процессов к ядру процессора

Если на хосте один процессор, то привязка процессов экземпляра PostgreSQL вряд ли даст преимущества. При этом операционная система выполняет привязку своих потоков к ядрам. Те потоки, которые работают с теми же структурами памяти, назначаются на то же ядро. При привязке процессов экземпляра стоит учесть на какие ядра назначены потоки операционной системы, чтобы эти ядра не были узким местом.

1) Посмотрите число ядер процессоров:

```
postgres@tantor:~$ nproc
4
```

2) Выполните команду, чтобы посмотреть привязку потоков ядра linux. У демона потоков ядра kthreadd идентификатор процесса pid=2:

```
for tid in $(pgrep -w 2 | tr '\n' ' '); do
 ps --no-headers -q "${tid}" -o pid -o user -o psr -o cmd;
 taskset -cp "${tid}";
done
     26 root       2 [cpuhp/2]
pid 26's current affinity list: 2
     27 root       2 [idle_inject/2]
pid 27's current affinity list: 2
     28 root       2 [migration/2]
pid 28's current affinity list: 2
     29 root       2 [ksoftirqd/2]
pid 29's current affinity list: 2
     73 root       3 [scsi_eh_2]
pid 73's current affinity list: 0-3
    137 root       2 [kworker/2:1H-kblockd]
...
```

Часть потоков ядра linux привязана к каждому из ядер cpu. kernel threads используются linux для выполнения фоновых операций.

3) Выполните команду, чтобы посмотреть привязку процессов с именем postgres:

```
for tid in $(pgrep postgres | tr '\n' ' '); do
 ps --no-headers -q "${tid}" -o pid -o user -o psr -o cmd;
 taskset -cp "${tid}";
done
```

```
 126673 postgres   1 /opt/tantor/db/16/bin/postgres -D /var/lib/postgresql/tantor-se-16/data
pid 126673's current affinity list: 0-3
 126674 postgres   1 postgres: checkpointer
pid 126674's current affinity list: 0-3
 126675 postgres   3 postgres: background writer
pid 126675's current affinity list: 0-3
 126677 postgres   0 postgres: walwriter
pid 126677's current affinity list: 0-3
 126678 postgres   1 postgres: autovacuum launcher
pid 126678's current affinity list: 0-3
 126679 postgres   3 postgres: autoprewarm leader
pid 126679's current affinity list: 0-3
 126680 postgres   0 postgres: pg_wait_sampling collector
pid 126680's current affinity list: 0-3
 126681 postgres   3 postgres: logical replication launcher
pid 126681's current affinity list: 0-3
...
```

По умолчанию у процессов postgres привязки к cpu нет. В примере процессы могут быть назначены на любое доступное ядро: 0,1,2,3.

4) Привяжите процесс checkpointer к ядру 1:

```
postgres@tantor:~$ taskset -p 12290
pid 12290's current affinity list: 0-3
pid 12290's new affinity list: 1
```

5) Уберите привязку процесса checkpointer к ядру 1:

```
postgres@tantor:~$ taskset -p -c 0-999 12290
pid 12290's current affinity list: 1
pid 12290's new affinity list: 0-3
```

Опции убрать привязку нет, можно только указать новую привязку, в которой перечислить все ядра. Самое простое - указать диапазон начиная с 0 до максимального числа ядер или выше.

Часть 3. Переключения контекстов выполнения ("VCX/ICX")

1) Посмотрите частоту переключений контекстов выполнения процессов postgres:

```
postgres@tantor:~$ pidstat -w -l -C postgres
Linux 6.6.28-1-generic (tantor)      _x86_64_        (4 CPU)

02:25:37 PM   UID       PID   cswch/s nvcswch/s  Command
02:25:37 PM   113      1405      0.10      0.02  /opt/tantor/db/16/bin/postgres -D /var/lib/postgresql/tantor-
se-16/data
02:25:37 PM   113      1485      0.07      0.01  /usr/lib/postgresql/15/bin/postgres -D
/var/lib/postgresql/15/main -c config_file=/etc/postgresql/15/main/postgresql.conf
02:25:37 PM   113      1586      0.01      0.00  postgres: checkpointer
02:25:37 PM   113      1587      0.39      0.00  postgres: background writer
02:25:37 PM   113      1620      0.02      0.00  postgres: 15/main: logger
02:25:37 PM   113      1693      0.26      0.00  postgres: walwriter
02:25:37 PM   113      1694      0.10      0.01  postgres: autovacuum launcher
02:25:37 PM   113      1695      0.02      0.04  postgres: autoprewarm leader
02:25:37 PM   113      1696     94.35      0.11  postgres: pg_wait_sampling collector
02:25:37 PM   113      1698      0.01      0.00  postgres: logical replication launcher
02:25:37 PM   113      1751      0.01      0.00  postgres: 15/main: checkpointer
02:25:37 PM   113      1752      0.39      0.00  postgres: 15/main: background writer
02:25:37 PM   113      1817      0.26      0.00  postgres: 15/main: walwriter
02:25:37 PM   113      1818      0.05      0.01  postgres: 15/main: autovacuum launcher
02:25:37 PM   113      1819      0.01      0.00  postgres: 15/main: logical replication launcher
02:25:37 PM     0      7653      0.00      0.00  pidstat -w -l -C postgres
```

Если экземпляр работал больше часа или операционная система перегружалась, то pidstat покажет, что у процесса **pg_wait_sampling collector** контексты переключаются с частотой ~100 (в примере 94.46) в секунду. Этот процесс занимается сбором статистики, опрашивая процессы экземпляра. Большинство циклов опроса завершается быстро, процесс освобождает ядро и статистика cswch увеличивается. Jiffy - единица времени, которая представляет

количество прерываний таймера от загрузки linux. При каждом прерывании таймера количество Jiffy увеличивается на единицу. Связь между Jiffy и секундами определяется константой HZ в ядре linux. Это значение определяет количество Jiffies в секунду. В linux до версии 2.4 частота тика между Jifies была 10 мс. Начиная с linux 2.4 тик стал 1мс. Начиная с версии ядра 3.10 частота тика варьируется и может быть замедлена, если при компиляции ядра установить CONFIG_NO_HZ_FULL=y. В этом случае прерывания таймера отключаются не только для простаивающих процессоров, но при исполнении кода, если на ядре нет задач в очереди на исполнение.

2) Посмотрите текущее число переключений контекста выполнения для процесса `pg_wait_sampling collector`. С интервалом в ~секунду выполните команду:

```
postgres@tantor:~$ grep ctxt /proc/1696/status
voluntary_ctxt_switches:        304222
nonvoluntary_ctxt_switches:     334
postgres@tantor:~$ grep ctxt /proc/1696/status
voluntary_ctxt_switches:        304343
nonvoluntary_ctxt_switches:     335
```

Число произвольных переключений контекста по этой кумулятивной статистике около 100 в секунду. Непроизвольные переключения тоже присутствуют. Данные совпадают с результатом pidstat. Процесс `pg_wait_sampling collector` опрашивает состояние всех процессов экземпляра с частотой, заданной параметром `pg_wait_sampling.history_period` или `pg_wait_sampling.profile_period`. Значения этих параметров по умолчанию 10 миллисекунд, что соответствует частоте 100 раз в секунду.

3) Если рестартовать экземпляр, то после перезапуска значения `cswch/s` будут постепенно увеличиваться. Выполните в терминале root:

```
root@tantor:~# systemctl restart tantor-se-server-16
root@tantor:~# pidstat -w -l -C postgres | grep sampling
02:36:31 PM   113      7728      4.25      0.00  postgres: pg_wait_sampling collector
```

Период усреднения - **с момента запуска операционной системы**, что делает использование утилиты pidstat бессмысленным.
Для получения осмысленного результата стоит пользоваться кумулятивными статистиками. До 10 в секунду при реальной частоте 100 в секунду значение дойдёт примерно через 7 минут. При этом кумулятивная статистика частоту показывает корректно 100 раз в секунду. Выполните два раза команду с интервалом примерно в секунду:

```
root@tantor:~# grep ctxt /proc/7728/status
voluntary_ctxt_switches:        13433
nonvoluntary_ctxt_switches:     10
root@tantor:~# grep ctxt /proc/7728/status
voluntary_ctxt_switches:        13551
nonvoluntary_ctxt_switches:     11
```

4) Для мониторинга переключений контекстов по всей операционной системе можно использовать утилиту `perf`:

```
root@tantor:~# perf stat -a
^C
 Performance counter stats for 'system wide':

         68,010.80 msec cpu-clock                 #    2.000 CPUs utilized
             9,885      context-switches          #  145.345 /sec
               250      cpu-migrations            #    3.676 /sec
             2,774      page-faults               #   40.788 /sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
```

```
    <not supported>      branch-misses

      34.005359735 seconds time elapsed
```

Утилита с такими параметрами работает до ее остановки комбинацией клавиш <ctrl+c>.
Утилита выдаёт правильные значения: 145 переключений контекстов в секунду. Недостаток: не выдаёт число involuntary context switches.
Для сбора такой же статистики по отдельному процессу используется команда:
`perf stat -p номер_процесса`

Часть 4. Мониторинг нагрузки на процессор

1) Установите, если не установлены, утилиты atop и htop в терминале под пользователем root:

```
root@tantor:~# apt update
Hit:1 cdrom://OS Astra Linux 1.8.1.6  DVD 1.8_x86-64 InRelease
Hit:2 https://download.astralinux.ru/astra/stable/1.8_x86-64/repository-extended 1.8_x86-64 InRelease
Ign:3 https://download.astralinux.ru/astra/stable/1.8_x86-64/repository-devel 1.8_x86-64 InRelease
Hit:4 https://download.astralinux.ru/astra/stable/1.8_x86-64/repository-main 1.8_x86-64 InRelease
Err:5 https://download.astralinux.ru/astra/stable/1.8_x86-64/repository-devel 1.8_x86-64 Release
  404  Not Found [IP: 130.193.50.59 443]
Reading package lists... Done
E: The repository 'https://download.astralinux.ru/astra/stable/1.8_x86-64/repository-devel 1.8_x86-64 Release'
does not have a Release file.
N: Updating from such a repository can't be done securely, and is therefore disabled by default.
N: See apt-secure(8) manpage for repository creation and user configuration details.
root@tantor:~# apt install htop -y
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
htop is already the newest version (3.2.2-2+b1).
0 upgraded, 0 newly installed, 0 to remove and 603 not upgraded.
root@tantor:~# apt install atop -y
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
atop is already the newest version (2.8.1-1+b1).
0 upgraded, 0 newly installed, 0 to remove and 603 not upgraded.
```

2) Удалите директорию, если она есть и запустите в терминале пользователя postgres утилиту `pg_basebackup` с замедлением резервирования до 50 килобайт в секунду:

```
postgres@tantor:~$ rm -rf /var/lib/postgresql/backup/1
postgres@tantor:~$ time pg_basebackup -c fast -D $HOME/backup/1 -P -r 50k
```

Утилита начнет резервирование. Окно терминала с работающей утилитой не нужно закрывать.

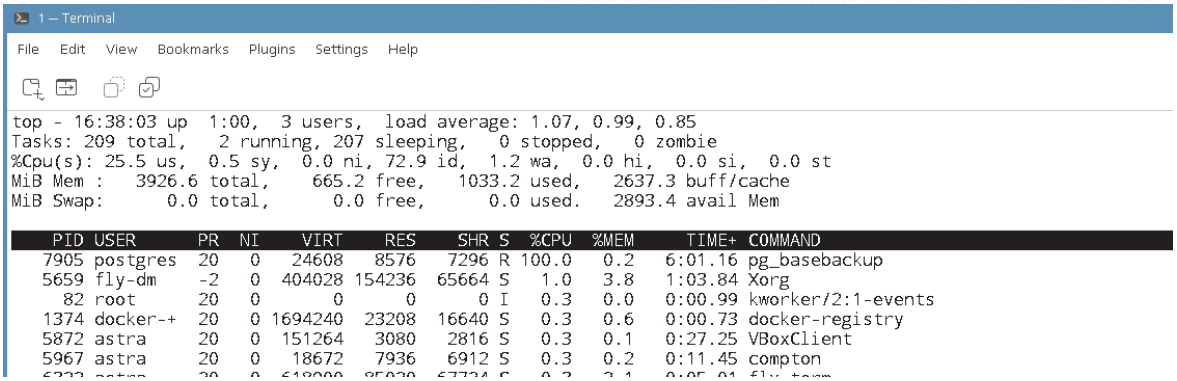
3) В другом терминале, под пользователем root, выполните команду top:

```
root@tantor:~# ps -e -o pcpu,vsz,rss,pss,cls,psr,rops,wops,s,cmd --sort -pcpu | head -2
%CPU    VSZ    RSS    PSS CLS PSR  ROPS  WOPS S CMD
99.9  24608   8576   1486  TS   1    43  8385 R pg_basebackup -c fast -D
/var/lib/postgresql/backup/1 -P -r 1M
```

Какая проблема видна в выводе команды?
Загрузка процессора 99.9%
Дальше последовательно посмотрите результат вывода утилит top, atop, htop, чтобы понять какой утилитой удобнее выявлять проблему нагрузки на процессор.

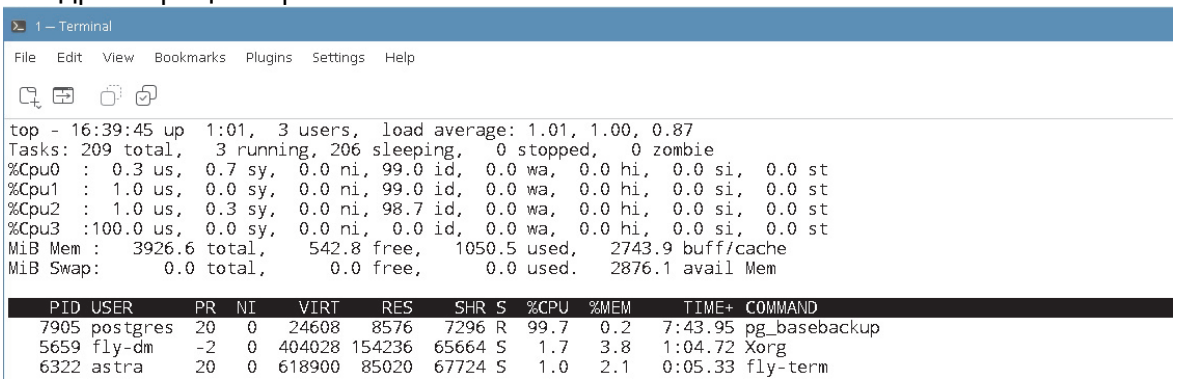
4) Запустите утилиту top:

```
root@tantor:~# top
```



Какую нагрузку на процессор показывает утилита?
В столбце %CPU утилита показывает 100.0.
В поле %Cpu(s): 25.5 us, 0.5 sy.

5) Нажмите на клавиатуре клавишу 1. Вывод утилиты изменится, она покажет нагрузку отдельно по ядрам процессора:



Нагрузка на 3 ядре: %Cpu3: 100.0 us, 0.0 sy и это реальная нагрузка.
Это означает что третье ядро полностью нагружено, причем отношение USER/SYS 100:0.

Посмотрите справа вверху на метрику `load average: 1.01, 1.00, 0.87`
Метрика также присутствует в файле `/proc/loadavg` и выводе команды `uptime`:

```
root@tantor:~# uptime
 09:39:07 up 18:01,  3 users,  load average: 0.19, 0.20, 0.18
```

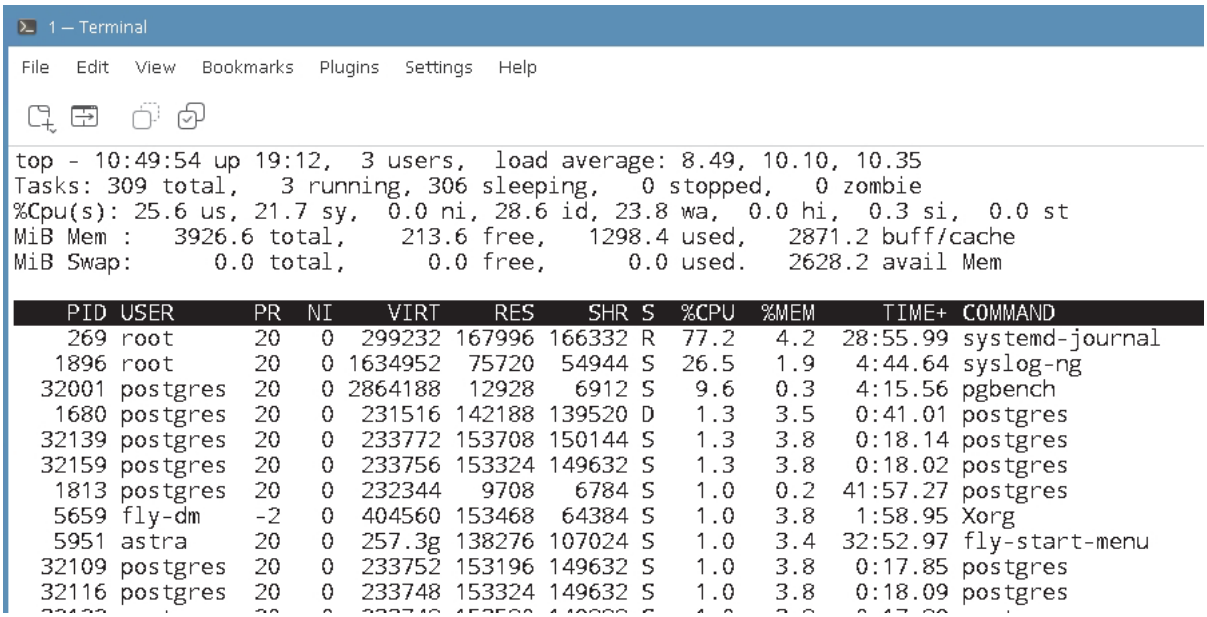
Это метрика показывает число активных (работающих или желающих работать) процессов, усредненная на интервалах 1, 5, 15 минут. В метрике учитываются процессы, потребляющие процессорное время и ожидающие получения результата от дисковой подсистемы. Процессы, ожидающие поступления данных с сетевых интерфейсов в метрике не учитываются.
Если число активных процессов меньше числа ядер, то если все процессы активны, они не будут испытывать недостатка в вычислительных ресурсах. Но число процессов обычно больше, чем число ядер. Если появляется большое число активных процессов, то доступ к ядрам делится между активными процессами. При работе СУБД процессор обычно не является узким местом (дефицитным ресурсом) и метрика не указывает на какие-либо проблемы. Метрика может быть полезна тем, что сравнив три числа можно оценить, является ли текущая нагрузка пиковой или постоянной. Если создать большое число активных процессов, например, запустив в отдельном терминале утилиту pgbench с числом сессий 90:

```
postgres@tantor:~$ pgbench -c 90 -j 90 -T 10000 -P 5
pgbench (16.2)
starting vacuum...end.
progress: 5.0 s, 892.4 tps, lat 105.630 ms stddev 87.005, 0 failed
progress: 10.0 s, 413.0 tps, lat 237.783 ms stddev 257.410, 0 failed
...
```

По умолчанию параметр конфигурации `max_connections=100`, поэтому число сессий выбрано 90.

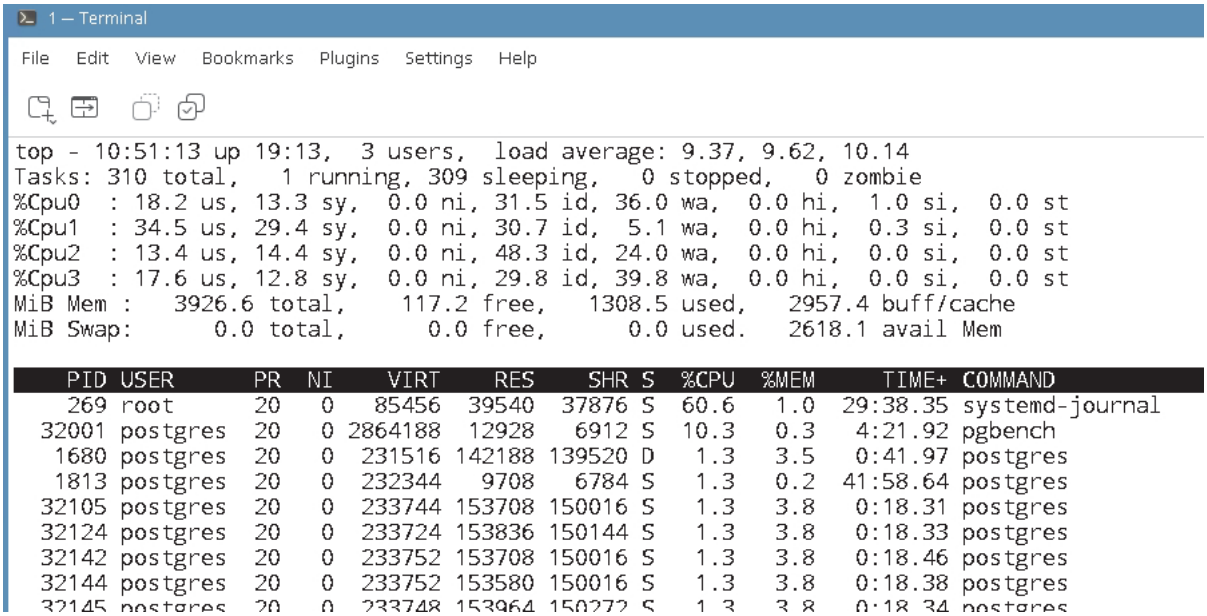
Дальше можно посмотреть вывод утилиты top суммарно по всем процессорам и по отдельности и можно будет увидеть, что если load превышает число ядер, то суммарная нагрузка на процессора становится полезной. Если суммарная нагрузка %Cpu(s) больше 90%, то процессор является узким местом. При долговременной нагрузке граница 90% (с целью определения дефицитности ресурса) может варьироваться от 85% до 95% в зависимости от доли ожидания дискового ввода-вывода.

Пример суммарной нагрузки - присутствует метрика **`%Cpu(s)`**:



Средняя загрузка по всем ядрам: 100-28.6=71.4% процессор не является узким местом. Однако, если снизить число серверных процессов с 90 до ~50 для четырех ядер в данном примере, то будут достигнуты максимальные tps. При большой загрузке процессоров переключения контекстов, а под сильно большой ожидания получения таймслотов на ядре процессора добавляют задержку. Накладные расходы на переключения контекстов (замена содержимого кэшей ядра) напрямую измерить нельзя.

Пример нагрузки по процессорам - метрики **`%Cpu0`** ... **`%Cpu3`** :



Нагрузка по процессорам полезна при load меньшим, чем число ядер и для определения нет ли процессов, которые загружают ядро на 100% и именно для этого процесса (потока который не

может распаралеллиться и обслуживаться несколькими ядрами) процессор является узким местом.

Закройте утилиту top, нажав на клавиатуре клавишу `q`.
Если работает pgbench, то его можно остановить комбинацией клавиш <ctrl+c>.

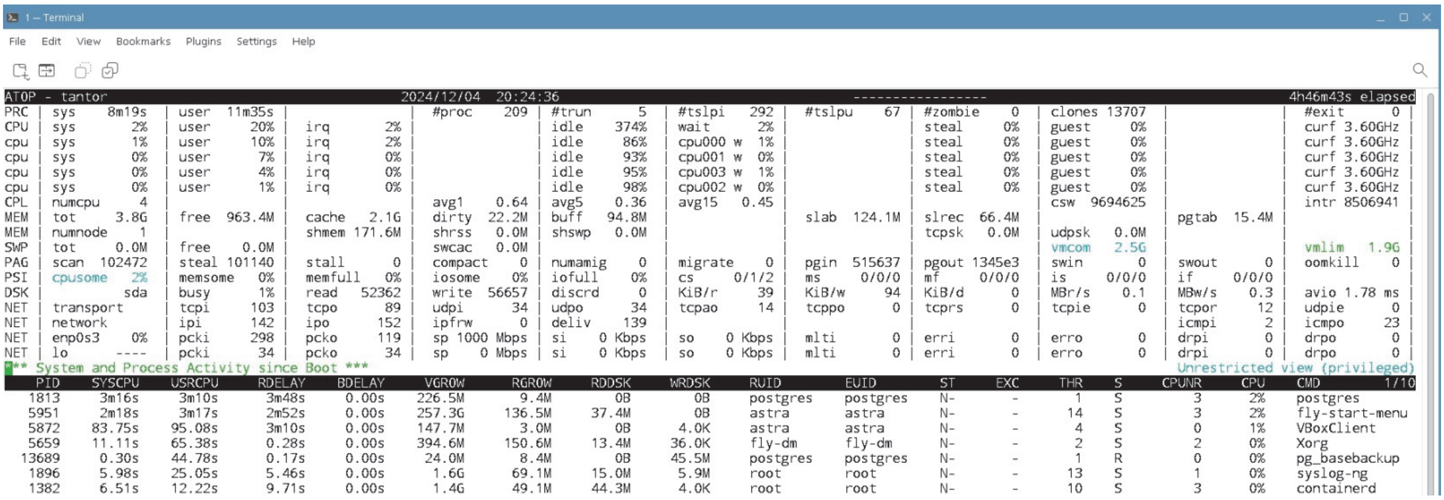
6) Запустите утилиту mpstat:

```
root@tantor:~# mpstat -n
Linux 6.6.28-1-generic (tantor)        _x86_64_        (4 CPU)

08:40:00 PM NODE    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
08:40:00 PM  all    5.71    0.05    0.39    0.42    0.00    0.46    0.00    0.00    0.00   92.97
```

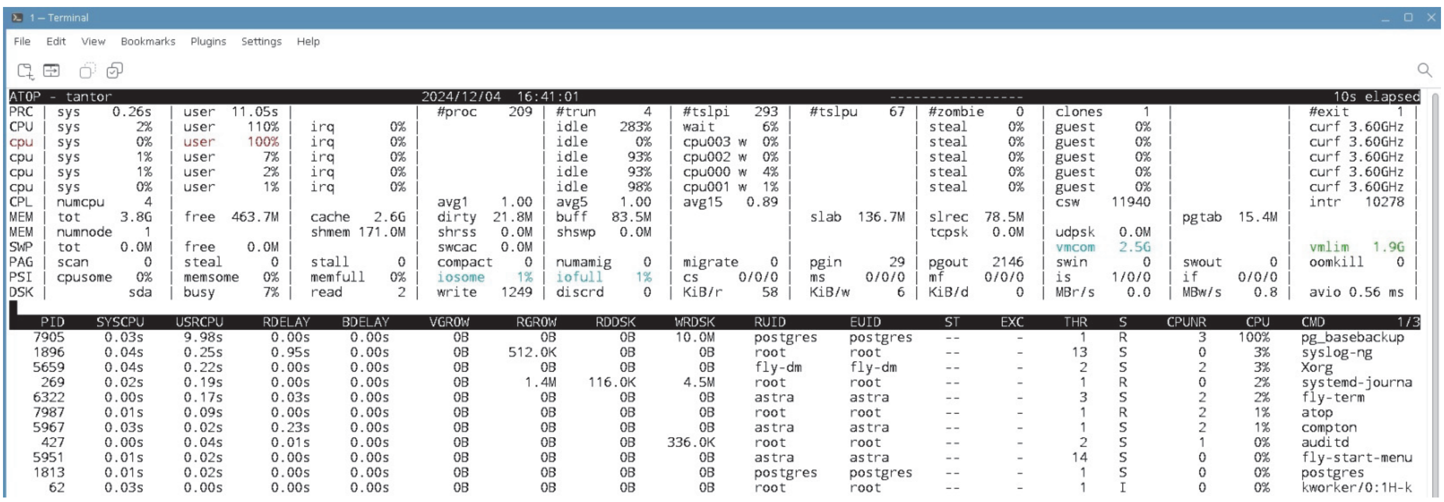
Посмотрите удобно ли наблюдать этой утилитой нагрузку одного ядра на 100% и отношение USER/SYS?

7) Запустите утилиту atop:

```
root@tantor:~# atop
```



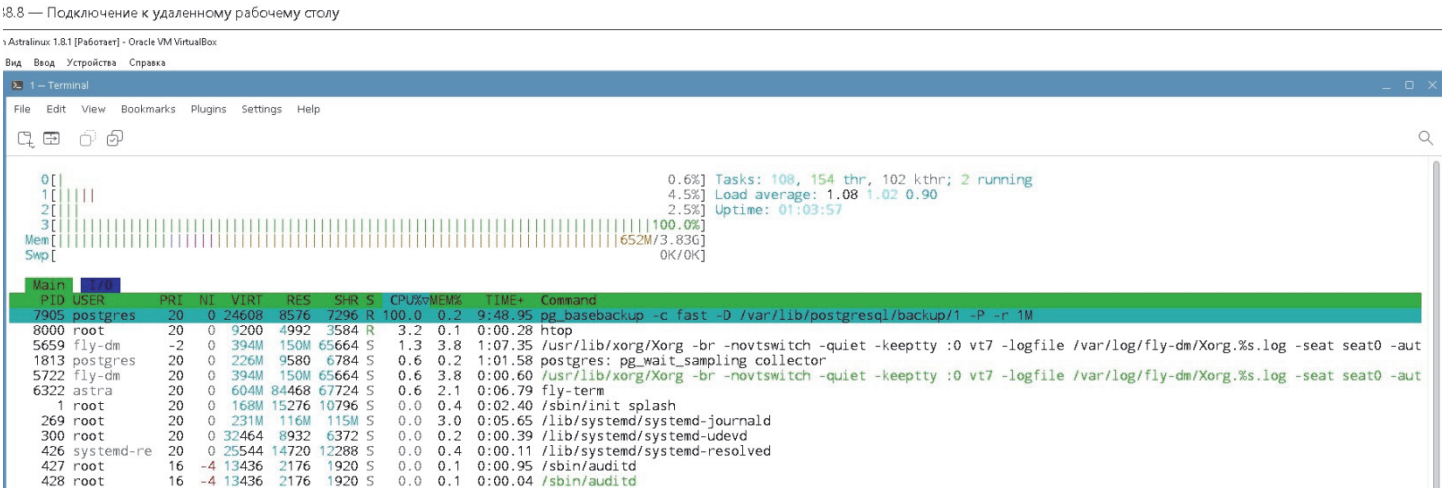
Зная реальную нагрузку найдите в окне утилиты `atop` нагрузку на процессор, которую даёт утилита `pg_basebackup`.
Через 10 секунд окно поменяется и выдаст другие цифры:



Отображаемые данные обновляются по умолчанию раз в 10 секунд.
Отметьте для себя: удобно ли просматривать нагрузку и видно ли отношение USER/SYS.
Закройте утилиту, нажав на клавиатуре клавишу q.

8) Запустите утилиту htop:

```
root@tantor:~# htop
```



Отметьте для себя: удобно ли просматривать нагрузку и видно ли отношение USER/SYS в утилите htop.
Нагрузка на ядра процессоров выдаются символами псевдографики.
Закройте утилиту, нажав на клавиатуре клавишу q.
Цель этой части практики выбрать утилиту, с помощью которой удобнее просматривать нагрузку на ядра процессоров.

Часть 5. Сбор статистик в файл и его просмотр утилитой `atop`

1) Утилита atop позволяет собирать и записывать статистики в бинарный файл. Файл можно "проигрывать" на другом хосте и это удобно для анализа. Минимальная частота сбора статистик 1 раз в секунду, по умолчанию раз в 10 секунд. Объем собираемых данных большой, но можно параметрами настроить какие статистики собирать для уменьшения размера файла.

2) Запустите сбор статистик с интервалом в 1 секунду в файл /atop.record :

```
root@tantor:~# atop -w /atop.record 1
```

Терминал не вернёт промпт, это нормально. Если файл существовал, то утилита не сотрёт собранные данные, а начнет дописывать данные в конец файла.

3) В отдельном терминале пользователя root запустите утилиту atop в режиме просмотра бинарного файла:

```
astra@tantor:~$ atop -r /atop.record
```

Используйте клавиши на клавиатуре 't' и 'T' (<Shift+t>) для перехода к следующему и возврату к предыдущему интервалу. В первой строке, выдаваемой утилитой показывается время на которое были собраны статистики. Для выхода из утилиты нажмите клавишу 'q'.

4) В терминале, где запущена утилита нажмите <ctrl+c>:

```
root@tantor:~# atop -w /atop.record 1
^C
root@tantor:~#
```

Утилита остановится. Файл можно повторно просматривать утилитой atop. Можно скопировать файл на другой хост и просматривать файл на другом хосте.

5) Удалите файл `/atop.record`:

```
root@tantor:~# rm -f /atop.record
```

6) Если утилита `pg_basebackup` не закончила резервирование, прервите работу утилиты, нажав на клавиатуре комбинацию клавиш <ctrl+c>:

```
postgres@tantor:~$ time pg_basebackup -c fast -D $HOME/backup/1 -P -r 50k
^C3781/902114 kB (68%), 0/1 tablespace

real    19m57.486s
user    19m53.055s
sys      0m3.413s
```

Обратите внимание, что утилита time выдала корректные значения USER/SYS.

7) Удалите директорию с бэкапом:

```
postgres@tantor:~$ rm -rf /var/lib/postgresql/backup/1
```

Часть 6. Источник времени linux

1) Посмотрите список источников, которые linux счел возможными к использованию:

```
postgres@tantor:~$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc acpi_pm
```

2) Посмотрите, какой источник времени используется:

```
postgres@tantor:~$ cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

3) Создайте текстовый файл с названием `clock_timing.c` и содержимым:

```
#include <time.h>
int main()
{
 int rc;
 long i;
 struct timespec ts;
 for(i=0; i<10000000; i++) rc = clock_gettime(CLOCK_MONOTONIC, &ts);
 return 0;
}
```

4) Скомпилируйте файл:

```
 postgres@tantor:~$ gcc clock_timing.c -o clock_timing -lrt
```

5) Программа 10 миллионов раз считывает показатель времени. Измерьте время выполнения программы:

```
postgres@tantor:~$ time ./clock_timing
real 0m13,967s
user 0m13,938s
```

```
sys 0m0,008s
```

Утилита time удобна для получения времени выполнения программ. Она возвращает реальные USER/SYS и общее время выполнения программы.

6) Выполните команду `pg_test_timing`:

```
postgres@tantor:~$ pg_test_timing
Testing timing overhead for 3 seconds.
Per loop time including overhead: 3931.22 ns
Histogram of timing durations:
  < us   % of total      count
     1      0.02475        497
     2     60.52305    1215427
     4     39.20322     787281
     8      0.02480        498
    16      0.08869       1781
    32      0.06384       1282
    64      0.05348       1074
   128      0.01215        244
   256      0.00388         78
   512      0.00065         13
  1024      0.00060         12
  2048      0.00030          6
  4096      0.00035          7
  8192      0.00020          4
 16384      0.00005          1
```

Программа `pg_test_timing` стандартно поставляется с PostgreSQL, используется для измерения скорости источника времени, выдаёт распределение выдаваемых источником времени значений.
Максимум распределения на ~2.5 миллисекундах.
Символами "us" обозначаются миллисекунды (µs). Колебания выдачи времени в приведённом выводе утилиты не меньше 1 миллисекунды. Это значит, что числа менее 1 миллисекунды являются случайными.
Программа `pg_test_timing` даёт более детальную информацию, чем `clock_timing.c`.
Программа `clock_timing.c` приведена для иллюстрации простоты создания собственных тестов на языке C. На языке C написан код PostgreSQL.

7) Создайте таблицу для теста, выполнив команды в psql:

```
postgres=# drop table if exists t;
create table t(pk bigserial, c1 text default
'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa');
insert into t select *, 'a' from generate_series(1, 100000);
DROP TABLE
CREATE TABLE
INSERT 0 100000
```

8) Выполните команду:

```
postgres=# explain analyze select count(pk) from t;
                                  QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=1352.50..1352.51 rows=1 width=8) (actual time=307.270..307.276 rows=1 loops=1)
   ->  Seq Scan on t  (cost=0.00..1190.20 rows=64920 width=8) (actual time=0.017..152.937 rows=100000 loops=1)
 Planning Time: 0.038 ms
 Execution Time: 307.383 ms
(4 rows)
```

Время выполнения команды 307 миллисекунд.

9) Замените источник времени tsc на acpi_pm. В консоли root выполните команду:

```
root@tantor:~# echo acpi_pm>/sys/devices/system/clocksource/clocksource0/current_clocksource
```

Команду можно выполнить, не переключаясь в консоль root:

```
postgres@tantor:~$
sudo sh -c 'echo acpi_pm>/sys/devices/system/clocksource/clocksource0/current_clocksource'
```

10) Выполните команду:

```
postgres=# explain analyze select count(pk) from t;
                                        QUERY PLAN
--------------------------------------------------------------------------------
 Aggregate  (cost=1352.50..1352.51 rows=1 width=8) (actual time=805.557..805.573 rows=1 loops=1)
   ->  Seq Scan on t  (cost=0.00..1190.20 rows=64920 width=8) (actual time=0.015..399.513 rows=100000 loops=1)
 Planning Time: 0.148 ms
 Execution Time: 805.792 ms
(4 rows)
```

Время выполнения команды увеличилось с 0.3 до 0.8 секунд.
На время выполнения простой команды `explain analyze` заметно повлияло изменение источника времени.

11) Измерьте время выполнения программы:

```
postgres@tantor:~$ time ./clock_timing

real	0m38.669s
user	0m16.329s
sys	0m22.331s
```

Время выполнения заметно увеличилось - почти в 3 раза. Пропорция USER/SYS изменилась с 1:1 на 8:11. Можно убедиться, что пропорция USER/SYS может использоваться для выявления неэффективного кода. Перекос пропорции в сторону SYS (отклонение от пропорции 60:40) указывает на то, что код ядра, выдающий время, неэффективен.

12) Выполните команду `pg_test_timing`:

```
postgres@tantor:~$ pg_test_timing
Testing timing overhead for 3 seconds.
Per loop time including overhead: 3889.59 ns
Histogram of timing durations:
  < us   % of total      count
     1      0.00389         30
     2      0.00052          4
     4     25.33700     195422
     8     74.23528     572570
    16      0.21470       1656
    32      0.08376        646
    64      0.10411        803
   128      0.01776        137
   256      0.00117          9
   512      0.00013          1
  1024      0.00052          4
  2048      0.00052          4
  4096      0.00039          3
  8192      0.00000          0
 16384      0.00026          2
```

Максимум распределения скорости выдачи временной метки с источником времени **acpi_pm** стал ~7 миллисекунд, а с источником времени tsc был ~ 2.5 миллисекунды.

13) Верните источник времени tsc:

```
root@tantor:~# echo tsc>/sys/devices/system/clocksource/clocksource0/current_clocksource
```

или

```
postgres@tantor:~$
sudo sh -c 'echo tsc>/sys/devices/system/clocksource/clocksource0/current_clocksource'
```

По умолчанию, источник времени каждый раз выбирается linux при загрузке. Есть вероятность, что будет произвольно выбран медленный источник времени. Такое может произойти и после обновления ядра linux. Вероятность увеличивают: неотключение энергосбережения в BIOS, включение Hyper Threading в BIOS, любая активность оборудования в процессе тестирования источника времени, например, активность Intel ME. **Вероятность произвольной замены источника времени высока на виртуальных машинах, так как при загрузке виртуальной машины нагрузка на аппаратуру неравномерна.** Рекомендуется проверять, какой источник времени используется или фиксировать желаемый источник времени в параметрах загрузке ядра linux. Пример изменения `/boot/grub/grub.cfg` приведён в теоретической части курса. Наиболее быстрый источник времени tsc.

Часть 7. Сетевые соединения

1) Откройте терминал пользователя root или переключитесь в консоль root:

```
postgres@tantor:~$ su -
Password: root
```

2) Посмотрите включён ли алгоритм медленного старта после простоя:

```
root@tantor:~# sysctl -a | grep net.ipv4.tcp_slow_start_after_idle
net.ipv4.tcp_slow_start_after_idle = 1
```

Алгоритм медленного старта после простоя включён. Если сессия простаивала и сетевого трафика не было, то в сетях с заметной сетевой задержкой (network latency) скорость передачи данных будет нарастать постепенно, а не сразу использовать полосу пропускания сети. Первые три пакета TCP передаются без задержки. Этого обычно достаточно для передачи команд и небольших результатов. Большая часть OLTP систем не испытывают замедления, так как запрос и результат вписывается в три сетевых пакета.

3) Посмотрите какой алгоритм управления нагрузкой на сетевой канал передачи данных используется:

```
root@tantor:~# sysctl -a | grep net.ipv4.tcp_congestion_control
net.ipv4.tcp_congestion_control = cubic
```

Используется алгоритм cubic. Это алгоритм, который устанавливает объем передаваемых данных измеряя потери пакетов TCP, что нежелательно и приводит к повторной передаче пакетов. Вероятность передачи больших объемов данных, использования сети с большой сетевой задержкой мала, поэтому на выбор алгоритма редко обращают внимание и сеть не является узким местом для СУБД и их клиентов. Однако, если по сетевому каналу передаются большие объемы данных, а это поточная репликация и резервирование, то их эффективность может быть меньше, чем могла бы быть.

4) Посмотрите параметры, определяющие поддержание сокета открытым:

```
root@tantor:~# sysctl -a | grep keepalive
net.ipv4.tcp_keepalive_intvl = 75
net.ipv4.tcp_keepalive_probes = 9
net.ipv4.tcp_keepalive_time = 7200
```

Это значения по умолчанию и их можно уменьшить.
Кроме этих параметров меняют значения параметра `net.ipv4.tcp_retries2` и упомянутого ранее `net.ipv4.tcp_slow_start_after_idle`.

Часть 8. Замена политики планирования и проверка работы планировщика

1) Создайте файл для тестирования нагрузки на процессор:

```
astra@tantor:~$ mcedit i.sh

#!/bin/sh
while true; do echo 1 > /dev/null ; done;
```

2) Проверьте, что содержимое файла успешно сохранилось:

```
astra@tantor:~$ cat i.sh
#!/bin/sh
while true; do echo 1 > /dev/null ; done;
```

3) Запустите процессы равное числу ядер процессоров виртуальной машины. Приведён пример для четырёх ядер:

```
astra@tantor:~$ nproc
4
astra@tantor:~$ taskset./i.sh &
[2] 212239
astra@tantor:~$ taskset./i.sh &
[3] 212242
astra@tantor:~$ taskset./i.sh &
[4] 212243
astra@tantor:~$ taskset./i.sh &
[5] 212244
```

4) Два процесса нагружают оба ядра на 100%. Это можно проверить утилитой top.
Убедитесь, что утилита pidstat вводит в заблуждение, показывает небольшую нагрузку:

```
root@tantor:~# pidstat -w | grep i.sh
06:58:52 AM  1000    212239      0.00      0.02  i.sh
06:58:52 AM  1000    212242      0.00      0.02  i.sh
06:58:52 AM  1000    212243      0.00      0.02  i.sh
06:58:52 AM  1000    212244      0.00      0.02  i.sh
```

5) Посмотрите, что показывает утилита vmstat:

```
root@tantor:~# vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 7  0      0 179276 271824 720548    0    0     2     3   75   78  2  0 98  0  0
```

Утилита vmstat показывает отсутствие нагрузки и небольшое число переключений контекстов. При том, что число переключений довольно большое.

Выполните команду с интервалом примерно в секунду:

```
astra@tantor:~$ grep ctxt /proc/22242/status
voluntary_ctxt_switches:        0
nonvoluntary_ctxt_switches:     3841
astra@tantor:~$ grep ctxt /proc/22242/status
voluntary_ctxt_switches:        0
nonvoluntary_ctxt_switches:     3945
```

Контексты переключаются с частотой 100 раз в секунду.

6) Установите для процессов политику планировщика SCED_RR (Round Robin Scheduler) - timeslice выделенный процессам одинаков и равен значению параметра `kernel.sched_rr_timeslice_ms`:

```
astra@tantor:~# sudo chrt -r -p 10 212242
astra@tantor:~# sudo chrt -r -p 10 212239
astra@tantor:~# sudo chrt -r -p 10 212243
astra@tantor:~# sudo chrt -r -p 10 212244
```

Процессы с политикой SCHED_DEADLINE могут вытеснять процессы с SCHED_FIFO и SCHED_RR, то есть являются наиболее приоритетными.

7) Проверьте результат выполнения команды:

```
astra@tantor:~$ chrt -p 212242
pid 212242's current scheduling policy: SCHED_RR
pid 212242's current scheduling priority: 10
```

8) Выполните команду с интервалом примерно в секунду:

```
root@tantor:~# grep ctxt /proc/212233/status
voluntary_ctxt_switches:        0
nonvoluntary_ctxt_switches:     27518
root@tantor:~# grep ctxt /proc/212233/status
voluntary_ctxt_switches:        0
nonvoluntary_ctxt_switches:     27523
```

Контексты стали переключаться с частотой 10 раз в секунду

9) Посмотрите timeslice планировщика RR:

```
astra@tantor:~$ sudo sysctl -a | grep kernel.sched_rr_timeslice_ms
kernel.sched_rr_timeslice_ms = 100
```

timeslice 100 миллисекунд = 1/10 секунды, что соответствует переключению 10 раз в секунду.

10) Установите timeslice в одну секунду:

```
astra@tantor:~# sudo sysctl kernel.sched_rr_timeslice_ms=1000
kernel.sched_rr_timeslice_ms = 1000
```

Частота переключений контекста не будет чаще 100 раз в секунду независимо от настроек. Это правильно, так как позволяет не производить большие накладные расходы на работу планировщика по переключению контекстов.

11) Проверьте, что контексты процессов с RR политикой стали переключаться раз в секунду. Выполните команды с интервалом раз в секунду и посмотрите насколько поменяются значения счетчика принудительных переключений контекста:

```
root@tantor:~# grep ctxt /proc/212233/status
```

```
voluntary_ctxt_switches:        0
nonvoluntary_ctxt_switches:     62348
root@tantor:~# grep ctxt /proc/212233/status
voluntary_ctxt_switches:        0
nonvoluntary_ctxt_switches:     62349
```

Значения будут увеличиваться на единицу с частотой раз в секунду.

12) Запустите команду и примерно через 20 секунд прервите её выполнение набрав <ctrl+c>:

```
root@tantor:~# perf stat -p 212233
^C
 Performance counter stats for process id '22381':

          9,340.10 msec task-clock:u                     #    0.477 CPUs utilized
                 0      context-switches:u               #    0.000 /sec
                 0      cpu-migrations:u                 #    0.000 /sec
                 0      page-faults:u                    #    0.000 /sec
   <not supported>      cycles:u
   <not supported>      instructions:u
   <not supported>      branches:u
   <not supported>      branch-misses:u

      19.595645993 seconds time elapsed
```

Утилита `perf` выдает верные данные ноль произвольных переключений контекстов в секунду. Число принудительных переключений, которые более ценны для диагностики не показывает, чем вводит в заблуждение.
Утилита top при этом показывает правильные данные с достаточной точностью для частоты переключений контекстов в 1 секунду: загрузка ядер процессоров доходит до 100%. Суммарная загрузка всех ядрер %CPU по ~50% у каждого процесса i.sh

```
top - 23:04:28 up  3:03,  3 users,  load average: 4.08, 3.65, 2.91
Tasks: 173 total,   6 running, 167 sleeping,   0 stopped,   0 zombie
%Cpu0  :  1.0 us,  0.7 sy,  0.0 ni, 98.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  : 94.0 us,  1.0 sy,  0.0 ni,  5.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3927.2 total,   1439.9 free,   1005.2 used,   1898.6 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   2922.0 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  22381 astra     -11   0    7580   3200   2944 R  58.8   0.1  10:25.68 i.sh
  22380 astra     -11   0    7580   3200   2944 R  35.9   0.1  10:30.74 i.sh
     16 root      20   0       0      0      0 R   0.7   0.0   4:52.74 rcu_preempt
   6420 astra     20   0  257.0g 139696 108444 S   0.7   3.5   1:58.75 fly-start-menu
```

13) Остановите процессы:

```
astra@tantor:~$ killall i.sh
[1]-  Terminated              taskset./i.sh
[2]+  Terminated              taskset./i.sh
```

В этой части практики вы изучили как можно протестировать результат изменений параметров планировщика. По приведённому примеру можно протестировать работу планировщиков `SCHED_OTHER, SCHED_FIFO, SCHED_DEADLINE.`
Политика `SCHED_OTHER` (и её производные `SCHED_BATCH, SCHED_IDLE`) не являются политиками реального времени. **К политикам реального времени** (для интерактивных задач, где важна отзывчивость - получать timeslice с какой-то частотой) относятся: `SCHED_FIFO` - вытесняется только процессами с более высоким приоритетом или политикой `SCHED_DEADLINE`. Установка процессу политики `SCHED_FIFO` приводит к тому, что процесс может надолго занять ядро.

Практика к главе 4

Часть 1. Параметры дисковой подсистемы

1) Блочные устройства располагаются в директории /dev смонтированной на виртуальной файловой системе devtmpfs. Посмотрите какие устройства из /dev смонтированы:

```
root@tantor:~# mount | grep /dev
udev on /dev type devtmpfs
(rw,nosuid,relatime,size=1966296k,nr_inodes=491574,mode=755,inode64)
devpts on /dev/pts type devpts
(rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000)
/dev/sda1 on / type ext4 (rw,relatime)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev,inode64)
hugetlbfs on /dev/hugepages type hugetlbfs (rw,relatime,pagesize=2M)
mqueue on /dev/mqueue type mqueue (rw,nosuid,nodev,noexec,relatime)
```

В /dev находятся файлы только тех устройств, которые в настоящий момент доступны (подключены). Если устройство отключается, то файл удаляется из /dev.
В виртуальной машине смонтирован первый раздел **/dev/sda**.

2) Посмотрите список блочных устройств:

```
root@tantor:~# ls -l /dev | grep br
brw-rw----  1 root    disk      7,   0 loop0
brw-rw----  1 root    disk      7,   1 loop1
brw-rw----  1 root    disk      7,   2 loop2
brw-rw----  1 root    disk      7,   3 loop3
brw-rw----  1 root    disk      7,   4 loop4
brw-rw----  1 root    disk      7,   5 loop5
brw-rw----  1 root    disk      7,   6 loop6
brw-rw----  1 root    disk      7,   7 loop7
brw-rw----  1 root    disk    259,   0 nvme0n1
brw-rw----  1 root    disk      8,   0 sda
brw-rw----  1 root    disk      8,   1 sda1
brw-rw----+ 1 root    cdrom    11,   0 sr0
```

В виртуальной машине подключен виртуальный диск /dev/sda с одним разделом /dev/sda1. Первая буква b обозначает block device. Вместо размера файла выдаются два числа: тип и порядковый номер (или режим работы) устройства.

3) Посмотрите содержимое директории `/sys/dev/block`:

```
root@tantor:~# ls -al /sys/dev/block
total 0
drwxr-xr-x 2 root root 0 .
drwxr-xr-x 4 root root 0 ..
lrwxrwxrwx 1 root root 0 11:0 ->
../../devices/pci0000:00/0000:00:01.1/ata2/host1/target1:0:0/1:0:0:0/block/sr0
lrwxrwxrwx 1 root root 0 259:0 ->
../../devices/pci0000:00/0000:00:0e.0/nvme/nvme0/nvme0n1
lrwxrwxrwx 1 root root 0 7:0 -> ../../devices/virtual/block/loop0
lrwxrwxrwx 1 root root 0 7:1 -> ../../devices/virtual/block/loop1
lrwxrwxrwx 1 root root 0 7:2 -> ../../devices/virtual/block/loop2
lrwxrwxrwx 1 root root 0 7:3 -> ../../devices/virtual/block/loop3
lrwxrwxrwx 1 root root 0 7:4 -> ../../devices/virtual/block/loop4
lrwxrwxrwx 1 root root 0 7:5 -> ../../devices/virtual/block/loop5
lrwxrwxrwx 1 root root 0 7:6 -> ../../devices/virtual/block/loop6
lrwxrwxrwx 1 root root 0 7:7 -> ../../devices/virtual/block/loop7
lrwxrwxrwx 1 root root 0 8:0 ->
../../devices/pci0000:00/0000:00:0d.0/ata1/host0/target0:0:0/0:0:0:0/block/sda
```

```
lrwxrwxrwx 1 root root 0 8:1 ->
../../devices/pci0000:00/0000:00:0d.0/ata1/host0/target0:0:0/0:0:0:0/block/sda/sda1
```

Директория содержит символические ссылки на блочные устройства. Устройство sda подключено к шине PCI через интерфейс ATA.

4) Посмотрите какой используется планировщик ввода-вывода с устройством sda:

```
root@tantor:~# cat /sys/block/sda/queue/scheduler
[none] mq-deadline
root@tantor:~# cat /sys/dev/block/8:0/queue/scheduler
[none] mq-deadline
```

С устройством sda используется планировщик с названием `none`.

5) Выполните следующие команды:

```
root@tantor:~# echo kyber > /sys/block/sda/queue/scheduler
root@tantor:~# cat /sys/dev/block/8:0/queue/scheduler
none mq-deadline [kyber]
root@tantor:~# echo bfq > /sys/block/sda/queue/scheduler
root@tantor:~# cat /sys/block/sda/queue/scheduler
none mq-deadline kyber [bfq]
root@tantor:~# echo abc > /sys/block/sda/queue/scheduler
-bash: echo: write error: Invalid argument
root@tantor:~# echo none > /sys/block/sda/queue/scheduler
root@tantor:~# cat /sys/block/sda/queue/scheduler
[none] mq-deadline kyber bfq
```

Команды меняют планировщик. Несуществующий планировщик abc указать нельзя. Изначально показывались не все доступные планировщики, сейчас показываются.

6) Посмотрите содержимое файла `/etc/udev/rules.d/70-schedulerset.rules`, в котором устанавливаются параметры выбора планировщика ввода-вывода:

```
 root@tantor:~# cat /etc/udev/rules.d/70-schedulerset.rules
ACTION=="add|change", KERNEL=="sd[a-z]", ATTR{queue/rotational}=="0",
ATTR{queue/scheduler}="none"
```

Для всех устройств с названиями `sda..sdz` указан планировщик **none**.

7) Посмотрите размер физического сектора устройства:

```
root@tantor:~# lsblk -td
NAME ALIGNMENT MIN-IO OPT-IO PHY-SEC LOG-SEC ROTA SCHED       RQ-SIZE  RA WSAME
sda          0    512      0     512     512    0 none             32 128    0B
sr0          0    512      0     512     512    1 mq-deadline       2 128    0B
```

Linux считает, что размер сектора устройства `sda` 512 байт.

8) Посмотрите параметры, с которыми была смонтирована файловая система на разделе `/dev/sda1`:

```
root@tantor:~# tune2fs -l /dev/sda1 | grep opt
Default mount options:    user_xattr acl
root@tantor:~# mount | grep ext4
/dev/sda1 on / type ext4 (rw,relatime)
root@tantor:~# cat /etc/fstab | grep ext4
UUID=aca1a090-eba2-49ba-a8fc-ba12e9e2bf26 /   ext4    defaults    1 1
```

Файловая система смонтирована с параметрами по умолчанию. Параметры, которые определяют функционал файловой системы могут храниться в самом разделе (в "суперблоке").
9) Посмотрите полный список параметров:

```
root@tantor:~# cat /proc/fs/ext4/sda1/options
rw
bsddf
nogrpid
block_validity
dioread_nolock
nodiscard
delalloc
nowarn_on_error
journal_checksum
barrier
auto_da_alloc
user_xattr
acl
noquota
resuid=0
resgid=0
errors=continue
commit=5
min_batch_time=0
max_batch_time=15000
stripe=0
data=ordered
inode_readahead_blks=32
init_itable=10
max_dir_size_kb=0
```

10) Установите свойство discard в суперблоке файловой системы и проверьте, что оно установлено:

```
root@tantor:~# tune2fs -o +discard /dev/sda1
tune2fs 1.47.0 (5-Feb-2023)
root@tantor:~# tune2fs -l /dev/sda1 | grep opt
Default mount options:    user_xattr acl discard
```

К опциям, которые были добавилось свойство discard.
Опция discard может устанавливаться:
в суперблоке файловой системы как опция монтирования по умолчанию
в файле параметров монтирования файловой системы — /etc/fstab
в конфигурации cryptsetup — /etc/crypttab
в конфигурации LVM — /etc/lvm/lvm.conf
в конфигурации загрузчика — /boot/grub/grub.cfg

11) Проверьте, посылались ли команды discard операционной системой в контроллер sda:

```
root@tantor:~# fstrim -v /
fstrim: /: the discard operation is not supported
root@tantor:~# lsblk --discard
NAME   DISC-ALN DISC-GRAN DISC-MAX DISC-ZERO
sda            0        0B       0B         0
└─sda1         0        0B       0B         0
sr0            0        0B       0B         0
```

Нули в DISC-GRAN и DISC-MAX означают, что discard не использовался.

Команды discard не посылались.

В виртуальной машине, на которой выполняются практики, разделы являются файлами. Файлы занимают место в файловой системе хоста, поэтому discard не используется.

12) Посмотрите как называется пакет, в котором устанавливалась утилита fstrim:

```
root@tantor:~# dpkg -S fstrim
util-linux: /sbin/fstrim
util-linux: /usr/share/man/man8/fstrim.8.gz
util-linux: /lib/systemd/system/fstrim.service
util-linux: /lib/systemd/system/fstrim.timer
util-linux: /usr/share/bash-completion/completions/fstrim
```

Пакет называется `util-linux`

Часть 2. Установка пакетов в Astralinux

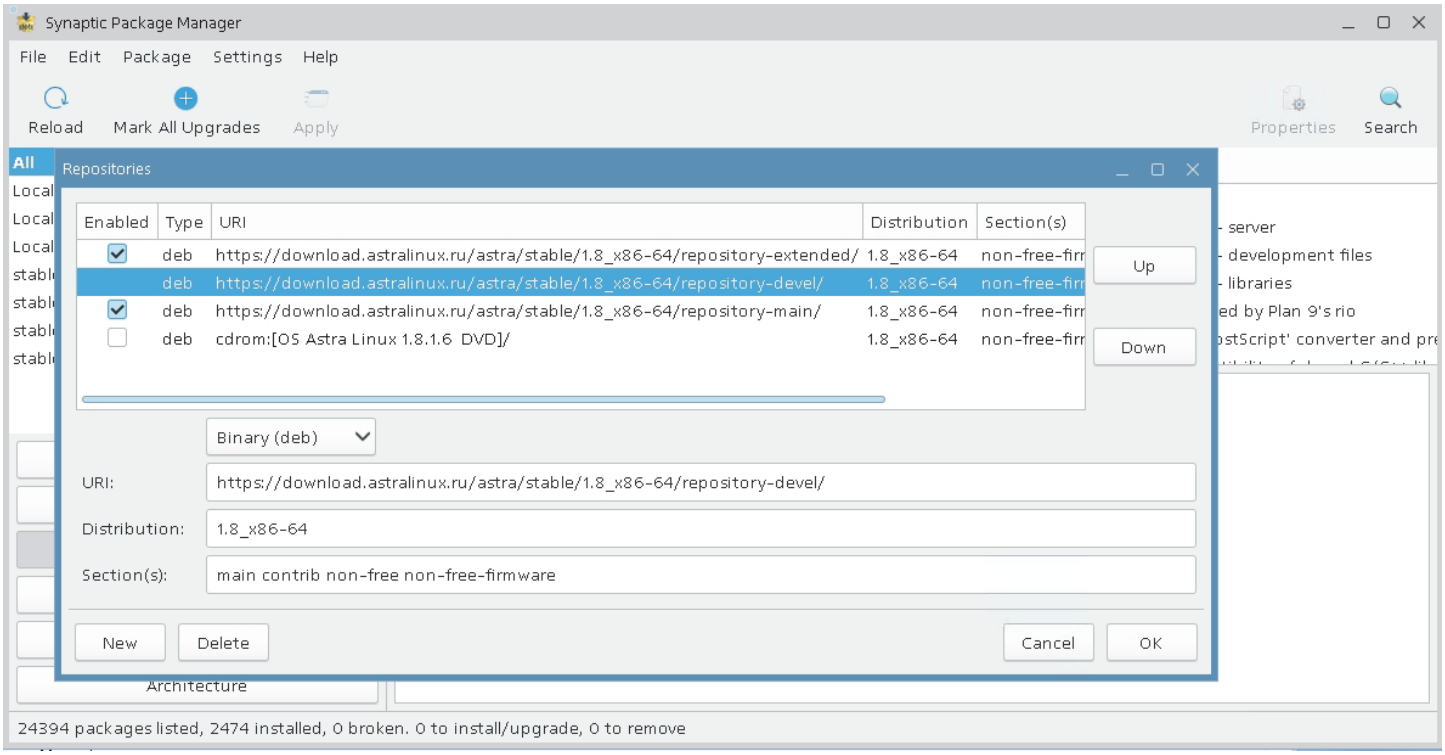
1) Запустите новый терминал. В терминале пользователя astra запустите утилиту synaptic-pkexec:

```
astra@tantor:~$ synaptic-pkexec
```

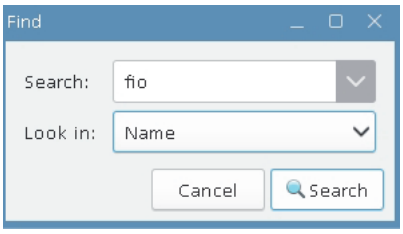
В окне ввода пароля введите **astra** и нажмите кнопку OK

Synaptic это графический пакетный менеджер в Astralinux.

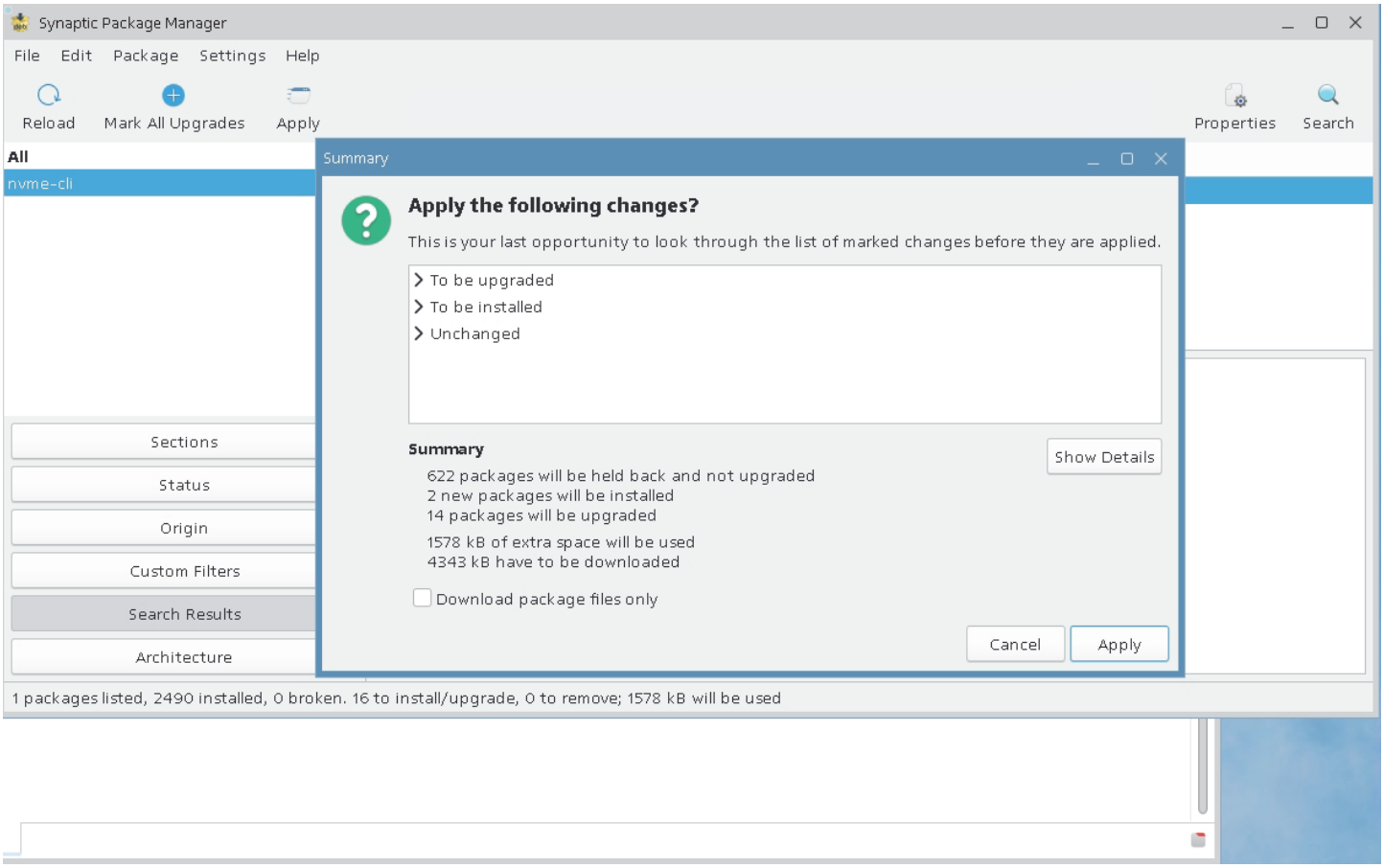
2) В меню Settings -> Repositories проверьте, что установлены две галочки на основном и расширенном репозиториях. Нажмите кнопку OK.



3) В меню Edit -> Search наберите fio. В поле Look in выберите Name. Нажмите кнопку Search:



4) В высветившимся списке пакетов выберите пакет fio. Если утилита установлена (есть зеленый кружок перед названием утилиты), то пропустите этот пункт. Также можно установить пакет uuid-runtime, если пакет uuid-runtime не установлен.
Если утилита fio не установлена, то в выпадающем меню выберите "Mark for installation". Если окно не закрылось нажмите Mark. На toolbar (три кнопки под меню) нажмите кнопку Apply. В появившемся окне нажмите кнопку Apply. Если появится окно с вопросом про Kernel update нажмите Next.



5) Закройте окно Synaptic нажав крестик справа вверху окна приложения Synaptic.

6) В окне терминала пользователя astra наберите команду:

```
astra@tantor:~$ sudo apt install fio -y
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
fio is already the newest version (3.33-3+b1).
0 upgraded, 0 newly installed, 0 to remove and 636 not upgraded.
```

В примере команда показывает, что пакет fio установлен.
Закройте терминал пользователя astra набрав в нем комбинацию клавиш <**ctrl+d**>

7) Установите пакеты nvme-cli и uuid-runtime:

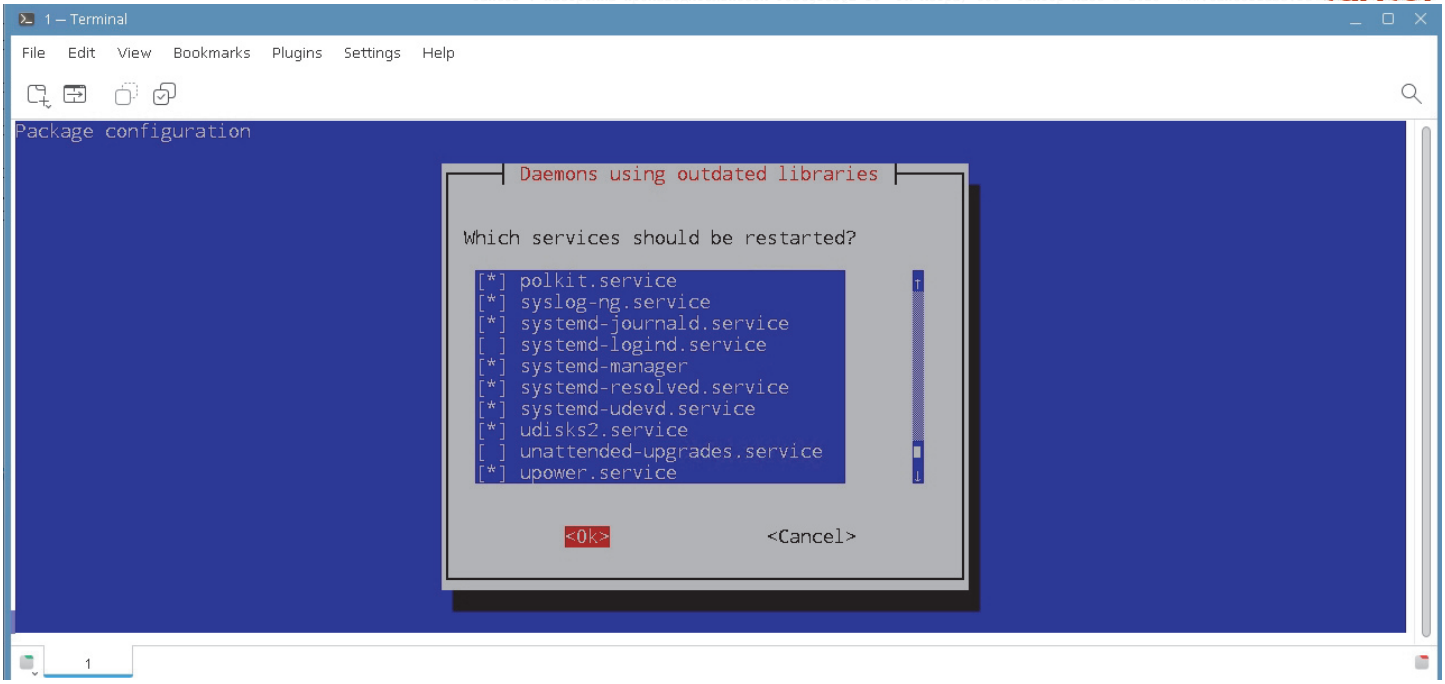
```
root@tantor:~# apt install nvme-cli uuid-runtime -y
```

Если откроется окно с псевдографикой синего цвета, нажмите OK в первом окне:

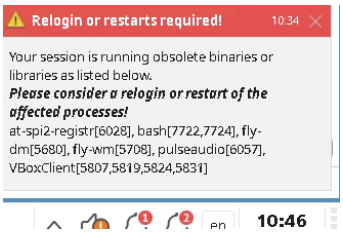
Окно уведомляет о том, что нужно обновить все пакеты для которых вышли обновления. Убрать уведомления можно деинсталлировав пакет `apt -y remove needrestart`

Активный элемент окон псевдографики подсвечивается красным цветом. Для перехода между визуальными элементами окна используется клавиша <TAB>, клавиши стрелок на клавиатуре или мышь.

Откроется следующее окно. Нажмите клавишу <TAB>, в окне подсветится визуальный элемент <OK>.
Служба uuidd будет запущена без перезагрузки linux и предлагается перезагрузить службы, которые зависят от того, что было выполнено при установке пакета. Набор служб менять не нужно. Если подсвечен красным цветом элемент окна <OK>, то нажмите на клавиатуре клавишу <ENTER> или кликните мышкой на элемент окна <OK.



Сообщение о желательности перелогиниться можно игнорировать и закрыть:



Была установлена утилита командной строки nvme и служба uuidd.
Утилита nvme, устанавливаемая в пакете nvme-cli используется для чтения характеристик и метрик устройств NVMe. Утилита используется для определения состояния чипов памяти, отслеживания срока службы, обновления прошивки, стирания содержимого, переинициализации, чтения журналов с ошибками.
Пакет uuidd обеспечивает уникальность генерируемых uuid на хостах с несколькими процессорами. Он устанавливается потому, что требуется для установки nvme-cli.
Утилита nvme используется для получения информации об устройствах NVMe.

Вы научились устанавливать пакеты в Astralinux.

Часть 3. Работа с SSD и тестирование производительности диска утилитой fio

1) Выполните несколько команд с помощью утилиты `nvme`:

```
root@tantor:~# nvme list
Nod           Generic     SN           Model                Namespace Usage Format                   FW Rev
------------ ---------- ------------ -------------------- --------- ----- ---------------------- --------
/dev/nvme0n1 /dev/ng0n1 VB1234-56789 ORCL-VBOX-NVME-VER12              1 8.59GB/8.59GB 512B + 0B 1.0
root@tantor:~# nvme list-subsys
nvme-subsys0 - NQN=nqn.2014.08.org.nvmexpress:80ee80eeVB1234-56789 ORCL-VBOX-NVME-VER12
\
 +- nvme0 pcie 0000:00:0e.0 live
root@tantor:~# nvme id-ctrl /dev/nvme0n1 | head -6
NVME Identify Controller:
vid       : 0x80ee
ssvid     : 0x80ee
sn        : VB1234-56789
mn        : ORCL-VBOX-NVME-VER12
fr        : 1.0
```

2) Посмотрите список устройств nvme:

```
root@tantor:~# ls /dev/nvm*
/dev/nvme0  /dev/nvme0n1  /dev/nvme-fabrics
```

Имеется одно устройство NVMe.

3) Создайте файловую систему на устройстве NVMe:

```
root@tantor:~# mkfs -E discard /dev/nvme0n1
mke2fs 1.47.0 (5-Feb-2023)
Creating filesystem with 2097152 4k blocks and 524288 inodes
Filesystem UUID: 0c0f68b7-d1d9-47ff-908c-9831503837f3
Superblock backups stored on blocks: 32768, 98304, 163840, 229376, 294912,
819200, 884736, 1605632
Allocating group tables: done
Writing inode tables: done
Writing superblocks and filesystem accounting information: done
```

Опция используется по умолчанию. Приведена для того, чтобы подчеркнуть важность очистки содержимого SSD. В описании команды (man mkfs.ext4) указано, что опция discard посылает на контроллер устройства команду очистки содержимого:
Attempt to discard blocks at mkfs time (discarding blocks initially is useful on solid state devices and sparse / thin-provisioned storage). When the device advertises that discard also zeroes data (any subsequent read after the discard and before write returns zero), then mark all not-yet-zeroed inode tables as zeroed. This significantly speeds up file system initialization. This is set as default.

Обратите внимание, что журнал не был создан, отсутствует строка:
`Creating journal (16384 blocks): done`

4) Измените суперблок файловой системы, включив опцию discard:

```
root@tantor:~# tune2fs -o +discard /dev/nvme0n1
tune2fs 1.47.0 (5-Feb-2023)
```

5) Создайте директорию `/u01`, смонтируйте в неё созданную файловую систему, посмотрите какой размер у смонтированного раздела:

```
root@tantor:~# mkdir /u01
root@tantor:~# mount /dev/nvme0n1 /u01
root@tantor:/# df -Th | grep "^/dev"
/dev/sda1      ext4       47G   16G   29G  36% /
/dev/nvme0n1   ext2      7.9G   24K  7.5G   1% /u01
```

Раздел смонтирован в `/u01`, его размер 8Гб, файловая система **ext2**.

6) В терминале пользователя root выполните команду:

```
root@tantor:~# fio --ioengine=psync --filename=/tmp/test --size=1G --time_based --
-name=fio --group_reporting --runtime=10 --direct=1 --sync=1 --iodepth=1 --
rw=read --bs=8k --numjobs=1
fio: (g=0): rw=read, bs=(R) 8192B-8192B, (W) 8192B-8192B, (T) 8192B-8192B, ioengine=psync, iodepth=1
fio-3.33
Starting 1 process
fio: Laying out IO file (1 file / 1024MiB)
Jobs: 1 (f=1): [R(1)][100.0%][r=36.4MiB/s][r=4664 IOPS][eta 00m:00s]
fio: (groupid=0, jobs=1): err= 0: pid=7432: Mon Dec  9 00:07:59 2024
  read: IOPS=4682, BW=36.6MiB/s (38.4MB/s)(366MiB/10001msec)
    clat (usec): min=92, max=8607, avg=207.12, stdev=306.94
     lat (usec): min=93, max=8608, avg=208.56, stdev=306.95
    clat percentiles (usec):
     |  1.00th=[  143],  5.00th=[  149], 10.00th=[  151], 20.00th=[  155],
```

```
     | 30.00th=[  157], 40.00th=[  159], 50.00th=[  159], 60.00th=[  161],
     | 70.00th=[  165], 80.00th=[  172], 90.00th=[  237], 95.00th=[  253],
     | 99.00th=[ 2245], 99.50th=[ 2900], 99.90th=[ 3490], 99.95th=[ 3654],
     | 99.99th=[ 3982]
   bw (  KiB/s): min=33213, max=40688, per=100.00%, avg=37505.37, stdev=1932.35, samples=19
   iops        : min= 4151, max= 5086, avg=4687.95, stdev=241.58, samples=19
  lat (usec)   : 100=0.01%, 250=94.32%, 500=3.66%, 750=0.61%, 1000=0.03%
  lat (msec)   : 2=0.15%, 4=1.22%, 10=0.01%
  cpu          : usr=12.33%, sys=11.35%, ctx=46830, majf=0, minf=14
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=46827,0,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
   READ: bw=36.6MiB/s (38.4MB/s), 36.6MiB/s-36.6MiB/s (38.4MB/s-38.4MB/s), io=366MiB (384MB), run=10001-
10001msec

Disk stats (read/write):
  sda: ios=45651/4, merge=0/1, ticks=8268/7, in_queue=8282, util=97.89%
```

Красным выделены результаты теста. Остальной многословный вывод не требует внимания и используется в редких случаях.

Параметры теста:
Работа PostgreSQL с файловой системой соответствует **`--ioengine=psync`**
размер блока PostgreSQL **`--bs=8k`**
Размер очереди (`iodepth`) выбран 1, так как выбран direct=1
Тест выполнял только чтение из файла. Результат теста: максимальное число операций ввода-вывода `IOPS=4682`, средняя скорость `38.4MB/s`.

7) Протестируйте раздел NVMe:

```
root@tantor:~# fio --ioengine=psync --filename=/u01/test --size=1G --time_based --name=fio --group_reporting --runtime=10 --direct=1 --sync=1 --iodepth=1 --rw=read --bs=8k --numjobs=1
```

```
fio: (g=0): rw=read, bs=(R) 8192B-8192B, (W) 8192B-8192B, (T) 8192B-8192B, ioengine=psync, iodepth=1
fio-3.33
Starting 1 process
fio: Laying out IO file (1 file / 1024MiB)
Jobs: 1 (f=1): [R(1)][100.0%][r=43.1MiB/s][r=5520 IOPS][eta 00m:00s]
fio: (groupid=0, jobs=1): err= 0: pid=7441: Mon Dec  9 00:08:59 2024
  read: IOPS=5312, BW=41.5MiB/s (43.5MB/s)(415MiB/10001msec)
    clat (usec): min=77, max=11930, avg=181.67, stdev=300.00
     lat (usec): min=78, max=11932, avg=183.12, stdev=300.04
    clat percentiles (usec):
     |  1.00th=[  128],  5.00th=[  133], 10.00th=[  137], 20.00th=[  139],
     | 30.00th=[  141], 40.00th=[  143], 50.00th=[  145], 60.00th=[  147],
     | 70.00th=[  149], 80.00th=[  151], 90.00th=[  167], 95.00th=[  229],
     | 99.00th=[ 2147], 99.50th=[ 2835], 99.90th=[ 3523], 99.95th=[ 3752],
     | 99.99th=[ 5932]
   bw (  KiB/s): min=38848, max=43936, per=99.80%, avg=42413.47, stdev=1514.01, samples=19
   iops        : min= 4856, max= 5492, avg=5301.68, stdev=189.25, samples=19
  lat (usec)   : 100=0.11%, 250=97.55%, 500=0.99%, 750=0.12%, 1000=0.08%
  lat (msec)   : 2=0.08%, 4=1.04%, 10=0.02%, 20=0.01%
  cpu          : usr=12.13%, sys=12.54%, ctx=53130, majf=0, minf=15
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=53128,0,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
   READ: bw=41.5MiB/s (43.5MB/s), 41.5MiB/s-41.5MiB/s (43.5MB/s-43.5MB/s), io=415MiB (435MB), run=10001-
10001msec

Disk stats (read/write):
  nvme0n1: ios=52790/2, merge=0/1, ticks=8283/5, in_queue=8289, util=98.93%
```

Физически виртуальная машина хранит `/dev/sda` и `/dev/nvme0` в соседних файлах VMDK в файловой системе хоста. Используются разные типы шин ввода-вывода: SATA и NVMe. Заметная

разница в производительности ~10%. Практический результат в том, что использование NVMe в VirtualBox имеет преимущество по сравнению с SATA AHCI.

8) Были созданы временные файлы `/tmp/test` и `/u01/test` размером 1Гб каждый. Удалите эти файлы:

```
root@tantor:~# rm -f /tmp/test
root@tantor:~# rm -f /u01/test
```

9) Размонтируйте /u01:

```
root@tantor:~# umount /u01
```

10) Этот пункт выполнять необязательно.
Пример тестирования записи в пределах раздела. Раздел должен быть отмонтирован и после теста пересоздан, так как содержимое раздела будет затёрто:

```
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=4k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=1 | grep BW=
  write: IOPS=583, BW=2335KiB/s (2391kB/s)(22.8MiB/10002msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=8k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=1 | grep BW=
  write: IOPS=561, BW=4491KiB/s (4599kB/s)(43.9MiB/10002msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=16k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=1 | grep BW=
  write: IOPS=504, BW=8066KiB/s (8260kB/s)(78.8MiB/10001msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=32k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=1 | grep BW=
  write: IOPS=411, BW=12.9MiB/s (13.5MB/s)(129MiB/10002msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=64k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=1 | grep BW=
  write: IOPS=321, BW=20.1MiB/s (21.1MB/s)(201MiB/10002msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=128k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=1 | grep BW=
  write: IOPS=276, BW=34.5MiB/s (36.2MB/s)(346MiB/10004msec); 0 zone resets

root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=4k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=0 | grep BW=
  write: IOPS=26.6k, BW=104MiB/s (109MB/s)(1038MiB/10002msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=8k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=0 | grep BW=
  write: IOPS=16.6k, BW=130MiB/s (136MB/s)(1296MiB/10007msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=16k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=0 | grep BW=
  write: IOPS=9845, BW=154MiB/s (161MB/s)(1539MiB/10006msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=32k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=0 | grep BW=
  write: IOPS=8015, BW=250MiB/s (263MB/s)(2508MiB/10015msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=64k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=0 | grep BW=
  write: IOPS=4582, BW=286MiB/s (300MB/s)(2864MiB/10001msec); 0 zone resets
```

В примерах тестировалась параллельная произвольная запись блоков разного размера с fsync и без fsync. Опция fsync передаёт пожелание контроллеру устройства произвести запись, очистив внутренние кэши устройства хранения. Гарантия записи определяется контроллером устройства. Утилита fio может читать и писать как в файлы, так и напрямую работать с разделами дисков и с устройствами. Это позволяет измерять работу физических интерфейсов и устройств без задействования страничного кэша linux.

Часть 4. Тестирование журнала быстрой фиксации ext4

1) Создайте журналируемую файловую систему **ext4** на разделе **/dev/nvme0n1**:

```
root@tantor:~# mkfs.ext4 /dev/nvme0n1
mke2fs 1.47.0 (5-Feb-2023)
/dev/nvme0n1 contains a ext2 file system
Proceed anyway? (y,N) y
Creating filesystem with 2097152 4k blocks and 524288 inodes
```

Filesystem UUID: 9c8c5140-2744-4f2f-acc6-baeab282efd6
Superblock backups stored on blocks:
32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done

Примечание: добавление журнала возможно без форматирования командой:
tune2fs -O +has_journal /dev/nvme0n1
Данные в файловой системе сохраняются.

2) Посмотрите характеристики созданной файловой системы:

```
root@tantor:/# dumpe2fs /dev/nvme0n1 | head -55
dumpe2fs 1.47.0 (5-Feb-2023)
Filesystem volume name:   <none>
Last mounted on:          <not available>
Filesystem UUID:          d38ab2fb-3ba2-4593-9ce4-e5ce9ed6c06c
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode dir_index filetype extent 64bit flex_bg sparse_super
large_file huge_file dir_nlink extra_isize metadata_csum
Filesystem flags:         signed_directory_hash
Default mount options:    user_xattr acl
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              524288
Block count:              2097152
Reserved block count:     104857
Overhead clusters:        58505
Free blocks:              2038641
Free inodes:              524277
First block:              0
Block size:               4096
Fragment size:            4096
Group descriptor size:    64
Reserved GDT blocks:      1023
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         8192
Inode blocks per group:   512
Flex block group size:    16
Filesystem created:       Sun Dec  8 01:14:49 2035
Last mount time:          n/a
Last write time:          Sun Dec  8 01:14:49 2035
Mount count:              0
Maximum mount count:      -1
Last checked:             Sun Dec  8 01:14:49 2035
Check interval:           0 (<none>)
Lifetime writes:          4129 kB
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
First inode:              11
Inode size:               256
Required extra isize:     32
Desired extra isize:      32
Journal inode:            8
Default directory hash:   half_md4
Directory Hash Seed:      3e913b49-f24d-4c17-a94d-b66abf6f0e9e
Journal backup:           inode blocks
Checksum type:            crc32c
Checksum:                 0x37fd9c6b
Journal features:         (none)
Total journal size:       64M
Total journal blocks:     16384
Max transaction length:   16384
Fast commit length:       0
Journal sequence:         0x00000001
Journal start:            0
```

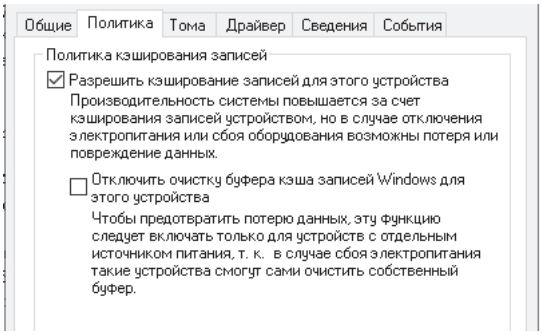
3) Смонтируйте созданную файловую систему в /u01:

```
root@tantor:/# mount /dev/nvme0n1 /u01
```

4) Перейдите в корень смонтированной файловой системы, создайте директорию. перейдите в неё и запустите тест скорости записи в WAL-файлы:

```
root@tantor:/# cd /u01
root@tantor:/u01# mkdir a1
root@tantor:/u01# cd a1
root@tantor:/u01/a1# pg_test_fsync

5 seconds per test
O_DIRECT supported on this platform for open_datasync and open_sync.

Compare file sync methods using one 8kB write:
(in wal_sync_method preference order, except fdatasync is Linux's default)
        open_datasync                     566.595 ops/sec    1765 usecs/op
        fdatasync                         567.752 ops/sec    1761 usecs/op
        fsync                             194.505 ops/sec    5141 usecs/op
        fsync_writethrough                            n/a
        open_sync                         202.077 ops/sec    4949 usecs/op

Compare file sync methods using two 8kB writes:
(in wal_sync_method preference order, except fdatasync is Linux's default)
        open_datasync                     287.440 ops/sec    3479 usecs/op
        fdatasync                         506.992 ops/sec    1972 usecs/op
        fsync                             189.935 ops/sec    5265 usecs/op
        fsync_writethrough                            n/a
        open_sync                          97.948 ops/sec   10209 usecs/op

Compare open_sync with different write sizes:
(This is designed to compare the cost of writing 16kB in different write
open_sync sizes.)
         1 * 16kB open_sync write         185.468 ops/sec    5392 usecs/op
         2 *  8kB open_sync writes        100.674 ops/sec    9933 usecs/op
         4 *  4kB open_sync writes         48.484 ops/sec   20625 usecs/op
         8 *  2kB open_sync writes         25.928 ops/sec   38569 usecs/op
        16 *  1kB open_sync writes         12.982 ops/sec   77033 usecs/op

Test if fsync on non-write file descriptor is honored:
(If the times are similar, fsync() can sync data written on a different
descriptor.)
        write, fsync, close               192.591 ops/sec    5192 usecs/op
        write, close, fsync               195.865 ops/sec    5106 usecs/op

Non-sync'ed 8kB writes:
        write                          450551.547 ops/sec       2 usecs/op
```

Утилита **pg_test_fsync** тестирует скорость записи различными способами в файлы в той директории из которой запущена. Она была запущена в директории **/u01/a1**
Создание поддиректории не влияет на результат теста.
Если вы получили значения на порядок больше: вместо 567 получили ~8000, то у вас в операционной системе хоста, в котором работает виртуальная машина, включено кэширование записи. Пример, если операционная система хоста (в котором работает виртуальная машина) Windows:



Если повторить тест несколько раз, то скорее всего повторяемость результатов будет низкой: разброс значений будет ~10%. Для выполнения практик это нормально, единственно нельзя сравнивать полученные тестами значения. Раздел nvme0 лежит в файле операционной системы хоста.

5) Посмотрите статистику обычного (jbd2) журнала:

```
root@tantor:/u01/a1# cat /proc/fs/jbd2/nvme0n1-8/info
10438 transactions (10436 requested), each up to 4096 blocks
average:
  0ms waiting for transaction
  0ms request delay
  4ms running transaction
  0ms transaction was being locked
  0ms flushing data (in ordered mode)
  0ms logging transaction
  5040us average transaction commit time
  1 handles per transaction
  1 blocks per transaction
  3 logged blocks per transaction
```

3 блока на транзакцию, 10438 транзакций. Число транзакций зависит от того сколько операций успела выполнить утилита **pg_test_fsync** за отведённое ей время.

6) Посмотрите статистику журнала fast commit ("fc"):

```
root@tantor:/u01/a1# cat /proc/fs/ext4/nvme0n1/fc_info
fc stats:
0 commits
0 ineligible
0 numblks
0us avg_commit_time
Ineligible reasons:
"Extended attributes changed":  0
"Cross rename": 0
"Journal flag changed": 0
"Insufficient memory":  0
"Swap boot":    0
"Resize":       0
"Dir renamed":  0
"Falloc range op":      0
"Data journalling":     0
"Encrypted filename":   0
```

Журнал быстрой фиксации отсутствует, поэтому статистика пустая.

7) Пересоздайте файловую систему на /dev/nvme0n1:

```
root@tantor:/u01# cd /
root@tantor:/# umount /u01
root@tantor:/# mkfs.ext4 /dev/nvme0n1
mke2fs 1.47.0 (5-Feb-2023)
/dev/nvme0n1 contains a ext4 file system
Proceed anyway? (y,N) y
Creating filesystem with 2097152 4k blocks and 524288 inodes
Filesystem UUID: 102365ba-9b22-4ac6-95f9-9f35c146e2ae
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done
```

8) Включите журнал быстрой фиксации и смонтируйте файловую систему:

```
root@tantor:/# tune2fs -O +fast_commit /dev/nvme0n1
tune2fs 1.47.0 (5-Feb-2023)
```

```
root@tantor:/# dumpe2fs /dev/nvme0n1 | grep commit
dumpe2fs 1.47.0 (5-Feb-2023)
Filesystem features:      has_journal ext_attr resize_inode dir_index fast_commit
filetype extent 64bit flex_bg sparse_super large_file huge_file dir_nlink
extra_isize metadata_csum
Fast commit length:       256
```

9) Смонтируйте файловую систему:

```
root@tantor:/# mount /dev/nvme0n1 /u01
```

10) Повторите тест:

```
root@tantor:/# cd /u01
root@tantor:/u01# mkdir a1
root@tantor:/u01# cd a1
root@tantor:/u01/a1# pg_test_fsync
5 seconds per test
O_DIRECT supported on this platform for open_datasync and open_sync.

Compare file sync methods using one 8kB write:
(in wal_sync_method preference order, except fdatasync is Linux's default)
        open_datasync                     509.642 ops/sec    1962 usecs/op
        fdatasync                         554.320 ops/sec    1804 usecs/op
        fsync                             290.867 ops/sec    3438 usecs/op
        fsync_writethrough                            n/a
        open_sync                         294.644 ops/sec    3394 usecs/op

Compare file sync methods using two 8kB writes:
(in wal_sync_method preference order, except fdatasync is Linux's default)
        open_datasync                     285.667 ops/sec    3501 usecs/op
        fdatasync                         505.847 ops/sec    1977 usecs/op
        fsync                             273.459 ops/sec    3657 usecs/op
        fsync_writethrough                            n/a
        open_sync                         143.576 ops/sec    6965 usecs/op

Compare open_sync with different write sizes:
(This is designed to compare the cost of writing 16kB in different write
open_sync sizes.)
         1 * 16kB open_sync write         273.442 ops/sec    3657 usecs/op
         2 *  8kB open_sync writes        144.108 ops/sec    6939 usecs/op
         4 *  4kB open_sync writes         74.927 ops/sec   13346 usecs/op
         8 *  2kB open_sync writes         37.400 ops/sec   26738 usecs/op
        16 *  1kB open_sync writes         19.074 ops/sec   52429 usecs/op

Test if fsync on non-write file descriptor is honored:
(If the times are similar, fsync() can sync data written on a different
descriptor.)
        write, fsync, close               288.831 ops/sec    3462 usecs/op
        write, close, fsync               292.142 ops/sec    3423 usecs/op

Non-sync'ed 8kB writes:
        write                          440343.998 ops/sec       2 usecs/op
```

11) Посмотрите статистику записи в журналы файловой системы:

```
root@tantor:/u01# cat /proc/fs/jbd2/nvme0n1-8/info
69 transactions (67 requested), each up to 4032 blocks
average:
  0ms waiting for transaction
  0ms request delay
  1376ms running transaction
  0ms transaction was being locked
  0ms flushing data (in ordered mode)
  0ms logging transaction
  5194us average transaction commit time
  278 handles per transaction
  1 blocks per transaction
  3 logged blocks per transaction
root@tantor:/u01# cat /proc/fs/ext4/nvme0n1/fc_info
```

```
fc stats:
16175 commits
62 ineligible
16175 numblks
1607us avg_commit_time
Ineligible reasons:
"Extended attributes changed":  0
"Cross rename": 0
"Journal flag changed": 0
"Insufficient memory":  0
"Swap boot":    0
"Resize":       0
"Dir renamed":  0
"Falloc range op":      0
"Data journalling":     0
"Encrypted filename":   0
```

3 блока на транзакцию, `69*3=207` блоков в журнале jbd2. Но **в дополнение** к этому журналу было записано 16175 блока в журнал быстрой фиксации. Запись в журнал не требовал отдельных операций fsync, запись выполнялась поблочно с признаком немедленной записи Force Unit Access (FUA). FUA применяется только для одного блока и не вызывает скидывание всего кэша контроллера устройства хранения.
В предыдущем тесте в журнал jbd2 было записано `10438*3=31314` блока за 10438 операций.

Что лучше {16382 и 31314 блока} или {16244 и 10438 операции записи}? SSD выполняют запись в чипы NAND блоками размера 16Кб (или больше), что равно 4 блокам (по 4Кб) журнала. Поэтому для SSD важнее число операций. Поэтому результаты проведённого теста примерно одинаковы: 567 и 554 ops/sec. У HDD размер блока 512 байт или 4Кб, но последовательная запись трёх блоков быстрее, чем одного из-за того, что во втором случае придётся ждать оборот диска. Для объединения дисков HDD с энергонезависимой памятью предсказать результат ещё сложнее. Поэтому критерием истины будет ли давать преимущество журнал быстрой фиксации или не будет является тестирование. В данном примере для тестирования использовалась утилита **pg_test_fsync**, которая предназначена для тестирования работы с WAL-файлами.

Журнал быстрой фиксации раскрывает свои преимущества на файловых системах, где часто меняются метаданные, то есть часто создаются, удаляются файлы и меняется их размер. Такое может происходить, например, в директории табличного пространства PostgreSQL где часто создаются и удаляются временные файлы. В директории, где находятся WAL файлы создаются нечасто. Для директорий, например, PGDATA или директории табличного пространства для временных файлов нужны другие тесты, а не `pg_test_fsync`.
Скорее всего, с такими директориями журнал быстрой фиксации даст преимущества с тем же уровнем отказоустойчивости.
У журнала JBD2 всегда записывается не меньше трёх блоков, поэтому FUA не может использоваться. Инициирование записи в журнал JBD2 вызывает опустошение всего кэша контроллера. Если в кэше контроллера успело накопиться больше 16Кб, запись теоретически может быть медленнее. Практически же скорость записи в чипы NAND сравнима со скоростью работы памяти, запись контроллером SSD может выполняться асинхронно.

12) Этот и следующие пункты до конца этой части практики опциональны.
До этого тестировали файловую систему ext4. Можно протестировать скорость работы без журнала (функционал ext2). Отключите журнал на `/dev/nvme0n1`:

```
root@tantor:/u01# cd /
root@tantor:/# umount /u01
root@tantor:/# tune2fs -O ^has_journal /dev/nvme0n1
root@tantor:/# dumpe2fs /dev/nvme0n1 | grep commit
dumpe2fs 1.47.0 (5-Feb-2023)
Filesystem features:   ext_attr resize_inode dir_index filetype extent 64bit
flex_bg sparse_super large_file huge_file dir_nlink extra_isize metadata_csum
Fast commit length:       0
```

В выводе команды отсутствуют свойства `has_journal` и `fast_commit`. Это означает, что файловая система без журнала.

13) Смонтируйте файловую систему:

```
root@tantor:/# mount /dev/nvme0n1 /u01
```

14) Повторите тест:

```
root@tantor:/# cd /u01
root@tantor:/u01# mkdir a1
root@tantor:/u01# cd a1
root@tantor:/u01/a1# pg_test_fsync
5 seconds per test
O_DIRECT supported on this platform for open_datasync and open_sync.

Compare file sync methods using one 8kB write:
(in wal_sync_method preference order, except fdatasync is Linux's default)
        open_datasync                   561.778 ops/sec    1780 usecs/op
        fdatasync                       567.933 ops/sec     1761 usecs/op
        fsync                           323.302 ops/sec    3093 usecs/op
        fsync_writethrough                        n/a
        open_sync                       343.145 ops/sec    2914 usecs/op

Compare file sync methods using two 8kB writes:
(in wal_sync_method preference order, except fdatasync is Linux's default)
        open_datasync                   284.528 ops/sec    3515 usecs/op
        fdatasync                       505.311 ops/sec     1979 usecs/op
        fsync                           311.811 ops/sec    3207 usecs/op
        fsync_writethrough                        n/a
        open_sync                       163.685 ops/sec    6109 usecs/op

Compare open_sync with different write sizes:
(This is designed to compare the cost of writing 16kB in different write
open_sync sizes.)
         1 * 16kB open_sync write       301.329 ops/sec    3319 usecs/op
         2 *  8kB open_sync writes      166.580 ops/sec    6003 usecs/op
         4 *  4kB open_sync writes       85.399 ops/sec   11710 usecs/op
         8 *  2kB open_sync writes       44.381 ops/sec   22532 usecs/op
        16 *  1kB open_sync writes       22.332 ops/sec   44779 usecs/op

Test if fsync on non-write file descriptor is honored:
(If the times are similar, fsync() can sync data written on a different
descriptor.)
        write, fsync, close             321.995 ops/sec    3106 usecs/op
        write, close, fsync             344.132 ops/sec    2906 usecs/op

Non-sync'ed 8kB writes:
        write                        348190.711 ops/sec       3 usecs/op
```

Для fdatasync значения такие же, как с журналом.

Часть 5. Снятие ограничения на число открытых файлов

1) По умолчанию, максимальное число соединений 100. Установите максимальное число соединений в 10000:

```
postgres@tantor:~$ psql -c "alter system set max_connections=1100;"
ALTER SYSTEM
```

2) Попробуйте выполнить тест pgbench, указав 1100 соединений:

```
postgres@tantor:~$ pgbench -c 1100 -T 10 -P 5
pgbench: error: need at least 1103 open files, but system limit is 1024
pgbench: hint: Reduce number of clients, or use limit/ulimit to increase the
system limit.
```

Выдана ошибка о том, что утилита pgbench хочет открыть 1103 файла, а операционная система ограничивает процессы пользователя postgres 1024 файлами.

3) Проверьте реальные ограничения уже запущенных процессов с именем postgres:

```
postgres@tantor:~$ for PID in $(pgrep "postgres"); do cat /proc/$PID/limits | grep files; done | uniq
Max open files   1024         524288 files
```

Мягкое ограничение 1024 файла, жесткое 524288 файлов.

4) Посмотрите ограничение для новых процессов для пользователей postgres, root, astra:

```
postgres@tantor:~$ sudo -u postgres bash -c 'ulimit -n'
1024
postgres@tantor:~$ sudo -u root bash -c 'ulimit -n'
1024
postgres@tantor:~$ sudo -u astra bash -c 'ulimit -n'
1024
```

У процессов всех пользователей ограничение 1024. Такое ограничение подходит для десктопного использования, но не для сервера. Хотя это мягкое ограничение, но приложения и утилиты могут не обрабатывать предупреждения и отказываться работать, как утилита pgbench.

Влияет ли это ограничение на процессы экземпляра? По умолчанию параметр конфигурации `max_files_per_process=1000`, поэтому не влияет. Однако, перед увеличением значения `max_files_per_process` нужно снять ограничения на уровне операционной системы.

5) Чтобы поменять лимиты для экземпляров, запускаемых вручную утилитой pg_ctl, в файле /etc/security/limits.conf нужно добавить или поменять строки:

```
* hard nofile infinity
root hard nofile infinity
* soft nofile infinity
root soft nofile infinity

postgres@tantor:~$ sudo mcedit /etc/security/limits.conf
```

6) Проверьте, что изменения подействовали:

```
postgres@tantor:~$ sudo -u postgres bash -c 'ulimit -n'
1048576
postgres@tantor:~$ sudo -u astra bash -c 'ulimit -n'
1048576
postgres@tantor:~$ sudo -u root bash -c 'ulimit -n'
1048576
postgres@tantor:~$ sudo -u postgres bash -c 'ulimit -Ht'
unlimited
postgres@tantor:~$ sudo -u astra bash -c 'ulimit -Ht'
unlimited
postgres@tantor:~$ sudo -u root bash -c 'ulimit -Ht'
unlimited
```

7) На запускаемые через `systemd` экземпляры это не подействует.
Отредактируйте файл `/usr/lib/systemd/system/tantor-se-server-16.service`, добавив после `[Service]`

```
LimitNOFILE=infinity
LimitNOFILESoft=infinity
```

```
postgres@tantor:~$ sudo mcedit /usr/lib/systemd/system/tantor-se-server-16.service
```

8) Обновите конфигурацию systemd и перезапустите экземпляр:

```
postgres@tantor:~$ sudo systemctl daemon-reload
postgres@tantor:~$ sudo systemctl restart tantor-se-server-16
```

9) Проверьте реальные ограничения уже запущенных процессов с именем postgres:

```
postgres@tantor:~$ for PID in $(pgrep "postgres"); do cat /proc/$PID/limits | grep files; done | uniq
Max open files      1024          524288         files
Max open files   1048576         1048576         files
```

Есть процессы с ограничением 1024 - это экземпляр Astralinux PostgreSQL, его файл службы не редактировали. У процессов экземпляра tantor ограничения поменялись.

10) Запустите тест pgbench, указав 1100 соединений:

```
postgres@tantor:~$ pgbench -c 1100 -T 10 -P 5
starting vacuum...end.
progress: 7.9 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 10.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 15.0 s, 55.4 tps, lat 4859.114 ms stddev 1268.481, 0 failed
progress: 20.0 s, 67.8 tps, lat 9833.140 ms stddev 1435.181, 0 failed
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1100
number of threads: 1
maximum number of tries: 1
duration: 10 s
number of transactions actually processed: 1100
number of failed transactions: 0 (0.000%)
latency average = 10740.735 ms
latency stddev = 4219.216 ms
initial connection time = 7798.795 ms
tps = 65.627380 (without initial connection time)
```

Тест выполнился.

Часть 6. Пример создания plpgsql кода для тестирования

1) Создайте файл `init.sql`:

```
postgres@tantor:~$ mcedit init.sql
```

```
create schema test;
select format('create table test.film_summary%s (film_id int, title varchar, release_year smallint) with (autovacuum_enabled=off);', g.id) from generate_series(1, 10000) as g(id)
\gexec
```

Команда создаёт 10000 пустых таблиц, 10000 TOAST таблиц и 10000 TOAST-индексов. Автовакуум отключён, чтобы при внесении изменений в строки таблиц внезапно не срабатывал и тем самым не создавал колебания метрик в результатах тестов. Схема используется для удобства, чтобы таблицы не были видны в пути поиска и чтобы можно было их удалять путем удаления схемы.

2) Выполните скрипт:

```
postgres@tantor:~$ time psql -f init.sql > /dev/null
```

```
real    2m29.225s
user    0m1.687s
sys     0m0.999s
```

Скрипт за 2,5 минуты создал схему test и 10000 таблиц в схеме.

3) Попробуйте удалить таблицы и схему:

```
postgres=# \timing
Timing is on.
postgres=# DO
$$
begin
 for i in 1..10000 loop
   execute concat('drop table if exists test.film_summary',i);
 end loop;
 execute 'drop schema if exists test cascade';
end;
$$
LANGUAGE plpgsql;

ERROR:  out of shared memory
HINT:  You might need to increase max_locks_per_transaction.
CONTEXT:  SQL statement "drop table if exists test.film_summary2571"
PL/pgSQL function inline_code_block line 4 at EXECUTE
Time: 936.719 ms
```

Использование plpgsql привело к проблеме превышения числа блокировок: **команды выполнялись одной транзакцией**. Именно поэтому таблицы создавались скриптом, а не блоком plpgsql.

Через 0,9 секунд на удалении 2571 таблицы возникло превышение числа блокировок. Было установлено 2571 блокировок. Число блокировок зависит от параметров `max_locks_per_transaction*max_connextions=64*100=6400`. Как соотносятся числа 2571 и 6400? Для каждой таблицы установлено 3 блокировки: сама таблица, её TOAST-таблица и TOAST индекс. 2571*3=7713 что больше 6400. Больше из-за того, что память под структуру блокировки и любые структуры памяти выделяется с округлением.

4) Добавьте команду `commit` и выполните блок:

```
postgres=# DO
$$
begin
 for i in 1..10000 loop
   execute concat('drop table if exists test.film_summary',i);
commit;
 end loop;
 execute 'drop schema if exists test cascade';
end;
$$
LANGUAGE plpgsql;

DO
Time: 46568.714 ms (00:46.569)
```

Блок успешно выполнится и удалил 30000 объектов за 46 секунд. Это долго.

5) Создайте таблицы заново:

```
postgres@tantor:~$ time psql -f init.sql > /dev/null

real    3m12.168s
user    0m0.404s
sys     0m0.223s
```

6) Создайте более эффективный plpgsql код с промежуточной фиксацией транзакций:

```
postgres=# \timing on \\
DO
$$
begin
 for i in 1..10000 by 100 loop
  for j in 0..99 loop
   execute concat('drop table if exists test.film_summary',i+j);
  end loop;
  commit;
 end loop;
 execute 'drop schema if exists test cascade';
end;
$$
LANGUAGE plpgsql;

DO
Time: 6597.938 ms (00:06.598)
```

то время выполнения будет 6,5 секунд. Промежуточная фиксация транзакций удерживает горизонт немного дольше, но создает меньше нагрузку на легковесные блокировки структур WAL и выполняется быстрее.

Это был пример того, что при написании скриптов тестирования важно обращать внимание на то, какие транзакции явно или неявно формируются для выполнения команд.

Если увеличить число `max_connections` до 1000, перезапустить экземпляр, создать таблицы, и выполнить блок который без `commit`, то таблицы удалятся за 6 секунд:

```
Time: 6030.153 ms (00:06.030)
```

Проверять это не нужно, чтобы не тратить время (по 3 минуты на создание таблиц).

Если увеличить число `max_connections` до 1000, перезапустить экземпляр, создать таблицы и удалить командой:

```
postgres=# \timing
Timing is on.
postgres=# drop schema test cascade;
NOTICE:  drop cascades to 10000 other objects
DETAIL:  drop cascades to table test.film_summary1
drop cascades to table test.film_summary2
drop cascades to table test.film_summary3
...
drop cascades to table test.film_summary99
drop cascades to table test.film_summary100
and 9900 other objects (see server log for list)
DROP SCHEMA
Time: 7526.961 ms (00:07.527)
```

то время удаления будет 7,5 секунд.

7) Файл `init.sql` этой части практики полезен и может использоваться для создания большого числа объектов в целях тестирования.
Если нужно наполнить таблицы строками, можно использовать команду:

```
select format('insert into test.film_summary%1$s select i, %2$s || i, i from
generate_series(0, 153) as i;', g.id, E'\'text number \'') from
generate_series(1, 10000) as g(id)
\gexec
```

Команда создаёт 154 строки в каждой таблице в одном блоке, почти полностью заполняя блок. Большинство блоков в реальных таблицах почти полностью заполнены и таблицы будут этому соответствовать. Максимальное число строк в блоке 156, место под пару строк нужно, чтобы при обновлении строки в блоке сработал HOT cleanup.
Скрипт с этой командой выполнится за 46 секунд.

Если нужно прочесть блоки, чтобы обновить признак фиксации вставки, то можно выполнить команду:

```
select format('select * from test.film_summary%s where film_id = 0;', g.id) from
generate_series(1, 10000) as g(id)
\gexec
```

Скрипт с этой командой выполнится за 3,5 секунды.

Также можно использовать команду, которая собирает статистику и замораживает строки, чтобы они в будущем внезапно не грязнились:

```
select format('vacuum (freeze, analyze) test.film_summary%s;', g.id) from
generate_series(1, 10000) as g(id)
\gexec
```

Однако, скрипт с этой командой будет выполнятся 1 минуту 45 секунды, что долго.

8) Таблицы можно использовать для нагрузочного тестирования.

Для этого создать функцию:

```
create or replace function inittest1(clientid anyelement) returns void
as $$
begin
  for i in 1..15 loop
   execute concat('update test.film_summary', i ,' set title=title ', 'where
film_id = ', clientid);
  end loop;
end;
$$
LANGUAGE plpgsql;
```

Функция обновляет в 15 таблицах (16 - максимальное число fastpath блокировок) строку со значением, передаваемым в параметре функции.

Скрипт теста для pgbench:

```
echo "select inittest1( :client_id);" > inittest1.sql
```

`client_id` - номер сессии pgbench, устанавливается в параметре, который доступен в скриптах тестов.

Запуск теста:

```
pgbench -n -c 36 -T 10 -P 3 -f inittest1.sql
progress: 3.0 s, 1358.0 tps, lat 25.905 ms stddev 10.597, 0 failed
progress: 6.0 s, 1203.0 tps, lat 29.948 ms stddev 10.698, 0 failed
progress: 9.0 s, 1141.7 tps, lat 31.510 ms stddev 11.303, 0 failed
```

```
number of transactions actually processed: 12406
latency average = 28.895 ms
latency stddev = 11.061 ms
tps = 1242.842959 (without initial connection time)
```

Оптимальное число сессий на 4 ядрах при настройках по умолчанию начиная с ~36 сессий. При числе сессий больше 1 HOT cleanup не сможет часто работать и число блоков в таблицах увеличится.По мере увеличения числа блоков tps уменьшается.

Часть 7. Пример создания программ для тестирования

1) Рассмотренные утилиты pgbench, pg_test_fsync, fio отлично подходят для тестирования. Для тестирования могут использоваться простые тесты, написанные на языках программирования c, java, python. Пример самописного теста и задачи, которую он решает https://www.percona.com/blog/fsync-performance-storage-devices/
Создайте файл `fsync.py`:

```
postgres@tantor:~$ mcedit fsync.py
```

```
#!/usr/bin/python
import os, sys, mmap
# Open a file
#fd = os.open( "testfile", os.O_RDWR|os.O_CREAT|os.O_DSYNC )
fd = os.open( "testfile", os.O_RDWR|os.O_CREAT )
fd2 = os.open( "testfile2", os.O_RDWR|os.O_CREAT )
m = mmap.mmap(-1, 8192)
for i in range (1,5000):
   os.lseek(fd,os.SEEK_SET,0)
   m[1] = 1
   os.write(fd, m)
   os.lseek(fd2,os.SEEK_SET,0)
   os.write(fd2, m)
   os.fsync(fd)
   os.fsync(fd2)
# Close opened file
os.close( fd )
os.close( fd2 )
```

2) Запустите тест:

```
postgres@tantor:~$ time python3 ./fsync.py
```

```
real    0m5.379s
user    0m0.276s
sys     0m0.727s
```

Можно менять параметры открытия файла в функции `os.open` и проверять время выполнения программы.

3) Создайте файл `gen.py`:

```
postgres@tantor:~$ mcedit gen.py
```

```
def main():
    for count in [5000, 10000, 25000]:
        with open(f'workload1c_{count}.sql', 'w') as f:
            f.write('BEGIN;\n')
```

```
            for i in range(1, count):
                f.write(f"""create temporary table ttt{i} (
    _C_1RRef bytea,
    _C_2RRef bytea,
    _C_3RRef bytea,
    _C_4 varchar(150),
    _C_5 numeric(9,0),
    _C_6RRef bytea,
    _C_7RRef bytea,
    _C_8RRef bytea,
    _C_9RRef bytea,
    _C_10 boolean,
    _C_11 numeric(5,0),
    _C_12 numeric(10,0),
    _C_13 varchar(430),
    _C_14 numeric(5,0),
    _C_15RRef bytea,
    _C_16RRef bytea,
    _C_17RRef bytea,
    _C_18RRef bytea,
    _C_19_TYPE bytea,
    _C_19_S varchar(150),
    _C_19_RTRef bytea,
    _C_19_RRRef bytea,
    _C_20 varchar,
    _C_21 varchar,
    _C_22 numeric(9,0),
    _TTC_1 bytea,
    _TTC_2 numeric(9,0),
    SDBL_IDENTITY int
);
""")
            f.write('ROLLBACK;\n')

if __name__ == "__main__":
    main()
```

4) Выполните команды в терминале `postgres@tantor:~$` :

```
python3 ./gen.py
psql -c "vacuum full pg_class;"
psql -c "select pg_table_size('pg_class');"
time psql -f workload1c_5000.sql > /dev/null
psql -c "select pg_table_size('pg_class');"
time psql -f workload1c_10000.sql > /dev/null
psql -c "select pg_table_size('pg_class');"
time psql -f workload1c_25000.sql > /dev/null
psql -c "select pg_table_size('pg_class');"
```

Команды выдадут результат:

```
VACUUM
 pg_table_size
---------------
         98304
(1 row)


real    0m10.947s
user    0m0.920s
```

```
sys     0m0.137s
 pg_table_size
---------------
       3932160
(1 row)


real    0m30.293s
user    0m2.695s
sys     0m0.384s
 pg_table_size
---------------
      10960896
(1 row)


real    1m27.637s
user    0m8.314s
sys     0m2.146s
 pg_table_size
---------------
      22822912
(1 row)
```

После вакуумирования таблица системного каталога занимает 98304 байта.
Программа gen.py создала три командных файла: `workload1c_25000.sql workload1c_10000.sql workload1c_5000.sql`. В этих скриптах в одной транзакции создаётся 5000, 10000, 25000 временных таблиц с 25 столбцами, после чего транзакция откатывается.
После запуска этих скриптов измеряется размер таблицы pg_class.
Для 25000 объектов размер таблицы pg_class становится 22Мб.

5) Выполните команду:

```
postgres@tantor:~$ psql -c "select pg_table_size('pg_class');"
 pg_table_size
---------------
        131072
```

Размер таблицы `pg_class` уменьшился до 131Кб. В результате чего уменьшилась таблица? В результате автовакуума. Если бы после создания таблиц в другой сессии были созданы постоянные таблицы, смогла ли таблица `pg_class` уменьшиться в размере? Вряд ли.

Практика к главе 5

Часть 1. Блокировки объектов

1) Число блокировок объектов и рекомендательных блокировок (advisory locks, которые автоматически не используются, их использование требует программирования) на экземпляре определяется произведением `max_locks_per_transaction * max_connections` (предполагается, что `max_prepared_transactions=0` и менять его не нужно).
По умолчанию параметр **`max_locks_per_transaction=64, a max_connections=100:`**

```
postgres=# show max_locks_per_transaction;
 max_locks_per_transaction
---------------------------
 64
(1 row)
postgres=# show max_connections;
 max_connections
-----------------
 100
(1 row)
```

2) Чтобы опечатки в командах внутри транзакции не переводили транзакцию в состояние сбоя, установите параметр:

```
postgres=# \set ON_ERROR_ROLLBACK INTERACTIVE
```

По умолчанию любая опечатка в команде переводит транзакцию в состояние сбоя и в транзакции нельзя продолжить работать, только откатить. Команда **`ON_ERROR_ROLLBACK INTERACTIVE`** указывает неявно устанавливать после каждой команды в транзакции точку сохранения и откатываться к последней точке сохранения, если команда в транзакции не смогла выполниться (достаточно опечатки). Опция INTERACTIVE устанавливает это правило для интерактивного ввода команд и не устанавливает точки сохранения при выполнении скриптов. Поэтому эта настройка и удобна и безопасна.

3) Посмотрите сколько памяти выделено серверному процессу:

```
postgres=# select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
   sum    |   sum    |   sum
----------+----------+--------
 1329320 | 947688 | 381632
(1 row)
```

1-2Мб это не так много. Если выдаётся больше 2Мб, переподсоединитесь.

4) Включите вывод времени выполнения команд, чтобы знать длительность выполнения:

```
postgres=# \timing on
Timing is on.
```

Измерение времени удобно для того, чтобы представлять себе длительность выполнения команд.

5) Создайте секционированную таблицу:

```
postgres=# create table parttab(n numeric,k numeric, v varchar(100)) partition by
range (n);
CREATE TABLE
```

```
Time: 0.976 ms
```

6) Начните транзакцию:

```
postgres=# begin transaction;
BEGIN
Time: 0.150 ms
```

7) Выполните анонимный plpgsql блок:

```
postgres=*# do
$$
declare
cnt integer;
v varchar(200);
begin
for i in 0..4000 loop
v:= concat('create table parttab_',i,' partition of parttab for values from
(',i*10,') to (',(i+1)*10,')');
execute v;
end loop;
end;
$$
;

DO
Time: 67764.380 ms (01:07.764)
```

Блок выполнялся 67 секунд.

8) Посмотрите сколько памяти выделено серверному процессу:

```
postgres=*# select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
   sum    |   sum    |   sum
----------+----------+---------
 57673256 | 47280560 | 10392696
(1 row)
```

Выделено 57Мб, что довольно много.

9) Посмотрите содержимое таблицы блокировок:

```
postgres=*# select locktype, relation, relation::regclass name, mode, granted,
fastpath from pg_locks order by relation limit 9;
 locktype | relation |               name                |        mode         | granted
----------+----------+-----------------------------------+---------------------+---------
 relation |    12073 | pg_locks                          | AccessShareLock     | t
 relation |    12138 | pg_backend_memory_contexts        | AccessShareLock     | t
 relation |  2943194 | parttab                           | AccessExclusiveLock | t
 relation |  2943197 | parttab_0                         | AccessExclusiveLock | t
 relation |  2943200 | pg_toast.pg_toast_2943197         | ShareLock           | t
 relation |  2943201 | pg_toast.pg_toast_2943197_index   | AccessExclusiveLock | t
 relation |  2943202 | parttab_1                         | AccessExclusiveLock | t
 relation |  2943205 | pg_toast.pg_toast_2943202         | ShareLock           | t
 relation |  2943206 | pg_toast.pg_toast_2943202_index   | AccessExclusiveLock | t
(9 rows)

Time: 27.150 ms
```

Установлены блокировки типов `AccessShare`, `AccessExclusive`, `Share`.

10) Посмотрите число блокировок:

```
postgres=# select count(*) from pg_locks;
select count(distinct relation) from pg_locks;
 count
-------
 12009
(1 row)

Time: 11.052 ms
 count
-------
 12006
(1 row)
```

Установлено по одной блокировке на каждый relation.

```
postgres=# select locktype, relation, mode, granted, fastpath from pg_locks where
fastpath=true;
   locktype    | relation |      mode       | granted | fastpath
---------------+----------+-----------------+---------+----------
 virtualxid    |          | ExclusiveLock   | t       | t
(1 row)
```

Всего одна блокировка получена по быстрому пути (fastpath). Блокировки, полученные по быстрому пути не занимают место в общей структуре блокировок.

```
postgres=# select mode, count(mode) from pg_locks group by mode;
        mode         | count
---------------------+-------
 ShareLock           |  4001
 AccessExclusiveLock |  8003
 ExclusiveLock       |     2
 AccessShareLock     |     3
(4 rows)
```

Число блокировок в общей структуре блокировок `12008` и существенно превышает `max_locks_per_transaction * max_connections = 64*100 = 6400`. Память выделяется по степеням двойки. Также блокировки fastpath не входят в ограничение.

11) Откатите транзакцию:

```
postgres=*# rollback;
ROLLBACK
Time: 588.883 ms
```

12) Проверьте, что блокировки были сняты:

```
postgres=# select locktype, relation::regclass relation, virtualxid,
virtualtransaction, mode, fastpath from pg_locks;
  locktype  | relation | virtualxid | virtualtransaction |      mode       | fastpath
------------+----------+------------+--------------------+-----------------+----------
 relation   | pg_locks |            | 3/11               | AccessShareLock | t
 virtualxid |          | 3/11       | 3/11               | ExclusiveLock   | t
(2 rows)
```

Часть 2. Наблюдение за памятью серверного процесса

1) Начните транзакцию:

```
postgres=# begin transaction;
BEGIN
Time: 0.150 ms

postgres=*# do
$$
declare
cnt integer;
v varchar(200);
begin
for i in 0..4291 loop
v:= concat('create table parttab_',i,' partition of parttab for values from
(',i*10,') to (',(i+1)*10,')');
execute v;
end loop;
end;
$$
;

ERROR:  out of shared memory
HINT:  You might need to increase max_locks_per_transaction.
CONTEXT:  SQL statement "create table parttab_4290 partition of parttab for
values from (42900) to (42910)"
PL/pgSQL function inline_code_block line 8 at EXECUTE
Time: 72606.127 ms (01:12.606)
```

Анонимный блок не смог добавить 4290 секцию а таблицу parttab из-за того, что в общей структуре памяти для блокировок не хватило места.

2) Откатите транзакцию:

```
postgres=!# rollback;
ROLLBACK
Time: 588.883 ms
```

Откат транзакции освобождает контексты локальной памяти процесса, выделенные в контексте транзакции.

3)
```
postgres=# \d parttab
                Partitioned table "public.parttab"
 Column |          Type          | Collation | Nullable | Default
--------+------------------------+-----------+----------+---------
 n      | numeric                |           |          |
 k      | numeric                |           |          |
 v      | character varying(100) |           |          |
Partition key: RANGE (n)
Number of partitions: 0
```

Секции не были добавлены.

4) Начните транзакцию:

```
postgres=# begin transaction;
BEGIN
Time: 0.150 ms
```

5) Уменьшите число секций до 2600 и выполните блок:

```
postgres=*# do
```

```
$$
declare
cnt integer;
v varchar(200);
begin
for i in 0..2600 loop
v:= concat('create table parttab_',i,' partition of parttab for values from
(',i*10,') to (',(i+1)*10,')');
execute v;
end loop;
end;
$$
;

CREATE TABLE
DO
Time: 35644.365 ms (00:35.644)

postgres=# \d parttab
                Partitioned table "public.parttab"
 Column |          Type          | Collation | Nullable | Default
--------+------------------------+-----------+----------+---------
 n      | numeric                |           |          |
 k      | numeric                |           |          |
 v      | character varying(100) |           |          |
Partition key: RANGE (n)
Number of partitions: 2601 (Use \d+ to list them.)
```

Была создана `2601` секция.

6) Попробуйте удалить таблицу:

```
postgres=*# drop table if exists parttab;
ERROR:  out of shared memory
HINT:  You might need to increase max_locks_per_transaction.
Time: 222.310 ms
```

Удалить таблицу нельзя - команда пытается установить много блокировок. Если бы транзакция была зафиксирована, то пришлось бы увеличивать параметр `max_connections` или `max_locks_per_transaction`. Увеличение любого из этих параметров требует предварительно увеличения на репликах и рестарта экземпляра.

7) При этом можно работать с таблицей. Выполните команды:

```
postgres=*# insert into parttab select generate_series(1, 10000-1),10000-
generate_series(1, 10000-1),'Initial value '||generate_series(1,
10000-1);
INSERT 0 9999
Time: 119.122 ms
```

8) Посмотрите сколько памяти выделено процессу:

```
postgres=*# select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
   sum    |   sum    |   sum
----------+----------+---------
 57576184 | 36993936 | 20582248
(1 row)
```

Размер памяти, выделенной процессу увеличился незначительно - менее, чем на 1Мб с 57**6**73256 до 57**5**76184.

9) Выполните команды:

```
postgres=*# prepare prepstmt1 (int) as select v from parttab where n=$1;
PREPARE
Time: 0.700 ms

postgres=*# execute prepstmt1(1);
        v
-----------------
 Initial value 1
(1 row)

Time: 0.441 ms
```

10) Посмотрите сколько памяти выделено процессу:

```
postgres=*# select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
   sum    |   sum    |   sum
----------+----------+---------
 57598712 | 37014832 | 20583880
(1 row)

Time: 2.493 ms
```

Размер памяти процесса увеличился незначительно.

11) Создайте индекс на таблицу и проверьте как поменялся объем выделенной процессу памяти:

```
postgres=*# create unique index parttabn on parttab(n);
 select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
CREATE INDEX
Time: 28621.034 ms (00:28.621)
   sum    |   sum    |   sum
----------+----------+----------
 61139752 | 47002536 | 14137216
(1 row)

Time: 5.722 ms
```

Размер памяти процесса увеличился на ~4Мб с 57598712 до 61139752.

12) Выполните команду, которая нужна для того, чтобы планировщик считал, что строк в таблице больше чем есть:

```
update pg_class set reltuples=reltuples*100 where relname like 'parttab%';
UPDATE 5204
Time: 245.317 ms
```

13) Дальше выполните 6 раз подготовленную команду с замером выделенной процессу памяти:

```
postgres=*# select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
execute prepstmt1(1000);
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
execute prepstmt1(5000);
```

```
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
execute prepstmt1(10000);
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
execute prepstmt1(100);
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
execute prepstmt1(300);
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
execute prepstmt1(2000);
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
   sum    |   sum    |   sum
----------+----------+----------
 61314488 | 47147896 | 14166592
(1 row)

Time: 5.532 ms
         v
--------------------
 Initial value 1000
(1 row)

Time: 28.541 ms
   sum    |   sum    |   sum
----------+----------+----------
 61470504 | 49987992 | 11482512
(1 row)

Time: 5.344 ms
         v
--------------------
 Initial value 5000
(1 row)

Time: 0.433 ms
   sum    |   sum    |   sum
----------+----------+----------
 61471528 | 49988816 | 11482712
(1 row)

Time: 5.606 ms
 v
---
(0 rows)

Time: 0.370 ms
   sum    |   sum    |   sum
----------+----------+----------
 61471528 | 49989528 | 11482000
(1 row)

Time: 5.772 ms
         v
-------------------
 Initial value 100
(1 row)

Time: 0.388 ms
   sum    |   sum    |   sum
----------+----------+----------
 61472552 | 49990352 | 11482200
(1 row)

Time: 5.190 ms
         v
-------------------
 Initial value 300
(1 row)

Time: 208.855 ms
   sum    |   sum    |   sum
----------+----------+----------
 72232560 | 58599344 | 13633216
(1 row)

Time: 5.309 ms
```

```
         v
--------------------
 Initial value 2000
(1 row)

Time: 0.315 ms
   sum    |   sum    |   sum
----------+----------+----------
 72232560 | 58599344 | 13633216
(1 row)

Time: 5.009 ms
```

На шестом (после подготовки) выполнении подготовленного запроса серверный процесс переключился на общий (generic) план выполнения. Размер плана 10Мб, что больше, чем любой из частных планов, так как включает в себя все секции таблицы. Размер памяти процесса увеличился с 61472552 до **72232560** байт.

14) Посмотрите какие контексты памяти есть и их размер:

```
postgres=*# with recursive dep as (select name, total_bytes as total, left(ident,10)
ident, 1 as level, left(name,38) as path from pg_backend_memory_contexts where parent is
null
union all
select c.name, c.total_bytes, left(c.ident,10), p.level+1, left(p.path||'->'||c.name,38)
from dep p, pg_backend_memory_contexts c
where c.parent = p.name)
select * from dep limit 40;
          name           |  total   |   ident    | level |                  path
-------------------------+----------+------------+-------+----------------------------------------
 TopMemoryContext        |  4259200 |            |     1 | TopMemoryContext
 Btree proof lookup cache |    8192 |            |     2 | TopMemoryContext->Btree proof lookup c
 Prepared Queries        |    16384 |            |     2 | TopMemoryContext->Prepared Queries
 TopTransactionContext   |  6394000 |            |     2 | TopMemoryContext->TopTransactionContex
 TableSpace cache        |     8192 |            |     2 | TopMemoryContext->TableSpace cache
 RegexpCacheMemoryContext |    1024 |            |     2 | TopMemoryContext->RegexpCacheMemoryCon
 Type information cache  |    24368 |            |     2 | TopMemoryContext->Type information cac
 Operator lookup cache   |    24576 |            |     2 | TopMemoryContext->Operator lookup cach
 PLpgSQL cast expressions |    8192 |            |     2 | TopMemoryContext->PLpgSQL cast express
 CFuncHash               |     8192 |            |     2 | TopMemoryContext->CFuncHash
 Rendezvous variable hash |    8192 |            |     2 | TopMemoryContext->Rendezvous variable
 PLpgSQL function hash   |     8192 |            |     2 | TopMemoryContext->PLpgSQL function has
 Record information cache |    8192 |            |     2 | TopMemoryContext->Record information c
 RowDescriptionContext   |     8192 |            |     2 | TopMemoryContext->RowDescriptionContex
 MessageContext          |   131072 |            |     2 | TopMemoryContext->MessageContext
 Operator class cache    |     8192 |            |     2 | TopMemoryContext->Operator class cache
 PgStat Shared Ref Hash  |   394288 |            |     2 | TopMemoryContext->PgStat Shared Ref Ha
 PgStat Shared Ref       |   573440 |            |     2 | TopMemoryContext->PgStat Shared Ref
 PgStat Pending          |  2146304 |            |     2 | TopMemoryContext->PgStat Pending
 smgr relation table     |  2097152 |            |     2 | TopMemoryContext->smgr relation table
 TransactionAbortContext |    32768 |            |     2 | TopMemoryContext->TransactionAbortCont
 Portal hash             |     8192 |            |     2 | TopMemoryContext->Portal hash
 TopPortalContext        |     8192 |            |     2 | TopMemoryContext->TopPortalContext
 Relcache by OID         |  1048576 |            |     2 | TopMemoryContext->Relcache by OID
 CacheMemoryContext      | 34046240 |            |     2 | TopMemoryContext->CacheMemoryContext
 WAL record construction |    50200 |            |     2 | TopMemoryContext->WAL record construct
 PrivateRefCount         |     8192 |            |     2 | TopMemoryContext->PrivateRefCount
 MdSmgr                  |   131072 |            |     2 | TopMemoryContext->MdSmgr
 LOCALLOCK hash          |  2097152 |            |     2 | TopMemoryContext->LOCALLOCK hash
 GUCMemoryContext        |    24576 |            |     2 | TopMemoryContext->GUCMemoryContext
 Timezones               |   104112 |            |     2 | TopMemoryContext->Timezones
 ErrorContext            |     8192 |            |     2 | TopMemoryContext->ErrorContext
 CurTransactionContext   |     8192 |            |     3 | TopMemoryContext->TopTransactionContex
 Combo CIDs              |   524288 |            |     3 | TopMemoryContext->TopTransactionContex
 RegexpMemoryContext     |    13360 | ^(parttab) |     3 | TopMemoryContext->RegexpCacheMemoryCon
 PortalContext           |     1024 | <unnamed>  |     3 | TopMemoryContext->TopPortalContext->Po
 CachedPlan              |  8551240 | prepare pr |     3 | TopMemoryContext->CacheMemoryContext->
 partition descriptor    |   153968 | parttab    |     3 | TopMemoryContext->CacheMemoryContext->
 partition key           |     1024 | parttab    |     3 | TopMemoryContext->CacheMemoryContext->
 index info              |     1024 | parttab_26 |     3 | TopMemoryContext->CacheMemoryContext->
(40 rows)
```

15) Во другом терминале запустите psql:

```
astra@tantor:~$ psql
psql
Type "help" for help.

postgres=#
```

16) В третьем терминале посмотрите сколько памяти занято в операционной системе:

```
postgres@tantor:~$ ps -e -o vsz,rss,pss,psr,rops,wops,pid,cmd --sort -vsz | grep
postgres:
336508 191924 143715 3  2662 10224  101373 postgres: postgres postgres [local] idle in
transaction
235100 94116 51708   3    24 22597  101196 postgres: checkpointer
230112 16804  3926   3    54     1  102783 postgres: postgres postgres [local] idle
229580 11812  2216   0   204     1  101200 postgres: autovacuum launcher
227480 10660  1879   1    10     1  101201 postgres: logical replication launcher
226296 14628  2800   0    29     1  101197 postgres: background writer
225908 31104 11337   0 28807 45318  101195 /opt/tantor/db/16/bin/postgres -D
/var/lib/postgresql/tantor-se-16/data
225908 11940  2232   0    71  1554  101199 postgres: walwriter
```

В операционной системе серверному процессу выделено 336508Кб виртуального адресного пространства. Из них: резидентный размер (вместе с исполняемым кодом)- объем памяти, выделенный процессу и в находящихся в физической памяти RSS=191924Кб, пропорциональный набор (доля исполняемого кода разделяемых библиотек) PSS=144852Кб.

Второму серверному процессу, в котором не выполнялась ни одна команда выделено 230112Кб виртуального адресного пространства. Из них: резидентный размер в физической памяти RSS=16804Кб, пропорциональный размер PSS=3926Кб.

В представлении **pg_backend_memory_contexts** указано, что процесс использует 72232560 байт.

17) Откатите транзакцию открытую в первом терминале:

```
postgres=*# rollback;
ROLLBACK
Time: 488.252 ms
```

18) Удалите таблицу:

```
postgres=# drop table if exists parttab;
DROP TABLE
```

19) Посмотрите, освободилась ли память:

```
postgres=# select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
   sum    |   sum    |   sum
----------+----------+----------
 56052848 | 12087192 | 43965656
(1 row)
```

Часть памяти освободилась и вернулась к размеру измеренному в 8 пункте 1 части практики.

20) Посмотрите сколько памяти выделено по данным операционной системы:

```
postgres@tantor:~$ ps -e -o vsz,rss,pss,psr,rops,wops,pid,cmd --sort -vsz | grep postgres
```

```
332408 189380 140903 2  2664 10224  101373 postgres: postgres postgres [local] idle
235100 94244 51706   3    29 22597  101196 postgres: checkpointer
230112 16804  3922   1    61     1  102783 postgres: postgres postgres [local] idle
229580 11812  2216   0   229     1  101200 postgres: autovacuum launcher
227480 10788  1903   1    12     1  101201 postgres: logical replication launcher
226296 14628  2777   1    30     1  101197 postgres: background writer
225908 31104 11337   0 28807 45318  101195 /opt/tantor/db/16/bin/postgres -D
/var/lib/postgresql/tantor-se-16/data
225908 11940  2231   2    78  1566  101199 postgres: walwriter
```

Оценочно можно сказать, что освободилось примерно 95Мб (332408-235100=97308Кб).

20) Посмотрите сколько памяти свободно и занято:

```
postgres@tantor:~$ free
               total        used        free      shared  buff/cache   available
Mem:         4020796     1245216     1527140      135824     1675284     2775580
Swap:              0           0           0
```

21) Закройтеутилиту psql, в котором создавали секционированную таблицу, чтобы разорвать сессию и серверный процесс завершился и освободил локальную память, которую по его данным он занимает 56052848 и выполните команду **free**:

```
postgres=# exit
postgres@tantor:~$ free
               total        used        free      shared  buff/cache   available
Mem:         4020796     1134344     1638008      135824     1675288     2886452
Swap:              0           0           0
```

Освободилось 110Мб.

22) Посмотрите сколько памяти показывает утилита ps:

```
postgres@tantor:~$ ps -e -o vsz,rss,pss,psr,rops,wops,pid,cmd --sort -vsz | grep postgres
235100 94244 82711   3    29 22897  101196 postgres: checkpointer
230112 16804  4995   1    61     1  102783 postgres: postgres postgres [local] idle
229580 11812  2611   0   233     1  101200 postgres: autovacuum launcher
227480 10788  2146   1    12     1  101201 postgres: logical replication launcher
226296 14628  3345   2    30     1  101197 postgres: background writer
225908 31104 11337   0 28807 45318  101195 /opt/tantor/db/16/bin/postgres -D
/var/lib/postgresql/tantor-se-16/data
225908 11940  2787   0    78  1566  101199 postgres: walwriter
  6980  2304   414   2    13     0  103846 grep postgres:
```

Локальная память остановленного серверного процесса не учитывалась в памяти других процессов. В памяти других процессов не учитывается локальная память, только общая память, к которой имеет доступ процесс. У остановленного серверного процесса показывалась память 332408. Из них локальной памяти по данным операционной системы было 110Мб, по данным процесса 56052848 байт. При выделении памяти linux использует выделение кусками (buddy allocation).

Часть 3. Временные таблицы и файлы

1) Посмотрите какие есть параметры для временных файлов и их значения:

```
postgres=# \dconfig temp*
List of configuration parameters
    Parameter     | Value
------------------+-------
 temp_buffers     | 8MB
 temp_file_limit  | -1
```

```
 temp_tablespaces |
(3 rows)
```

По умолчанию размер локального кэша буферов 8Мб. Ограничений на размер временных файлов нет и табличные пространства для временных файлов не установлены.

2) Создайте временную таблицу:

```
postgres=# create temp table temp1 (id integer);
CREATE TABLE
```

3) Посмотрите путь к первому файлу основного слоя временной таблицы:

```
postgres=# select pg_relation_filepath('temp1');
 pg_relation_filepath
----------------------
 base/5/t3_456929
(1 row)
```

Файл лежит в директории табличного пространства pg_default.

4) Вставьте 10000 строк во временную таблицу temp1:

```
postgres=# insert into temp1 select * from generate_series(1, 10000);
INSERT 0 10000
```

5) Посмотрите размер файлов таблицы temp1:

```
postgres=# \! ls -al $PGDATA/base/5/t*
-rw------- 1 postgres postgres 368640 /var/lib/postgresql/tantor-se-
16/data/base/5/t3_456932
-rw------- 1 postgres postgres  24576 /var/lib/postgresql/tantor-se-
16/data/base/5/t3_456932_fsm
```

Строки размещаются в файле, состоящем из 45 блоков по 8Кб: 368640/8192=45. Карта свободного пространства состоит из 3 блоков.

6) Посмотрите сколько строк находится в блоке временной таблицы:

```
postgres=# select count(*) from temp1 where ctid::text like '(43,%)';
 count
-------
   226
(1 row)
```

`ctid` - это служебный столбец, присутствующий у обычных таблиц (heap table). ctid является адресом строки. Первое число это номер блока, второе число номер строки в блоке. Нумерация блоков начинается с нуля. Во всех блоках кроме последнего находится по 226 строки. В последнем блоке находится 56 строк. 226 - максимальное число непустых строк, которое может поместиться в блок обычной таблицы СУБД Tantor SE.

7) Выполните команду:

```
postgres=# explain (analyze, buffers) select * from temp1;
                            QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on temp1  (cost=0.00..159.75 rows=11475 width=4) (actual
time=0.013..44.184 rows=10000 loops=1)
```

```
   Buffers: local hit=45
 Planning Time: 0.028 ms
 Execution Time: 85.129 ms
(4 rows)
```

Чтение из локального буферного кэша (`hit`), чтение с диска или страничного кэша linux (`read`), запись (`write`) отображается в плане выполнения команды словом **`local`**. Для разделяемого буферного кэша словом `shared`.

45 блоков временной таблицы поместились в локальный буферный кэш, размер которого по умолчанию 8Мб/8192=1024 буфера. Один буфер хранит содержимое одного блока.
Логика работы с локальным буфером аналогична логике работы с разделяемым кэшем буферов. Нет такого, что строки сначала находятся в памяти (temp_buffers), а файлы не создаются.

8) Установите ограничение на максимальный размер временных файлов на уровне сессии:

```
postgres=# set temp_file_limit = '1MB';
SET
```

9) Попробуйте вставить 1млн. строк:

```
postgres=# insert into temp1 select * from generate_series(1, 1000000);
ERROR:  temporary file size exceeds temp_file_limit (1024kB)
```

10) Верните значение к умолчанию (не ограничено):

```
postgres=# reset temp_file_limit;
RESET
```

11) Вставьте 1млн. строк во временную таблицу temp1:

```
postgres=# insert into temp1 select * from generate_series(1, 1000000);
INSERT 0 1000000
```

12) Выполните команду несколько раз:

```
postgres=# explain (analyze, buffers) select * from temp1;
                     QUERY PLAN
-----------------------------------------------------
 Seq Scan on temp1  (cost=0.00..15868.50 rows=1139850 width=4) (actual time=0.044..4123.828
rows=1010000 loops=1)
   Buffers: local read=4470 dirtied=4426 written=4424
 Planning Time: 0.028 ms
 Execution Time: 8117.431 ms
(4 rows)

postgres=# explain (analyze, buffers) select * from temp1;
                     QUERY PLAN
-----------------------------------------------------
 Seq Scan on temp1  (cost=0.00..15868.50 rows=1139850 width=4) (actual time=0.046..4077.274
rows=1010000 loops=1)
   Buffers: local read=4470 written=1024
 Planning Time: 0.028 ms
 Execution Time: 8069.024 ms
(4 rows)

postgres=# explain (analyze, buffers) select * from temp1;
                     QUERY PLAN
-----------------------------------------------------
 Seq Scan on temp1  (cost=0.00..15868.50 rows=1139850 width=4) (actual time=0.022..4077.606
rows=1010000 loops=1)
   Buffers: local read=4470
 Planning Time: 0.027 ms
 Execution Time: 8064.546 ms
```

```
(4 rows)
```

13) Посмотрите размер памяти, используемой серверным процессом:

```
postgres=# select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
   sum    |   sum    |  sum
---------+---------+--------
 9919784 | 9563824 | 355960
(1 row)
```

Размер используемой памяти 9,3Мб.

```
postgres=# with recursive dep as
(select name, total_bytes as total, left(ident,10) ident, 1 as level, left(name,38) as path from
pg_backend_memory_contexts where parent is null
union all
select c.name, c.total_bytes, left(c.ident,10), p.level+1, left(p.path||'->'||c.name,38)
from dep p, pg_backend_memory_contexts c
where c.parent = p.name)
select * from dep order by total desc limit 3;
        name         |  total  | ident | level |                  path
---------------------+---------+-------+-------+---------------------------------------
 LocalBufferContext | 8425808 |       |     2 | TopMemoryContext->LocalBufferContext
 CacheMemoryContext |  524288 |       |     2 | TopMemoryContext->CacheMemoryContext
 MessageContext     |  262144 |       |     2 | TopMemoryContext->MessageContext
(3 rows)
```

Локальный кэш буферов имеет название контекста `LocalBufferContext` занимает 8Мб. Этот контекст занимает большую часть локальной памяти серверного процесса.

Можно в другом терминале посмотреть как выглядит занятая память с точки зрения утилит операционной системы:

```
postgres@tantor:~ $ ps -e -o vsz,rss,pss,psr,rops,wops,pid,cmd --sort -vsz | grep postgres
   VSZ   RSS   PSS PSR  ROPS  WOPS     PID CMD
238624 28316 15916   2 15040 14988  118376 postgres: postgres postgres [local] idle
235100 95140 82302   2    29 25157  101196 postgres: checkpointer
229580 11812  2593   1  1664     1  101200 postgres: autovacuum launcher
227480 10788  2107   1    12     1  101201 postgres: logical replication launcher
226296 14628  3374   2    39     1  101197 postgres: background writer
225908 31104  9787   0 91480 62149  101195 /opt/tantor/db/16/bin/postgres -D
/var/lib/postgresql/tantor-se-16/data
225908 11940  2639   3   106  1607  101199 postgres: walwriter
 79868  6332   937   1   486   538  116927 postgres: 15/main: logger
```

Зная, что серверному процессу выделено 9919784 байт, можно посмотреть какой из показателей может быть полезен, если наблюдать за памятью утилитами операционной системы. Возможно, 238624-225908=12716, возможно PSS=**15916**. Ни один из показателей точно не отражает выделенную процессу локальную память. VSZ (размер виртуальной памяти) показывает ту память, к которой имеет доступ процесс. Почти у всех процессов экземпляра есть доступ к разделяемому пулу размером 128Мб и другим структурам разделяемой памяти общим размером около ~200Мб. Использовать значение столбца VSZ для оценки памяти выделенной процессу неинформативно. Если же из значения столбца вычесть общую память разделяемых структур, то можно получить оценку размера локальной памяти процесса. Какой процесс взять за основу? Возможно postmaster: 225908.
В примере вывода у фонового процесса logger VSZ (VmSize)=79868, а значит в настоящее время процесс не работает с разделяемыми структурами памяти экземпляра. При этом у процесса logger VmPeak=264500.
Можно использовать более детальные показатели использования процессом памяти:

```
postgres@tantor:~$ cat /proc/101195/status | head -33 | tail -16
VmPeak:   225908 kB
VmSize:   225908 kB
VmLck:         0 kB
VmPin:         0 kB
```

```
VmHWM:     31104 kB
VmRSS:     31104 kB
RssAnon:            2560 kB
RssFile:           17408 kB
RssShmem:          11136 kB
VmData:     1748 kB
VmStk:       132 kB
VmExe:      5240 kB
VmLib:     19096 kB
VmPTE:       180 kB
VmSwap:        0 kB
HugetlbPages:          0 kB
```

14) Посмотрите размер файлов таблицы temp1:

```
postgres=# \! ls -al $PGDATA/base/5/t*
-rw------- 1 postgres postgres 36618240 /var/lib/postgresql/tantor-se-16/data/base/5/t3_456983
-rw------- 1 postgres postgres    32768 /var/lib/postgresql/tantor-se-16/data/base/5/t3_456983_fsm
```

15) Закройте утилиту psql, чтобы завершить сессию с базой данных:

```
postgres=# \q
```

16) Посмотрите размер файлов таблицы temp1:

```
postgres@tantor:~$ ls -al $PGDATA/base/5/t*
ls: cannot access '/var/lib/postgresql/tantor-se-16/data/base/5/t*': No such file
or directory
```

Файлы были удалены. Временные файлы удаляются автоматически.

17) Пошлите на выполнение команду, которая породит 5Гб временных файлов в процессе выполнения:

```
postgres=# explain (analyze, buffers)
with recursive t as (
 select 0 i, '' s
union all
 select i + 1, repeat('a', i + 1) from t where i < 1e5 -- 100000
)
table t order by s desc limit 1;
```

18) в другом терминале запустите psql и выполните команду с интервалом в 1-2 минуты:

```
postgres=# select * from pg_ls_tmpdir();
       name        |    size    |      modification
-------------------+------------+------------------------
 pgsql_tmp119544.1 | 1073741824 | 2035-12-12 15:12:59+03
 pgsql_tmp119544.2 |   22331392 | 2035-12-12 15:13:00+03
(2 rows)

postgres=# select * from pg_ls_tmpdir();
       name        |    size    |      modification
-------------------+------------+------------------------
 pgsql_tmp119544.1 | 1073741824 | 2035-12-12 15:12:59+03
 pgsql_tmp119544.2 | 1073741824 | 2035-12-12 15:14:29+03
 pgsql_tmp119544.3 | 1073741824 | 2035-12-12 15:15:08+03
 pgsql_tmp119544.4 | 1073741824 | 2035-12-12 15:16:17+03
 pgsql_tmp119544.5 |  375881728 | 2035-12-12 15:16:45+03
```

```
(5 rows)
```

В директории для временных файлов табличного пространства по умолчанию появляются временные файлы размером до 1Гб.

19) После того как команда завершит свою работу через 300 секунд или будет прервана комбинацией клавиш <ctrl+c> файлы удалятся:

```
                          QUERY PLAN
-----------------------------------------------------------------------------------------------
 Limit  (cost=3.82..3.82 rows=1 width=36) (actual time=296625.350..296625.398 rows=1 loops=1)
   Buffers: temp written=610577
   CTE t
     ->  Recursive Union  (cost=0.00..3.04 rows=31 width=36) (actual time=0.011..66081.120 rows=100001 loops=1)
           ->  Result  (cost=0.00..0.01 rows=1 width=36) (actual time=0.004..0.011 rows=1 loops=1)
           ->  WorkTable Scan on t t_1  (cost=0.00..0.27 rows=3 width=36) (actual time=0.636..0.641 rows=1
loops=100001)
                 Filter: ((i)::numeric < '100000'::numeric)
                 Rows Removed by Filter: 0
   ->  Sort  (cost=0.78..0.85 rows=31 width=36) (actual time=296625.343..296625.355 rows=1 loops=1)
         Sort Key: t.s DESC
         Sort Method: top-N heapsort  Memory: 122kB
         Buffers: temp written=610577
         ->  CTE Scan on t  (cost=0.00..0.62 rows=31 width=36) (actual time=0.018..212744.403 rows=100001
loops=1)
               Buffers: temp written=610577
 Planning Time: 0.061 ms
 Execution Time: 299594.624 ms
(16 rows)
```

Запрос выполнялся 299 секунд. Было записано 610577 блоков = ~5Гб.

```
postgres=# SELECT * FROM pg_ls_tmpdir();
 name | size | modification
------+------+--------------
(0 rows)
```

20) Проверьте, что на временные файлы действует ограничение. Установите ограничение на максимальный размер временных файлов на уровне сессии:

```
postgres=# set temp_file_limit = '1MB';
SET
```

21) Повторите команду:

```
postgres=# explain (analyze, buffers)
with recursive t as (
 select 0 i, '' s
union all
 select i + 1, repeat('a', i + 1) from t where i < 1e5 -- 100000
)
table t order by s desc limit 1;
ERROR:  temporary file size exceeds temp_file_limit (1024kB)
```

Ограничение действует.

Часть 4. Влияние параметров конфигурации на разделяемую память

1) Посмотрите размер структур разделяемой памяти экземпляра с названиями <anonymous> и пустым именем (NULL):

```
postgres=# SELECT name, allocated_size, pg_size_pretty(allocated_size) FROM
pg_shmem_allocations ORDER BY size DESC limit 4;
      name       | allocated_size | pg_size_pretty
```

```
---------------+----------------+----------------
 Buffer Blocks |      134221824 | 128 MB
 <anonymous>   |        4946048 | 4830 kB
 XLOG Ctl      |        4208256 | 4110 kB
               |        1924224 | 1879 kB
(4 rows)
```

2) Выполните запрос, чтобы позже можно было сравнить как поменяются размеры структур разделяемой памяти экземпляра при изменении параметров конфигурации:

```
postgres=# SELECT name, allocated_size, pg_size_pretty(allocated_size) FROM
pg_shmem_allocations  where name like '%LOCK%' or name like '%roc%';
           name           | allocated_size | pg_size_pretty
--------------------------+----------------+----------------
 PROCLOCK hash            |           2944 | 2944 bytes
 Proc Header              |            256 | 256 bytes
 PREDICATELOCKTARGET hash |           2944 | 2944 bytes
 ProcSignal               |          11392 | 11 kB
 Proc Array               |            640 | 640 bytes
 PREDICATELOCK hash       |           2944 | 2944 bytes
 LOCK hash                |           2944 | 2944 bytes
(7 rows)
```

Размеры несерьёзные. На экземпляре может работать `max_connections = 100` серверных процессов. **256 bytes**/100=2,56 байта и даже **640 bytes**/100=6,4 байта вряд ли могут сохранить что-то полезное о процессе.

3) Выполните команду, чтобы посмотреть реальные размеры структур памяти:

```
postgres=# select * from (select *, lead(off) over(order by off) - off as
true_size  from pg_shmem_allocations) as a where name like '%LOCK%' or name like
'%roc%';
           name           |    off    | size  | allocated_size | true_size
--------------------------+-----------+-------+----------------+-----------
 LOCK hash                | 142667776 |  2896 |           2944 |    695168
 PROCLOCK hash            | 143362944 |  2896 |           2944 |    695168
 PREDICATELOCKTARGET hash | 144062336 |  2896 |           2944 |    445312
 PREDICATELOCK hash       | 144507648 |  2896 |           2944 |   1260416
 Proc Header              | 146621568 |   136 |            256 |    152192
 Proc Array               | 146773760 |   544 |            640 |       640
 ProcSignal               | 147177984 | 11272 |          11392 |     11392
(7 rows)
```

4) Проверим как повлияет на размер структур разделяемой памяти увеличение значения параметра `max_locks_per_transaction` с 64 до 10000.
Увеличьте значение параметра и перегрузите экземпляр:

```
postgres=# alter system set max_locks_per_transaction = 10000;
ALTER SYSTEM
postgres=# \q
astra@tantor:~$ sudo restart
astra@tantor:~$ psql
```

5) Посмотрите как изменился размер структур памяти:

```
postgres=# SELECT name, allocated_size, pg_size_pretty(allocated_size) FROM
pg_shmem_allocations ORDER BY size DESC limit 4;
     name      | allocated_size | pg_size_pretty
---------------+----------------+----------------
               |      270582272 | 258 MB
```

```
 <anonymous>    |      228811520 | 218 MB
 Buffer Blocks  |      134221824 | 128 MB
 XLOG Ctl       |        4208256 | 4110 kB
(4 rows)
```

Размер структур памяти с названием `<anonymous>` увеличился с **4830 kB** до **218 MB**.
`<anonymous>` это один из типов выделения памяти. Анонимная память выделяется в оперативной памяти, не имеет отображения в файлы и устройства.
Про память с пустым названием (NULL) пишут, что "не используется".

6) Посмотрите размер структур LOCK и Proc:

```
postgres=# select * from (select *, lead(off) over(order by off) - off as
true_size  from pg_shmem_allocations) as a where name like '%LOCK%' or name like
'%roc%';
           name            |    off    |  size  | allocated_size | true_size
---------------------------+-----------+--------+----------------+-----------
 LOCK hash                 | 142667776 |  66384 |          66432 | 110935552
 PROCLOCK hash             | 253603328 | 131920 |         131968 | 114512768
 PREDICATELOCKTARGET hash  | 368120320 |   2896 |           2944 |    445312
 PREDICATELOCK hash        | 368565632 |   2896 |           2944 |   1260416
 Proc Header               | 370679552 |    136 |            256 |    152192
 Proc Array                | 370831744 |    544 |            640 |       640
 ProcSignal                | 371235968 |  11272 |          11392 |     11392
(7 rows)
```

Размер структур `PROCLOCK hash` и `LOCK hash` увеличился. Увеличение структур памяти типа хэш обычно не критичен. Эти структуры используются как вспомогательные: для быстрого поиска указателей на другие структуры. Способов поиска много. Напимер, списки, b-tree, radix-tree. Хэш-структуры примечательны тем, что для поиска по ним используется меньше всего циклов процессора. Основной недостаток - хэш структуры занимают намного больше памяти, чем другие виды структур для поиска. Поэтому если разработчики используют хэш-структуру, то это означает, что размер не имеет значения с точки зрения производительности.

Размер структуры **`Proc Header`** при увеличении не изменился. Эта структура часто опрашивается процессами и скорее всего она займёт кэши процессора или будет активно за них "конкурировать". Помимо размера важен состав структуры: выравнивание полей в записях структуры.

7) Верните значение `max_locks_per_transaction` в значение по умолчанию и увеличьте `max_connections` до `10000`:

```
postgres=# alter system reset max_locks_per_transaction;
ALTER SYSTEM
postgres=# alter system set max_connections = 10000;
ALTER SYSTEM
postgres=# \q
astra@tantor:~$ sudo restart
astra@tantor:~$ psql
```

8) Посмотрите как изменился размер структур памяти:

```
postgres=# SELECT name, allocated_size, pg_size_pretty(allocated_size) FROM
pg_shmem_allocations ORDER BY size DESC limit 4;
          name           | allocated_size | pg_size_pretty
-------------------------+----------------+----------------
 <anonymous>             |      328024448 | 313 MB
                         |      156285824 | 149 MB
 Buffer Blocks           |      134221824 | 128 MB
```

```
 Backend Activity Buffer |        10268672 | 10028 kB
(4 rows)
```

Размер структур памяти с названием <anonymous> увеличился с **4830 kB** до **313 MB**.
Можно проверить, что увеличение max_connections без увеличения числа сессий (серверных процессов) на TPS не влияет.

9) Посмотрите размер структур LOCK и Proc:

```
postgres=# select * from (select *, lead(off) over(order by off) - off as
true_size  from pg_shmem_allocations) as a where name like '%LOCK%' or name like
'%roc%';
           name            |    off    |  size  | allocated_size | true_size
---------------------------+-----------+--------+----------------+-----------
 LOCK hash                 | 144093440 |  33616 |          33664 |  58106752
 PROCLOCK hash             | 202200192 |  66384 |          66432 |  59769728
 PREDICATELOCKTARGET hash  | 261974144 |  33616 |          33664 |  39211904
 PREDICATELOCK hash        | 301186048 |  66384 |          66432 | 109208448
 Proc Header               | 458218240 |    136 |            256 |  11982848
 Proc Array                | 470201088 |  40144 |          40192 |     40192
 ProcSignal                | 496702080 | 882472 |         882560 |    882560
(7 rows)
```

После увеличения max_connections структуры разделяемой памяти Proc увеличились в размерах.

Часть 5. Параметр max_connections и производительность

1) Проверим как повлияет увеличение числа простаивающих серверных процессов на TPS.
Создайте тестовый скрипт:

```
postgres@tantor:~$ echo "select * from pg_sleep(1);" > test1.sql
```

2) Подготовьте кластера к тестам:

```
postgres@tantor:~$
psql -p 5435 -c "alter system set max_connections=100"
psql -p 5435 -c "alter system set max_locks_per_transaction = 64"
psql -c "alter system set max_connections=100"
psql -c "alter system set max_locks_per_transaction = 64"
pgbench -i 2> /dev/null
pgbench -i -p 5435 2> /dev/null
sudo systemctl restart postgresql
sudo systemctl restart tantor-se-server-16
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
```

3) Получите базовые показатели:

```
postgres@tantor:~$ pgbench -b select-only -T 30 -p 5435 2> /dev/null | grep tps
tps = 6709.693060 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -T 30 2> /dev/null | grep tps
tps = 7119.172960 (without initial connection time)
```

У СУБД Tantor SE tps = 7119.
У СУБД PostgreSQL, стандартно поставляемой с Astralinux tps = 6709.

Будет использоваться тест только с SELECT, так как тесты с изменениями в таблицах снизят точность при повторных запусках. Поскольку в виртуальной машине немного памяти (4Гб), влияние размеров структур памяти начинается примерно тогда же, когда Huge Pages начинают давать эффект (~100Гб), то различия ожидаются минимальными. Интервалы тестирования выбраны небольшими (30 секунд), так как нет смысла тратить время курса на ожидания. Практика полезна не абсолютными значениями, а процедурой измерений.

4) Сейчас максимальное число сессий 100. Создайте 90 сессий с помощью pgbench:

```
postgres@tantor:~$ pgbench -T 10000 -P 5 -c 90 -f test1.sql
pgbench (16.2)
starting vacuum...end.
progress: 5.0 s, 72.1 tps, lat 1002.351 ms stddev 1.874, 0 failed
progress: 10.0 s, 90.0 tps, lat 1001.731 ms stddev 0.528, 0 failed
progress: 15.0 s, 90.0 tps, lat 1001.525 ms stddev 0.493, 0 failed
```

Если не нужны соединения с базой данных, то тест можно прервать комбинацией клавиш **<ctrl+c>**
tps в тесте должны показываться как одинаковые целые числа, равное числу сессий при секундной задержке. Если они показываются не целыми числами это означает, что есть конкуренция за ядра процессоров и точность тестов будет ниже.

5) Во втором окне запустите тест, который выполняет только SELECT:

```
postgres@tantor:~$ pgbench -b select-only -T 30 2> /dev/null | grep tps
tps = 6940.512696 (without initial connection time)
```

6) Остановите нагрузку и запустите её с другой базой данных:

```
<ctrl+c>
postgres@tantor:~$ pgbench -T 10000 -P 5 -c 90 -f test1.sql -p 5435
pgbench (16.2, server 15.6 (Debian 15.6-astra.se2))
starting vacuum...end.
progress: 5.0 s, 71.9 tps, lat 1002.589 ms stddev 2.201, 0 failed
progress: 10.0 s, 90.1 tps, lat 1002.279 ms stddev 1.216, 0 failed
progress: 15.0 s, 90.0 tps, lat 1002.023 ms stddev 1.286, 0 failed
```

7) Во втором окне запустите тест:

```
postgres@tantor:~$ pgbench -b select-only -T 30  -p 5435 2> /dev/null | grep tps
tps = 6649.501401 (without initial connection time)
```

Произошло некоторое снижение tps. Можно предположить, что замедление вносит проверка статуса 90 процессов при выполнении каждого select из теста с частотой равной tps. Для проверки статуса нужна разделяемая блокировка на структуру памяти. При этом каждую секунду серверный процесс получает блокировку, чтобы поменять свой статус (номер транзакции, если он установлен).

8) Поменяем команду для нагрузки, чтобы у нагрузочных сессий в течение 1 секунды транзакция была открыта:

```
postgres@tantor:~$ echo "select *, pg_sleep(1) from pg_current_xact_id();" > test1.sql
```

Дальше повторим тесты. При реальном тестировании стоит перезапустить экземпляры, чтобы очистить структуры памяти.

9) Запустите нагрузку из 90 сессий:

```
postgres@tantor:~$ pgbench -T 10000 -P 5 -c 90 -f test1.sql
pgbench (16.2)
starting vacuum...end.
progress: 5.0 s, 71.7 tps, lat 1003.276 ms stddev 2.201, 0 failed
progress: 10.0 s, 90.2 tps, lat 1020.448 ms stddev 36.319, 0 failed
progress: 15.0 s, 90.1 tps, lat 1002.278 ms stddev 1.112, 0 failed
progress: 20.0 s, 90.0 tps, lat 1001.755 ms stddev 0.542, 0 failed
```

10) Запустите тест:

```
postgres@tantor:~$ pgbench -b select-only -T 30 2> /dev/null | grep tps
tps = 6854.086439 (without initial connection time)
```

11) Остановите нагрузку и запустите её с другой базой данных:

```
<ctrl+c>
postgres@tantor:~$ pgbench -T 10000 -P 5 -c 90 -f test1.sql -p 5435
pgbench (16.2, server 15.6 (Debian 15.6-astra.se2))
starting vacuum...end.
progress: 5.0 s, 72.0 tps, lat 1005.050 ms stddev 5.612, 0 failed
progress: 10.0 s, 89.9 tps, lat 1002.030 ms stddev 0.824, 0 failed
progress: 15.0 s, 90.0 tps, lat 1002.266 ms stddev 1.887, 0 failed
```

12) Запустите тест:

```
postgres@tantor:~$ pgbench -b select-only -T 30 -p 5435 2> /dev/null | grep tps
tps = 6763.148130 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -T 30 -p 5435 2> /dev/null | grep tps
tps = 6573.925660 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -T 30 -p 5435 2> /dev/null | grep tps
tps = 6649.730773 (without initial connection time)
```

Разброс значений у PostgreSQL увеличился. Можно провести несколько измерений.
В целом, tps уменьшился.

13) Остановите нагрузку и запустите её с другой базой данных:

```
<ctrl+c>
```

14) Увеличьте **max_connections** до 2000 и задержку до 10 секунд:

```
postgres@tantor:~$
echo "select *, pg_sleep(10) from pg_current_xact_id();" > test1.sql
psql -p 5435 -c "alter system set max_connections=2000"
psql -c "alter system set max_connections=2000"
ALTER SYSTEM
ALTER SYSTEM
```

Задержка увеличена, чтобы пробуждение процессов было редким и не оказывало влияние на результат теста. Таймслоты планировщика 10 миллисекунд. 4 процессора. Без ожидания обслуживания может пробуждаться 400 процессов в секунду, 4000 процессов раз в 10 секунд. 4000 больше, чем 2000 процессов, которые возможно запустить со значением параметра max_connections=2000. Удержание горизонта не играет роли, так как в тесте нет команд, которые меняют содержимое таблиц.

15) Остановите экземпляры. Размер памяти экземпляров увеличился на сотни мегабайт. Чтобы не было нехватки памяти, экземпляры будут запускаться по одному.

```
sudo systemctl stop postgresql
sudo systemctl stop tantor-se-server-16
sudo systemctl start tantor-se-server-16
```

16) Запустите нагрузку из 990 сессий:

```
pgbench -T 10000 -P 20 -c 990 -f test1.sql
pgbench (16.2)
starting vacuum...end.
progress: 20.0 s, 49.5 tps, lat 10096.327 ms stddev 27.359, 0 failed
progress: 40.0 s, 99.0 tps, lat 10013.549 ms stddev 6.854, 0 failed
progress: 60.0 s, 99.0 tps, lat 10010.448 ms stddev 3.005, 0 failed
```

17) Запустите тест:

```
postgres@tantor:~$ pgbench -b select-only -T 30 2> /dev/null | grep tps
tps = 6844.133619 (without initial connection time)
```

18) Остановите нагрузку и запустите её с другой базой данных:

```
<ctrl+c>
sudo systemctl stop tantor-se-server-16
sudo systemctl start postgresql
pgbench -T 10000 -P 20 -c 990 -f test1.sql -p 5435
```

19) Запустите тест:

```
postgres@tantor:~$ pgbench -b select-only -T 30 -p 5435 2> /dev/null | grep tps
tps = 6632.468942 (without initial connection time)
```

Изменения при увеличении числа сессий с 90 до 990 незаметны или отсутствуют. PostgreSQL оптимизирован для работы и с 4000 сессиями.
Более интересны 10000 сессий, но для них нужно около 20Гб памяти.

Пример проведения теста с 10000 сессий:

```
echo "select *, pg_sleep(100) from pg_current_xact_id();" > test1.sql
psql -p 5435 -c "alter system set max_connections=10240"
psql -c "alter system set max_connections=10240"
sudo systemctl stop postgresql
sudo systemctl stop tantor-se-server-16
sudo systemctl start tantor-se-server-16

postgres@tantor:~$ free
               total        used        free      shared  buff/cache   available
Mem:        24020472     1280012    22022956      373344     1421396    22740460
Swap:             0           0           0

postgres@tantor:~$ pgbench -T 10000 -P 30 -c 10000 -f test1.sql
pgbench (16.2)
starting vacuum...end.
progress: 81.6 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 90.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 120.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 150.1 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed

               total        used        free      shared  buff/cache   available
Mem:        24020472    17913240     5365736      384800     1479104     6107232
Swap:             0           0           0
```

```
postgres@tantor:~$ pgbench -b select-only -T 30 2> /dev/null | grep tps
tps = 6409.405397 (without initial connection time)
```

С 10000 неактивными сессиями tps уменьшился до 6409.

Для PostgreSQL нестабильность результата осталась прежней:

```
postgres@tantor:~$ pgbench -b select-only -T 30  -p 5435 2> /dev/null | grep tps
tps = 6692.845789 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -T 30  -p 5435 2> /dev/null | grep tps
tps = 6436.574882 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -T 30  -p 5435 2> /dev/null | grep tps
tps = 6547.961317 (without initial connection time)
```

Часть 6. Размер кэша буферов и освобождение буферов

1) Автоматизируем тестирование работы экземпляра на примере изменения размера буферного кэша. Изменения части параметров конфигурации требует рестарта экземпляра. В предыдущей части практики вы неоднократно рестартовали экземпляр и это трудоёмко и приводит к ошибкам.
Создайте файл скрипта для теста. Будем тестировать скорость выполнения команд, создающих и удаляющих таблицы.

```
postgres@tantor:~$ mcedit run.sql

CREATE TABLE x(id int);
INSERT INTO x VALUES (1);
DROP TABLE x;
```

2) Создайте файл теста:

```
postgres@tantor:~$ mcedit x.sh

#/bin/sh
pg_ctl stop
for x in '128MB' '1GB' '2GB' '4GB' '8GB' '16GB' '18GB'
do
 pg_ctl -l /dev/null -o '--shared_buffers='$x'' start
 sleep 1
 echo tps for $x
 psql -c 'select * from (select *, lead(off) over(order by off) - off as true
from pg_shmem_allocations) as a where a.true<>a.allocated_size order by 1;'
 pgbench --file=run.sql -j 1-T 10 2> /dev/null | grep tps
 pg_ctl stop
 sleep 1
done
```

Уберите значения '4GB' '8GB' '16GB' '18GB', так как памяти в виртуальной машине немного.

3) Поменяйте разрешения на исполнения файла скрипта:

```
postgres@tantor:~$ chmod +x x.sh
```

4) Выполните файл теста:

```
postgres@tantor:~$ ./x.sh
waiting for server to start.... done
server started
tps for 128MB
           name            |    off    | size  | allocated_size |   true
```

```
-----------------------------+------------+--------+-----------------+-----------
 LOCK hash                   | 144128000  | 33616  |           33664 |  59396992
 PREDICATELOCK hash          | 304476928  | 66384  |           66432 | 111420288
 PREDICATELOCKTARGET hash    | 264527744  | 33616  |           33664 |  39949184
 PredXactList                | 415897216  |    88  |             128 |  19703168
 Proc Header                 | 464834560  |   136  |             256 |  12269696
 PROCLOCK hash               | 203524992  | 66384  |           66432 |  60998528
 RWConflictPool              | 439937792  |    24  |             128 |  24628992
 SERIALIZABLEXID hash        | 435600384  |  4944  |            4992 |   4337408
 Shared Buffer Lookup Table  | 143166464  |  2896  |            2944 |    961408
(9 rows)

tps = 79.221247 (without initial connection time)
waiting for server to shut down..... done
server stopped
waiting for server to start.... done
server started
tps for 1GB
            name             |    off     |  size  | allocated_size |   true
-----------------------------+------------+--------+----------------+-----------
 LOCK hash                   | 1121593088 | 33616  |          33664 |  59396992
 PREDICATELOCK hash          | 1281942016 | 66384  |          66432 | 111420288
 PREDICATELOCKTARGET hash    | 1241992832 | 33616  |          33664 |  39949184
 PredXactList                | 1393362304 |    88  |            128 |  19703168
 Proc Header                 | 1442299648 |   136  |            256 |  12269696
 PROCLOCK hash               | 1180990080 | 66384  |          66432 |  60998528
 RWConflictPool              | 1417402880 |    24  |            128 |  24628992
 SERIALIZABLEXID hash        | 1413065472 |  4944  |           4992 |   4337408
 Shared Buffer Lookup Table  | 1114202880 |  9040  |           9088 |   7390080
(9 rows)

tps = 74.513412 (without initial connection time)
waiting for server to shut down..... done
server stopped
waiting for server to start.... done
server started
tps for 2GB
            name             |    off     |  size  | allocated_size |   true
-----------------------------+------------+--------+----------------+-----------
 LOCK hash                   | 2224302592 | 33616  |          33664 |  59396992
 PREDICATELOCK hash          | 2384651520 | 66384  |          66432 | 111420288
 PREDICATELOCKTARGET hash    | 2344702336 | 33616  |          33664 |  39949184
 PredXactList                | 2496071808 |    88  |            128 |  19703168
 Proc Header                 | 2545009152 |   136  |            256 |  12269696
 PROCLOCK hash               | 2283699584 | 66384  |          66432 |  60998528
 RWConflictPool              | 2520112384 |    24  |            128 |  24628992
 SERIALIZABLEXID hash        | 2515774976 |  4944  |           4992 |   4337408
 Shared Buffer Lookup Table  | 2209564160 | 17232  |          17280 |  14738304
(9 rows)

tps = 71.288637 (without initial connection time)
waiting for server to shut down..... done
server stopped
waiting for server to start.... done
server started
tps for 4GB
            name             |    off     |  size  | allocated_size |   true
-----------------------------+------------+--------+----------------+-----------
 LOCK hash                   | 4429721984 | 33616  |          33664 |  59396992
 PREDICATELOCK hash          | 4590070912 | 66384  |          66432 | 111420288
 PREDICATELOCKTARGET hash    | 4550121728 | 33616  |          33664 |  39949184
 PredXactList                | 4701491200 |    88  |            128 |  19703168
 Proc Header                 | 4750428544 |   136  |            256 |  12269696
 PROCLOCK hash               | 4489118976 | 66384  |          66432 |  60998528
 RWConflictPool              | 4725531776 |    24  |            128 |  24628992
 SERIALIZABLEXID hash        | 4721194368 |  4944  |           4992 |   4337408
 Shared Buffer Lookup Table  | 4400287104 | 33616  |          33664 |  29434752
(9 rows)

tps = 66.595864 (without initial connection time)
waiting for server to shut down..... done
server stopped
waiting for server to start.... done
server started
tps for 8GB
            name             |    off     |  size  | allocated_size |   true
-----------------------------+------------+--------+----------------+-----------
 LOCK hash                   | 8806510976 | 33616  |          33664 |  59396992
 PREDICATELOCK hash          | 8966859904 | 66384  |          66432 | 111420288
 PREDICATELOCKTARGET hash    | 8926910720 | 33616  |          33664 |  39949184
 PredXactList                | 9078280192 |    88  |            128 |  19703168
```

```
 Proc Header                | 9127217536 |   136 |            256 |  12269696
 PROCLOCK hash              | 8865907968 | 66384 |          66432 |  60998528
 RWConflictPool             | 9102320768 |    24 |            128 |  24628992
 SERIALIZABLEXID hash       | 9097983360 |  4944 |           4992 |   4337408
 Shared Buffer Lookup Table | 8747683200 | 66384 |          66432 |  58827648
(9 rows)
```

tps = 59.421421 (without initial connection time)

```
waiting for server to shut down..... done
server stopped
waiting for server to start.... done
server started
```

tps for 16GB

```
            name             |     off     |  size  | allocated_size |   true
-----------------------------+-------------+--------+----------------+-----------
 LOCK hash                   | 17560088960 |  33616 |          33664 |  59396992
 PREDICATELOCK hash          | 17720437888 |  66384 |          66432 | 111420288
 PREDICATELOCKTARGET hash    | 17680488704 |  33616 |          33664 |  39949184
 PredXactList                | 17831858176 |     88 |            128 |  19703168
 Proc Header                 | 17880795520 |    136 |            256 |  12269696
 PROCLOCK hash               | 17619485952 |  66384 |          66432 |  60998528
 RWConflictPool              | 17855898752 |     24 |            128 |  24628992
 SERIALIZABLEXID hash        | 17851561344 |   4944 |           4992 |   4337408
 Shared Buffer Lookup Table  | 17442475392 | 131920 |         131968 | 117613440
(9 rows)
```

tps = 48.098817 (without initial connection time)

```
waiting for server to shut down..... done
server stopped
waiting for server to start.... done
server started
```

tps for 18GB

```
            name             |     off     |  size  | allocated_size |   true
-----------------------------+-------------+--------+----------------+-----------
 LOCK hash                   | 19744272768 |  33616 |          33664 |  59396992
 PREDICATELOCK hash          | 19904621696 |  66384 |          66432 | 111420288
 PREDICATELOCKTARGET hash    | 19864672512 |  33616 |          33664 |  39949184
 PredXactList                | 20016041984 |     88 |            128 |  19703168
 Proc Header                 | 20064979328 |    136 |            256 |  12269696
 PROCLOCK hash               | 19803669760 |  66384 |          66432 |  60998528
 RWConflictPool              | 20040082560 |     24 |            128 |  24628992
 SERIALIZABLEXID hash        | 20035745152 |   4944 |           4992 |   4337408
 Shared Buffer Lookup Table  | 19616173440 | 131920 |         131968 | 128099200
(9 rows)
```

tps = 46.381753 (without initial connection time)

```
waiting for server to shut down..... done
server stopped
```

Была протестирована скорость удаления таблиц в зависимости от размера кэша буферов. Скорость удаления таблиц достоверно уменьшается при увеличении размера буферного кэша даже при пустом буферном кэше.
Такое происходит после удаления объекта, когда буфера, содержащие его блоки инвалидируются или при удалении отдельного файла или усечении файла (в том числе вакуумом) или при удалении базы данных.

5) Верните параметры к значениям по умолчанию и перезапустите экземпляр:

```
postgres@tantor:~$
sudo systemctl start tantor-se-server-16
sudo systemctl start postgresql
psql -p 5435 -c "alter system reset max_locks_per_transaction"
psql -p 5435 -c "alter system reset max_connections"
psql -c "alter system reset max_connections"
sudo systemctl restart tantor-se-server-16
sudo systemctl restart tantor-se-server-16
```

Практика к главе 6

Часть 1. Карта свободного пространства

1) Установите расширение pageinspect:

```
postgres=# create extension if not exists pageinspect;
CREATE EXTENSION
```

2) Создайте две таблицы с разным порядком следования столбцов и вставьте строки:

```
postgres=#
begin;
drop table if exists t1;
drop table if exists t2;
create table t1 (c1 varchar(1), c2 bigserial , c3 date, c4 timestamp);
create table t2 (c1 bigserial , c2 timestamp, c3 date, c4 varchar(1));
insert into t1 values('A', 1, now(), current_timestamp);
insert into t2 values(1, current_timestamp, now(), 'A');
commit;
BEGIN
CREATE TABLE
CREATE TABLE
INSERT 0 1
INSERT 0 1
COMMIT
```

3) Посмотрите используя функции расширения pageinspect содержимое вставленных в таблицы строк:

```
postgres=# select t_data, lp_len, t_hoff from
heap_page_items(get_raw_page('t1','main',0));
                                t_data                                 | lp_len | t_hoff
-----------------------------------------------------------------------+--------+--------
 \x0541000000000000010000000000000009a2300000000000009e7a078838cc0200 |     56 |     24
(1 row)

postgres=# select t_data, lp_len, t_hoff from
heap_page_items(get_raw_page('t2','main',0));
                    t_data                     | lp_len | t_hoff
-----------------------------------------------+--------+--------
 \x0100000000000000009e7a078838cc02009a2300000541 |     46 |     24
(1 row)
```

В таблице t1 строка занимает больше места, чем в таблице t2. Значения в полях строк в обеих таблицах одинаковы, только порядок столбцов разный.

4) Посмотрите содержимое первых четырёх блоков карты свободного пространства таблицы `pg_class`. В этой таблице много блоков данных и строк в них:

```
postgres=#
SELECT * from fsm_page_contents(get_raw_page('pg_class', 'fsm', 0));
SELECT * from fsm_page_contents(get_raw_page('pg_class', 'fsm', 1));
SELECT * from fsm_page_contents(get_raw_page('pg_class', 'fsm', 2));
SELECT * from fsm_page_contents(get_raw_page('pg_class', 'fsm', 3));
 fsm_page_contents
-------------------
 0: 236           +
 1: 236           +
 3: 236           +
```

```
 7: 236           +
 15: 236          +
 31: 236          +
 63: 236          +
 127: 236         +
 255: 236         +
 511: 236         +
 1023: 236        +
 2047: 236        +
 4095: 236        +
 fp_next_slot: 0  +

(1 row)
...
 4095: 189        +
 4096: 115        +
 4097: 2          +
 4098: 2          +
 4099: 2          +
 4100: 2          +
 4101: 154        +
 4102: 236        +
 4103: 12         +
 4104: 3          +
 4105: 6          +
 4106: 10         +
 4107: 8          +
 4108: 11         +
 4109: 189        +
 fp_next_slot: 1  +

(1 row)
ERROR:  block number 3 is out of range for relation "pg_class"
```

Четвертого блока нет.
Для каждого блока данных в карте хранится 1 байт. Байт показывает свободное пространство, деленное на 32 с округлением в меньшую сторону. Для быстрого поиска свободного блока карта представляет собой не просто список, а имеет древовидную структуру над списком. Дерево хранится на каждой странице FSM как массив и оно не идеально. Практического смысла читать значения чисел в блоках FSM нет.

Алгоритм работы с картой предусматривает, чтобы разным серверным процессам выдавались разные блоки, чтобы снизить конкуренцию за доступ к блокам данных. Разные серверные процессы будут вставлять строки (или версии строк) в разные блоки. Алгоритм учитывает, чтобы по возможности блоки данных заполнялись последовательно, так как это позволяет использовать предварительную выборку блоков (prefetching).

5) Полностью вакуумируйте таблицу системного каталога `pg_class` и посмотрите карту свободного места FSM и видимости VM:

```
postgres=# vacuum full pg_class;
VACUUM
postgres=# SELECT * from fsm_page_contents(get_raw_page('pg_class', 'fsm', 0));
ERROR:  could not open file "base/5/535511_fsm": No such file or directory
postgres=# SELECT * from page_header(get_raw_page('pg_class', 'vm', 0));
ERROR:  could not open file "base/5/535511_vm": No such file or directory
```

Полный вакуум не создал FSM и VM слои, это нормально.

6) Выполните обычное вакуумирование:

```
postgres=# vacuum pg_class;
VACUUM
```

7) Посмотрите карту FSM и VM:

```
postgres=# SELECT * from fsm_page_contents(get_raw_page('pg_class', 'fsm', 0));
 fsm_page_contents
-------------------
 0: 159           +
 1: 159           +
 3: 159           +
 7: 159           +
 15: 159          +
 31: 159          +
 63: 159          +
 127: 159         +
 255: 159         +
 511: 159         +
 1023: 159        +
 2047: 159        +
 4095: 159        +
 fp_next_slot: 0  +

(1 row)
postgres=# SELECT * from page_header(get_raw_page('pg_class', 'vm', 0));
   lsn     | checksum | flags | lower | upper | special | pagesize | version | prune_xid
-----------+----------+-------+-------+-------+---------+----------+---------+----------
0/8BEFFDB0 |        0 |     0 |    24 |  8192 |    8192 |     8192 |       5 |         0
(1 row)
```

Обычное вакуумирование создало слои (файлы) FSM и VM.

Изменения в блоках FSM явно не журналируются, так как в алгоритме есть процедуры самокоррекции. Увеличение размера файла FSM не журналируется и расхождение в размерах не вызывает ошибки.
При обнаружении повреждений процесс, который обнаружил несостыковки в дереве, пытается скорректировать ошибки и перестроить часть дерева.
Обычные вакуум и автовакуум обновляет листовые страницы дерева FSM и пересоздает FSM и VM, если файлы FSM отсутствуют.

8) Проверим может ли создать серверный процесс FSM и когда.
Посмотрите есть ли карта свободного пространства у таблицы t1:

```
postgres=# SELECT * from fsm_page_contents(get_raw_page('t1', 'fsm', 0));
ERROR:  could not open file "base/5/535438_fsm": No such file or directory
```

Файла карты свободного пространства нет. Как только блок заполнится и серверный процесс захочет внести изменение в карту свободного пространства, он ее создаст.

9) Выполните анонимный блок для вставки 100 строк в таблицу t1:

```
postgres=#
DO $$
DECLARE
BEGIN
 FOR r IN 1..100 LOOP
 insert into t1 values('A', 1, clock_timestamp(), current_timestamp);
 END LOOP;
END$$;
```

DO

10) Проверьте создалась ли карта:

```
postgres=# SELECT * from fsm_page_contents(get_raw_page('t1', 'fsm', 0));
ERROR:  could not open file "base/5/535438_fsm": No such file or directory
```

Карта FSM не создалась серверным процессом.

11) Почему не создалась карта? Проверьте сколько блоков в таблице. Выполните запрос:

```
postgres=# select max(ctid) from t1;
   max
---------
 (0,100)
(1 row)
```

Один блок и в нём 100 строк.

12) Вставьте ещё 100 строк и проверьте сколько блоков в основном слое (файлах данных):

```
postgres=#
DO $$
DECLARE
BEGIN
 FOR r IN 1..100 LOOP
 insert into t1 values('A', 1, clock_timestamp(), current_timestamp);
 END LOOP;
END$$;

DO
postgres=# select max(ctid) from t1;
  max
--------
 (1,65)
(1 row)
```

Два блока. Во втором блоке 65 строк.

13) Проверьте, что файл FSM создан:

```
postgres=# SELECT * from fsm_page_contents(get_raw_page('t1', 'fsm', 0));
 fsm_page_contents
--------------------
 fp_next_slot: 0  +

(1 row)
```

14) Проверьте, что содержимое первого лока действительно пусто:

```
postgres=# SELECT * from page_header(get_raw_page('t1', 'fsm', 0));
 lsn | checksum | flags | lower | upper | special | pagesize | version | prune_xid
-----+----------+-------+-------+-------+---------+----------+---------+-----------
 0/0 |        0 |     0 |    24 |  8192 |    8192 |     8192 |       5 |         0
(1 row)
```

`upper` указывает на конец блока.

Хотя серверный процесс создал файл FSM, он его не заполнил. Почему? Потому что для заполнения нужно прочесть все блоки данных таблицы, а это не задача серверного процесса и привело бы к задержке выполнения команды. VM файл не был создан.
Только обычный вакуум и автовакуум заполнят FSM и VM. Серверные процессы вносят изменения в заполненную FSM и исправляют повреждения части дерева FSM.

Часть 2. Изменение порядка следования столбцов

1) Если на виртуальной машине есть доступ в сеть, скачайте архив с демонстрационной базой данных:

```
postgres@tantor:~$ wget https://edu.postgrespro.com/demo-medium-en.zip
Saving to: 'demo-medium.zip'
'demo-medium-en.zip' saved [64544920/64544920]
```

Демонстрационная база данных распространяется под лицензией PostgreSQL.
Демонстрационная база есть в трёх вариантах **demo-small.zip demo-medium.zip demo-big.zip**
Демонстрационная база также есть по адресу:

```
postgres@tantor:~$ wget https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/example-guided-project/flights_RUSSIA_small.sql
'flights_RUSSIA_small.sql' saved [103865229/103865229]
postgres=# \i flights_RUSSIA_small.sql
```

2) Импортируйте базу данных:

```
postgres@tantor:~$ time zcat demo-medium.zip | psql
SET
SET
...
real    1m3.339s
user    0m2.168s
sys     0m0.255s
```

3) Воспользуемся скриптом, который может показать оптимальный порядок следования столбцов. Скачайте скрипт:

```
postgres@tantor:~$ wget https://raw.githubusercontent.com/NikolayS/postgres_dba/refs/heads/master/sql/p1_alignment_padding.sql
'p1_alignment_padding.sql' saved [6598/6598]
postgres@tantor:~$ cat p1_alignment_padding.sql
```

4) Подсоединитесь к базе данных demo:

```
postgres@tantor:~$ psql -d demo
```

5) Выполните скрипт:

```
demo=# \i p1_alignment_padding.sql
          Table          |  Size  |     Wasted      |      Suggested Columns Reorder
-------------------------+--------+-----------------+-------------------------------------
 bookings.ticket_flights | 155 MB | ~18 MB (11.61%) | amount, fare_conditions, flight_id, ticket_no
bookings.boarding_passes | 110 MB | ~14 MB (13.13%) | boarding_no, flight_id, seat_no, ticket_no
 bookings.tickets        | 109 MB | ~6.4MB (5.79%)  | book_ref, contact_data, passenger_id,
                                                       passenger_name, ticket_no
 bookings.bookings       | 30 MB  |                 |
 bookings.flights        | 6.7MB  |                 |
 bookings.seats          | 96 kB  |                 |
 bookings.airports_data  | 48 kB  |~832 b (1.69%)   | airport_code, airport_name, city, timezone,
                                                       coordinates
```

```
 bookings.aircrafts_data| 8192 b|                    |
(8 rows)
```

Для 4 таблиц скрипт нашёл более оптимальный порядок следования столбцов. Ожидаемая экономия места по расчётам скрипта ~11%.

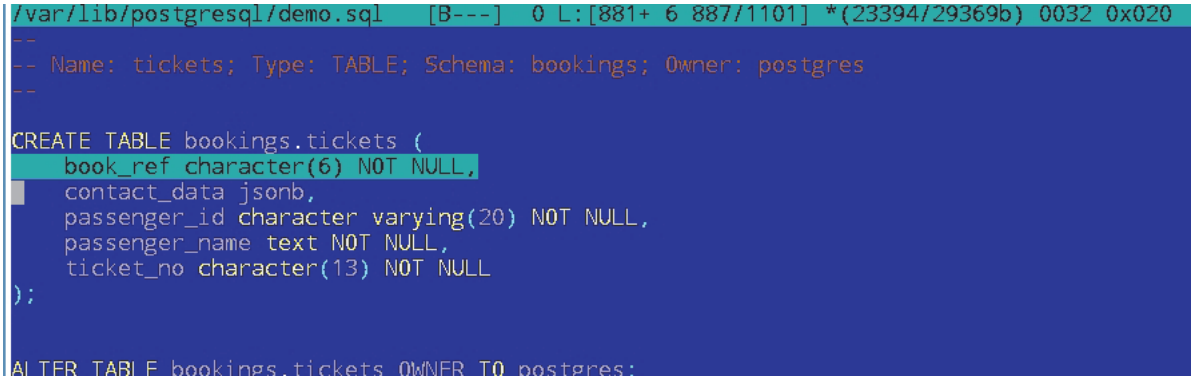
6) Выгрузите определения объектов базы данных demo:

```
postgres@tantor:~$ pg_dump -d demo -s -f demo.sql
```

Параметр -s указывает не выгружать строки таблиц.
Параметр -f название файла, куда выгружаются команды создания и изменения объектов (файл дампа)
Параметр -d указывает к какой базе данных подключиться, чтобы выгрузить её содержимое

7) Отредактируйте скрипт, поменяв порядок столбцов для таблиц `ticket_flights, boarding_passes, tickets`:

```
postgres@tantor:~$ mcedit demo.sql
```



8) Создайте базу данных с названием demo2, подсоединитесь к ней и выполните отредактированный скрипт `demo.sql`:

```
postgres=#
create database demo2;
\c demo2 \\
\i demo.sql
CREATE DATABASE
You are now connected to database "demo2" as user "postgres".
...
ALTER TABLE
demo2=#
```

9) Перегрузите данные из базы данных demo в базу данных demo2:

```
postgres@tantor:~$ time pg_dump -d demo -a | psql -d demo2
...
real    3m38.040s
user    0m1.981s
sys     0m0.259s
```

Перегрузка выполняется в один поток и будет длится ~3 минуты.
За эти три минуты выполните следующие два пункта практики.

10) Пока работает команда перегрузки данных выполните запрос:

```
demo2=# select * from pg_stat_progress_copy;
```

```
  pid  | datid  | datname | relid  |  command  | type |bytes_processed |bytes_total|tup_processed | excluded
-------+--------+---------+--------+-----------+------+----------------+-----------+--------------+--------
 59160 | 535928 | demo    | 535978 | COPY TO   | PIPE |         217467 |          0|         5654 |        0
 59161 | 536055 | demo2   | 536110 | COPY FROM | PIPE |       83424852 |          0|       829071 |        0
(2 rows)
```

Запрос позволяет оценить скорость выполнения команд COPY. Отдельного представления для наблюдения за прогрессом выполнения выгрузки-загрузки нет, так как выполнение demo.sql с точки зрения серверного процесса это набор отдельных команд.
Кроме представления **pg_stat_progress_copy** есть ещё несколько представлений для мониторинга прогресса выполнения некоторых, обычно долго выполняющихся команд:

```
postgres=# \dv *progress*
                    List of relations
   Schema   |              Name               | Type |  Owner
------------+---------------------------------+------+----------
 pg_catalog | pg_stat_progress_analyze        | view | postgres
 pg_catalog | pg_stat_progress_basebackup     | view | postgres
 pg_catalog | pg_stat_progress_cluster        | view | postgres
 pg_catalog | pg_stat_progress_copy           | view | postgres
 pg_catalog | pg_stat_progress_create_index   | view | postgres
 pg_catalog | pg_stat_progress_vacuum         | view | postgres
(6 rows)
```

11) Пока работает команда перегрузки данных выполните запрос:

```
postgres=# select * from pg_stat_activity where query like 'COPY%' \gx
-[ RECORD 1 ]----+-----------------------------------------------
datid            | 535645
datname          | demo2
pid              | 57607
leader_pid       |
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 17:50:39.986712+03
xact_start       | 17:59:27.486498+03
query_start      | 17:59:27.486498+03
state_change     | 17:59:27.4865+03
wait_event_type  | IO
wait_event       | DataFileWrite
state            | active
backend_xid      | 406744
backend_xmin     | 406744
query_id         |
query            | COPY bookings.boarding_passes (ticket_no, flight_id, boarding_no, seat_no) FROM
stdin;
backend_type     | client backend
```

12) После того как данные перегрузятся, выполните запрос:

```
postgres=# select pg_size_pretty(a), pg_size_pretty(b), 100*(a-b)/(a+b) "%" from
(select pg_database_size('demo') a , pg_database_size('demo2') b);
 pg_size_pretty | pg_size_pretty | %
----------------+----------------+----
  706 MB        | 815 MB         | -7
(1 row)
```

Запрос показывает размеры исходной базы данных, оптимизированной и процент экономии места. Место наоборот увеличилось. Почему такое произошло?
Потому, что данные загружались с созданными индексами. Из-за этого время загрузки данных существенно увеличилось и размеры индексов стали больше.

13) Посмотрите размеры индексов в двух базах данных demo и demo2:

```
demo2=# \di+ bookings.*
                               List of relations
  Schema  |                      Name                        |     Table      | Size   |
----------+--------------------------------------------------+----------------+--------+
 bookings | aircrafts_pkey                                   |aircrafts_data  | 16 kB  |
 bookings | airports_data_pkey                               |airports_data   | 16 kB  |
 bookings | boarding_passes_flight_id_boarding_no_key        |boarding_passes | 281 MB |
 bookings | boarding_passes_flight_id_seat_no_key            |boarding_passes | 274 MB |
 bookings | boarding_passes_pkey                             |boarding_passes | 404 MB |
 bookings | bookings_pkey                                    |bookings        | 45 MB  |
 bookings | flights_flight_no_scheduled_departure_key        |flights         | 9552 kB |
 bookings | flights_pkey                                     |flights         | 7328 kB |
 bookings | seats_pkey                                       |seats           | 48 kB  |
 bookings | ticket_flights_pkey                              |ticket_flights  | 423 MB |
 bookings | tickets_pkey                                     |tickets         | 99 MB  |
(11 rows)

demo2=# \c demo
You are now connected to database "demo" as user "postgres".
demo=# \di+ bookings.*
                               List of relations
  Schema  |                      Name                        |     Table      |  Size   |
----------+--------------------------------------------------+----------------+---------+
 bookings | aircrafts_pkey                                   |aircrafts_data  | 16 kB   |
 bookings | airports_data_pkey                               |airports_data   | 16 kB   |
 bookings | boarding_passes_flight_id_boarding_no_key        |boarding_passes | 170 MB  |
 bookings | boarding_passes_flight_id_seat_no_key            |boarding_passes | 170 MB  |
 bookings | boarding_passes_pkey                             |boarding_passes | 307 MB  |
 bookings | bookings_pkey                                    |bookings        | 45 MB   |
 bookings | flights_flight_no_scheduled_departure_key        |flights         | 6648 kB |
 bookings | flights_pkey                                     |flights         | 4744 kB |
 bookings | seats_pkey                                       |seats           | 48 kB   |
 bookings | ticket_flights_pkey                              |ticket_flights  | 325 MB  |
 bookings | tickets_pkey                                     |tickets         | 89 MB   |
(11 rows)
```

Семь индексов увеличились в размере. Индекс `bookings_pkey` не поменял размер. Чем он отличается от других индексов? Тем, что данные столбца, по которому он создан заполнялись монотонно возрастающими значениями:

```
demo2=# select * from bookings.bookings limit 3;
 book_ref |        book_date         | total_amount
----------+------------------------+--------------
 00000F   | 2017-07-05 03:12:00+03 |    265700.00
 000012   | 2017-07-14 09:02:00+03 |     37900.00
 00002D   | 2017-05-20 18:45:00+03 |    114700.00
(3 rows)
```

Структура индексов типа b-tree в PostgreSQL при вставке монотонно возрастающих значений заполняются оптимально. Такое происходит, если индекс создан по столбцу, заполняющемуся возрастающей последовательностью.

14) Перестройте индексы. Это можно сделать несколькими способами.
Первый способ выполнить команду:

```
demo=# \c demo2
demo2=# SELECT 'REINDEX INDEX ' || indexrelid::regclass || ';' FROM pg_index
where indexrelid::regclass::text like 'bookings.%'; \gexec
                               ?column?
--------------------------------------------------------------------
 REINDEX INDEX bookings.aircrafts_pkey;
 REINDEX INDEX bookings.airports_data_pkey;
 REINDEX INDEX bookings.boarding_passes_flight_id_boarding_no_key;
 REINDEX INDEX bookings.boarding_passes_flight_id_seat_no_key;
 REINDEX INDEX bookings.boarding_passes_pkey;
```

```
 REINDEX INDEX bookings.bookings_pkey;
 REINDEX INDEX bookings.flights_flight_no_scheduled_departure_key;
 REINDEX INDEX bookings.flights_pkey;
 REINDEX INDEX bookings.seats_pkey;
 REINDEX INDEX bookings.ticket_flights_pkey;
 REINDEX INDEX bookings.tickets_pkey;
(11 rows)
```

Команда удобна тем, что можно добавить предикат (условие WHERE) и отфильтровать те индексы, которые нужно перестроить.
Второй и третий способ выполнить команды:

```
demo2=# \timing on
Timing is on.
demo2=# REINDEX SCHEMA bookings;
REINDEX
Time: 27179.519 ms (00:27.180)
demo2=# REINDEX DATABASE;
REINDEX
Time: 28331.634 ms (00:28.332)
demo2=# \timing off
Timing is off.
```

Предполагается, что в таблицы не вносятся изменения в параллельных сессиях, поэтому REINDEX оптимальнее, чем REINDEX INDEX CONCURRENTLY. Команды REINDEX INDEX CONCURRENTLY выполняют два прохода по индексируемым данным, что медленнее чем один проход, который выполняет команда REINDEX.

15) Сравните время перестройки индекса с опцией CONCURRENTLY и обычной перестройки. Для этого выполните команды:

```
postgres@tantor:~$ sudo restart
postgres@tantor:~$ time psql -d demo2 -c "REINDEX INDEX CONCURRENTLY
bookings.boarding_passes_pkey;"
REINDEX

real    0m9.278s
user    0m0.009s
sys     0m0.000s
postgres@tantor:~$ sudo restart
postgres@tantor:~$ time psql -d demo2 -c "REINDEX INDEX
bookings.boarding_passes_pkey;"
REINDEX

real    0m7.982s
user    0m0.009s
sys     0m0.000s
```

Перезапуск экземпляра выполнялся для очистки буферного кэша. **Перестройка индекса с CONCURRENTLY медленнее на 16%.**

16) Как предотвратить создание индексов перед загрузкой данных? Таких параметров у pg_dump нет. При редактировании скрипта demo.sql в конце скрипта есть команды:

```
ALTER TABLE ONLY ..
    ADD CONSTRAINT ..
```

Этими командами создаются индексы. Можно скопировать скрипт, провести автозамену "`ADD CONSTRAINT`" на " `--ADD CONSTRAINT`". В конце набора команд с закомментированными частями добавить ";". Пример:

```
postgres@tantor:~$ cp demo.sql demo-withoutindexs.sql
postgres@tantor:~$ mcedit demo-withoutindexs.sql
```

```
<F4>
postgres@tantor:~$ cat demo-withoutindexs.sql | tail -16
ALTER TABLE ONLY bookings.ticket_flights
    -- ADD CONSTRAINT ticket_flights_ticket_no_fkey FOREIGN KEY (ticket_no) REFERENCES
bookings.tickets(ticket_no);


--
-- Name: tickets tickets_book_ref_fkey; Type: FK CONSTRAINT; Schema: bookings; Owner: postgres
--

ALTER TABLE ONLY bookings.tickets
    -- ADD CONSTRAINT tickets_book_ref_fkey FOREIGN KEY (book_ref) REFERENCES bookings.bookings(book_ref);
;

--
-- PostgreSQL database dump complete
--
```

Аналогично с командами создания индексов, если такие будут, но таких команд в снет, так как нет "аналитических индексов" (индексы, которые не используются ограничениями целостности, а используются для ускорения запросов). Последовательность действий:
a) выполнить изменённый скрипт `demo-withoutindexs.sql`
b) перегрузить данные командой `pg_dump -d demo -s -f demo.sql` **Команда перегрузки данных вместо 3 минут выполнится за 10 секунд.** Наличие индексов существенно **(в 12 раз)** замедлило внесение изменений в таблицы.
c) выполнить скрипт `demo.sql`. При выполнении скрипта `demo.sql` все команды выдадут ошибки, кроме добавления ограничений целостности.

17) Выполните запрос:

```
demo2=# select pg_size_pretty(a), pg_size_pretty(b), 100*(a-b)/(a+b) "%" from
(select pg_database_size('demo') a , pg_database_size('demo2') b);
 pg_size_pretty | pg_size_pretty | %
----------------+----------------+---
 706 MB         | 706 MB         | 0
(1 row)
```

Размер баз стал одинаковым.
Однако, перестановка столбцов по рекомендации скрипта, не привела к успеху. Размер самой большой таблицы остался тем же:

```
demo2=# select pg_table_size('bookings.ticket_flights');
 pg_table_size
---------------
     162660352
(1 row)

demo2=# \c demo
You are now connected to database "demo" as user "postgres".
demo=# select pg_table_size('bookings.ticket_flights');
 pg_table_size
---------------
     162660352
(1 row)
```

18) Список функций, в названии которых есть size:

```
demo2=# \df pg*size*
                                  List of functions
   Schema   |          Name           | Result data type | Argument data types | Type
------------+-------------------------+------------------+---------------------+------
 pg_catalog | pg_column_size          | integer          | "any"               | func
 pg_catalog | pg_database_size        | bigint           | name                | func
 pg_catalog | pg_database_size        | bigint           | oid                 | func
 pg_catalog | pg_indexes_size         | bigint           | regclass            | func
 pg_catalog | pg_relation_size        | bigint           | regclass            | func
 pg_catalog | pg_relation_size        | bigint           | regclass, text      | func
 pg_catalog | pg_size_bytes           | bigint           | text                | func
 pg_catalog | pg_size_pretty          | text             | bigint              | func
```

```
 pg_catalog | pg_size_pretty         | text             | numeric          | func
 pg_catalog | pg_table_size          | bigint           | regclass         | func
 pg_catalog | pg_tablespace_size     | bigint           | name             | func
 pg_catalog | pg_tablespace_size     | bigint           | oid              | func
 pg_catalog | pg_total_relation_size | bigint           | regclass         | func
(13 rows)
```

Имеется функция `pg_column_size(значение)`, которая показывает, сколько байт займёт при сохранении значение. Если в аргументы функции передать имя столбца таблицы, то функция выдаст, сколько реально места занимает поле в блоке таблицы с учетом сжатия, padding, вытеснения в TOAST. Если поле хранится в сжатом виде, то выдаст, сколько занимает поле в сжатом виде.
Функции для измерения размера строки не существует.

19) С помощью, функции `pg_column_size(таблица.*)` можно посмотреть размер всей строки вместе с заголовком. При этом функция выдает размер строки внутри блока без padding всей строки. Пример измерения реального размера области данных строк таблиц без padding всей строки:

```
demo2=# select pg_column_size(bookings.ticket_flights.*) from bookings.ticket_flights limit 1;
 pg_column_size
----------------
             58  -> 64
(1 row)

demo2=# \c demo
You are now connected to database "demo" as user "postgres".
demo=# select pg_column_size(bookings.ticket_flights.*) from bookings.ticket_flights limit 1;
 pg_column_size
----------------
             60  -> 64
(1 row)

demo=# select pg_column_size(bookings.tickets.*) from bookings.tickets limit 1;
 pg_column_size
----------------
            103  -> 104
(1 row)

demo=# \c demo2
You are now connected to database "demo2" as user "postgres".
demo2=# select pg_column_size(bookings.tickets.*) from bookings.tickets limit 1;
 pg_column_size
----------------
            103  -> 104
(1 row)

demo2=# select pg_column_size(bookings.boarding_passes.*) from bookings.boarding_passes limit 1;
 pg_column_size
----------------
             49  -> 56
(1 row)

demo2=# \c demo
You are now connected to database "demo" as user "postgres".
demo=# select pg_column_size(bookings.boarding_passes.*) from bookings.boarding_passes limit 1;
 pg_column_size
----------------
             51  -> 56
(1 row)
```

Разница небольшая - всего несколько байт.
Эта разница нивелируется paddingом всей строки: длинна всей строки (заголовок плюс область данных) должна быть кратна 8 байтам.
Кроме padding отдельных столбцов ВСЕГДА выполняется выравнивание (aligning) всей строки до 8 байт. Заголовок строки плюс область данных - это полный размер строки и именно он выравнивается до 8 байт.
Перестановка столбцов для таблиц демонстрационной базы данных эффекта не дала.

Размер всей строки должен быть кратным 8, а значит равен: .., 48, 56, 64,.., 96, 104,...
Пять предыдущих SELECT выдали числа: 58 - дополняется до **64**; 60 - дополняется до **64**; 49 и 51 дополняются до **56**.
Если текст пункта не понятен, перечитайте этот пункт ещё раз. Это может быть сложным для понимания, как и любая новая информация. Если один раз ощущение понимания пришло, то позже вы легко вспомните об этом. Даже высококвалифицированные специалисты упускают "padding всей строки". В частности, в сложном скрипте **p1_alignment_padding.sql** "padding всей строки" не учтён.

Интересно то, что если бы функция `pg_column_size(таблица.*)` по таблицам с переставленными строками выдала значения 64 и 65, то экономия места на каждой строке была бы 7 байт. А если функция выдаёт 57 и 63, то разницы нет.
Примечание: на 64-разрядных операционных системах выравнивание по 8 байт, на 32-разрядных по 4 байта. Выравнивание сохранено в управляющем файле и его можно посмотреть утилитой командной строки:

```
postgres@tantor:~$ pg_controldata | grep align
Maximum data alignment:               8
```

19) Проверим, реагирует и скрипт на те таблицы, где перестановка столбцов даст эффект. Создадим две таблицы с разным порядком следования столбцов и одинаковыми данными. Одна таблица из-за padding будет занимать на 18% больше места.

```
demo=#
drop table if exists t1;
drop table if exists t2;
create table t1 (c1 char(1), c2 integer, c3 char(1), c4 integer, c5 char(1), c6
integer);
create table t2 (c1 integer, c2 integer, c3 integer, c4 char(1), c5 char(1), c6
char(1));
ALTER TABLE t1 SET ( fillfactor = 100);
ALTER TABLE t2 SET ( fillfactor = 100);
DO
$$
BEGIN
FOR i IN 1 .. 500000 LOOP
INSERT INTO t1 VALUES ('1', 1,'1', 1,'1', null);
INSERT INTO t2 VALUES (null, 1, 1, '1','1','1');
END LOOP;
END;
$$;
DROP TABLE
DROP TABLE
CREATE TABLE
CREATE TABLE
ALTER TABLE
ALTER TABLE
DO
demo=# \i p1_alignment_padding.sql
          Table           | Table Size |      Wasted *       |      Suggested Columns Reorder
--------------------------+------------+---------------------+-------------------------------------
 bookings.ticket_flights  | 155 MB     | ~18 MB (11.61%)     | amount, fare_conditions, flight_id  +
                          |            |                     | ticket_no
 bookings.boarding_passes | 110 MB     | ~14 MB (13.13%)     | boarding_no, flight_id, seat_no     +
                          |            |                     | ticket_no
 bookings.tickets         | 109 MB     | ~6477 kB (5.79%)    | book_ref, contact_data, passenger_id+
                          |            |                     | passenger_name, ticket_no
 bookings.bookings        | 30 MB      |                     |
 bookings.t1              | 25 MB      |                     |
 bookings.t2              | 21 MB      |                     |
 bookings.flights         | 6688 kB    |                     |
 bookings.seats           | 96 kB      |                     |
 bookings.airports_data   | 48 kB      | ~832 bytes (1.69%)  | airport_code, airport_name, city    +
                          |            |                     | timezone, coordinates
```

```
 bookings.aircrafts_data  | 8192 bytes |                    |
(10 rows)
```

Размеры таблиц t1 и t2 различаются, хотя скрипт не рекомендовал перестановку столбцов у таблицы t1.

20) Посмотрим пример, где скрипт даёт точную оценку. Выполните команды:

```
postgres=#
drop table if exists t;
create table t(c1 int4, c2 int4, c3 int4, c4 int4, c5 int4, c6 int4, c7 int8, c8 int8);
DO
$$
BEGIN
FOR i IN 1 .. 100000 LOOP
INSERT INTO t VALUES (1,2,3,4,5,6,7,8);
END LOOP;
END;
$$
;
drop table if exists t1;
create table t1(c1 int8, c2 int4, c3 int4, c4 int4, c5 int8, c6 int4, c7 int4, c8 int4);
DO
$$
BEGIN
FOR i IN 1 .. 100000 LOOP
INSERT INTO t1 VALUES (1,2,3,4,5,6,7,8);
END LOOP;
END;
$$
;
select pg_table_size('t'), pg_table_size('t1'), 100*(pg_table_size('t1')-
pg_table_size('t'))/pg_table_size('t') "%";
DROP TABLE
CREATE TABLE
DO
DROP TABLE
CREATE TABLE
DO
 pg_table_size | pg_table_size |       %
---------------+---------------+----------
       6914048 |       7684096 |       11
(1 row)
```

Экономия места 11% (увеличение размера 10% если в знаменателе вместо таблицы t1 указать t).

```
postgres=# \i p1_alignment_padding.sql
        Table          | Table Size |     Wasted *      | Suggested Columns Reorder
-----------------------+------------+-------------------+---------------------------
 pgbench_accounts      | 13 MB      |                   |
 t1                    | 7512 kB    | ~781 kB (10.40%)  | c2, c3, c4                +
                       |            |                   | c6, c7, c8                +
                       |            |                   | c1, c5
 t                     | 6760 kB    |                   |
```

Скрипт правильно оценил экономию места.

21) В теоретической части были приведены примеры на слайде "Aligning (выравнивание)". Был дан пример для второй каринки:


```
create table t ( a boolean, b int4);
```

```
insert into t values (true, 1);
select t_data, lp_len, t_hoff from heap_page_items(get_raw_page('t','main',0));
       t_data       | lp_len | t_hoff
--------------------+--------+--------
 \x0100000001000000 |     32 |     24
```

Создайте пример для четвертой картинки на слайде:

Первое поле 1-байтовое тип "char", второе поле bigint.
Результат должен быть аналогичный тому, что показан на слайде. Пример:

```
postgres=# drop table if exists t;
DROP TABLE
postgres=# напишите команду создания таблицы
CREATE TABLE
postgres=# insert into t values ('1', 1);
INSERT 0 1
postgres=# select t_data, lp_len, t_hoff from
heap_page_items(get_raw_page('t','main',0));
                t_data                | lp_len | t_hoff
--------------------------------------+--------+--------
 \x3100000000000000010000000000000000 |     40 |     24
(1 row)
```

Неиспользуемых байт 7, что довольно много. Если переставить столбцы местами, то исчезнут ли между полями неиспользуемые байты? Между полями исчезнут, но те же 7 неиспользуемые байты появятся между строками в блоке, так как строки выравниваются по 8 байт.

Часть 3. Содержимое блоков таблицы

1) Создайте таблицу:

```
postgres=#
drop table if exists t;
create table t(s text);
insert into t values ('a');
select * from page_header(get_raw_page('t', 0));
NOTICE:  table "t" does not exist, skipping
DROP TABLE
CREATE TABLE
INSERT 0 1
    lsn     | checksum | flags | lower | upper | special | pagesize | version |prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+---------
 9/5507B2B8 |        0 |     0 |    28 |  8144 |    8176 |     8192 |       5 |        0
(1 row)
```

lower указывает на конец заголовка блока
special на конец области данных, он же начало специальной области, которая используется для поддержки 64-разрядных идентификаторов транзакций
pagesize - размер блока, в СУБД Tantor всегда 8192.
checksum - контрольная сумма блока. Она рассчитывается при сохранении блока в файл. При нахождении блока в буфере контрольная сумма не меняется.

prune_xid - по умолчанию равен 0. xmax наиболее старой неочищенной строки в блоке. Используется как подсказка (it's like a hint-bit, non-WAL-logged hint field that helps determine whether pruning will be useful. It is currently unused in index pages) процессу, который будет искать место в блоке, чтобы попытался очистить блок (First check whether there's any chance there's something to prune).

Наибольший интерес представляют **lower** и **upper**. По этим полям можно определить размер заголовка блока, размер области данных, сколько свободного места в блоке.

2) Чтобы увидеть ненулевое значение в поле контрольной суммы выполните полное вакуумирование таблицы:

```
postgres=# vacuum full t;
select * from page_header(get_raw_page('t', 0));
VACUUM
    lsn     | checksum | flags | lower | upper | special | pagesize | version |prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+---------
 9/BCF15670 |    -7207 |     0 |    28 |  8144 |    8176 |     8192 |       5 |       0
(1 row)
```

3) Посмотрите какие типы данных предпочтительны (preferred) в своей категории (category):

```
postgres=# select distinct typname, typalign, typstorage, typlen from pg_type
where typname not like 'pg_%' and typname not like '\_%' and typtype='b' and
typispreferred = true and typisdefined = true order by typname;
   typname   | typalign | typstorage | typlen
-------------+----------+------------+--------
 bool        | c        | p          |      1
 float8      | d        | p          |      8
 inet        | i        | m          |     -1
 interval    | d        | p          |     16
 oid         | i        | p          |      4
 text        | i        | x          |     -1
 timestamptz | d        | p          |      8
 varbit      | i        | x          |     -1
(8 rows)
```

`typispreferred = true` означает, что тип является предпочитаемым типом в своей категории (typcategory). Выдалось 8 строк, значит категорий типов данных 8. Если убрать это условие, запрос выдаст 62 строки.
`typtype = b` (base) базовый тип данных. Остальные типы создаются отдельно: c (composite) составной тип (образ строки таблицы), d (domain) домена (ограничение), e (enum) перечисление, p (pseudo-type) псевдотип, r (range) диапазонный тип, m (multirange) мультидиапазонный тип. typisdefined = true тип сформирован. false - тип-заготовка и не готов к использованию.

`typalign` - выравнивание. Принимает значения:
`c` (char), 1 байт, то есть без выравнивания
`s` (short), 2 байта
`i` (int), 4 байта
`d` (double) по 8 байт

`typlen` - число байт в которых хранится поле этого типа. Для типов переменной длины (называются `varlena`), `typlen=-1`.

`typstorage` - стратегия хранения столбца по умолчанию. Для типов данных фиксированной ширины (`typlen>0`) единственно возможная стратегия "`p`". Возможные значения:
`p` (plain, простое хранение): поле хранится в блоке таблицы без сжатия
`e` (external, внешнее хранение): значение поля перемещается в TOAST

m (main, в основном слое): поле сжимается и хранится в блоке таблицы
x (extended, расширенное): поле может быть несжато/сжато, храниться в блоке таблицы или значение поля может быть помещено в TOAST.

Часть 4. Выравнивание полей в строках таблиц

1) Создайте таблицы и посмотрите характеристики столбцов таблиц:

```
postgres=#
drop table if exists t1;
drop table if exists t2;
create table t1 (c1 varchar(1), c2 bigserial , c3 date, c4 timestamp);
create table t2 (c1 bigserial , c2 timestamp, c3 date, c4 varchar(1));
SELECT a.attname, t.typname, t.typalign, t.typlen
FROM pg_class c
 JOIN pg_attribute a ON (a.attrelid = c.oid)
 JOIN pg_type t ON (t.oid = a.atttypid)
WHERE c.relname = 't1' AND a.attnum >= 0;
SELECT a.attname, t.typname, t.typalign, t.typlen
FROM pg_class c
 JOIN pg_attribute a ON (a.attrelid = c.oid)
 JOIN pg_type t ON (t.oid = a.atttypid)
WHERE c.relname = 't2' AND a.attnum >= 0;
DROP TABLE
DROP TABLE
CREATE TABLE
CREATE TABLE
 attname |  typname  | typalign | typlen
---------+-----------+----------+--------
 c1      | varchar   | i        |     -1
 c2      | int8      | d        |      8
 c3      | date      | i        |      4
 c4      | timestamp | d        |      8
(4 rows)

 attname |  typname  | typalign | typlen
---------+-----------+----------+--------
 c1      | int8      | d        |      8
 c2      | timestamp | d        |      8
 c3      | date      | i        |      4
 c4      | varchar   | i        |     -1
(4 rows)
```

Последним в таблице t1 идёт столбец фиксированной длинны 8 байт. Это значит, что все предыдущие столбцы фиксированной ширины будут выравниваться по 8 байт. Столбцы переменной ширины varlena (text, bytea, numeric и много других) имеют выравнивание i (реже d). Поля таких типов будут выравниваться и заставлять выравниваться предыдущие столбцы, **если в таких полях хранится больше 126 байт.** Эта фраза сложна для понимания, её можно детализировать:
В таблице t1 все четыре столбца выровнены по 8 байт, в том числе и столбец c1 занимает минимум 8 байт. А вот в таблице t2 столбец c4 не выравнивается и не заставляет выравниваться предыдущие столбцы, если размер поля c4 меньше 126 байт.

2) Посмотрите список типов переменной ширины varlena:

```
postgres=# select distinct typname, typalign, typstorage, typlen from pg_type
where typname not like 'pg_%' and typname not like '\_%' and typtype='b' and
typisdefined = true and typlen= -1 order by typname;
    typname    | typalign | typstorage | typlen
```

```
---------------+----------+------------+--------
 bit           | i        | x          |     -1
 bpchar        | i        | x          |     -1
 bytea         | i        | x          |     -1
 cidr          | i        | m          |     -1
 gtsvector     | i        | p          |     -1
 inet          | i        | m          |     -1
 int2vector    | i        | p          |     -1
 json          | i        | x          |     -1
 jsonb         | i        | x          |     -1
 jsonpath      | i        | x          |     -1
 numeric       | i        | m          |     -1
 oidvector     | i        | p          |     -1
 path          | d        | x          |     -1
 polygon       | d        | x          |     -1
 refcursor     | i        | x          |     -1
 text          | i        | x          |     -1
 tsquery       | i        | p          |     -1
 tsvector      | i        | x          |     -1
 txid_snapshot | d        | x          |     -1
 varbit        | i        | x          |     -1
 varchar       | i        | x          |     -1
 xml           | i        | x          |     -1
(22 rows)
```

Выравнивание существенно ускоряет обработку данных, поэтому оно используется, несмотря на увеличение места хранения. То же самое со структурами в памяти - по возможности используется выравнивание.

3) Вставьте в таблицы t1 и t2 по одной строке:

```
postgres=#
insert into t1 values('A', 1, now(), current_timestamp);
insert into t2 values(1, current_timestamp, now(), 'A');
INSERT 0 1
INSERT 0 1
```

4) Посмотрите какой размер строки показывает функция `pg_column_size(таблица.*)`:

```
postgres=#
select pg_column_size(t1.*) from t1 limit 1;
select pg_column_size(t2.*) from t2 limit 1;
 row_size
----------
       56
(1 row)

 row_size
----------
       46
(1 row)
```

Разница в размере строк **по данным функции** 56-46=10 байт. Реальная разница в числе байт под хранение строки будет другой. Какой? Можно вернуться к 18 пункту предыдущей части практики (где написано, что пункт сложен для понимания) и рассчитать реальную разницу.
Ответ: 46 будет занимать 6*8=48 байт, 56 будет занимать 7*8=56 байт. Реальная разница в размере строки после выравнивания всей строки: 56-48=8 байт.

5) Что находится в этих 46 и 56 байтах? Выполните запросы:

```
postgres=# SELECT lp_off,lp_len,t_hoff,t_data FROM
heap_page_items(get_raw_page('t1','main',0))\gx
-[ RECORD 1 ]-------------------------------------------
```

```
lp_off | 8120
lp_len | 56
t_hoff | 24
t_data | \x0541000000000000010000000000000009b2300000000000015fcf714ccc0200
```

Размер данных строки (t_data) 32 байта.
Размер заголовка строки: 56-32=24 байта.
Все столбцы занимают по 8 байт и будут выравниваться по 8 байт.

```
postgres=# SELECT lp_off,lp_len,t_hoff,t_data FROM
heap_page_items(get_raw_page('t2','main',0))\gx
-[ RECORD 1 ]-----------------------------------------
lp_off | 8128
lp_len | 46
t_hoff | 24
t_data | \x0100000000000000a27fcf714ccc02009b2300000541
```

Размер данных строки (t_data) 22 байта.
Размер заголовка строки: 46-22=24 байта.
Последний столбец varchar не выравнивается и занимает 2 байта. Если длинна поля varchar превысит 126 байт, то поле будет выровнено по 4 байта.
Поля c1 и c2 занимают по 8 байт потому , что у них такой размер.

`lp_len` не показывает полный размер строки, который строка занимает в блоке. В `lp_len` не входит выравнивание всей строки. Это означает, что если `lp_len` не делится на 8, то в конец строки добавятся от 1 до 7 пустых байт, чтобы размер места, занимаемого строкой стал кратен 8 байтам. `lp_len` хранится в заголовке блока.
Во втором запросе `lp_len=46` (не делится на 8). Строка занимает в блоке 48 байт (делится на 8).
Почему `lp_len` не хранит полный размер строки, ведь так было бы удобнее подсчитывать реальный размер строки? **Потому, что `lp_len` используется процессами экземпляра, чтобы определить на каком байте заканчивается последнее поле в строке.** Расширение pageinspect просто выдаёт то, что хранится в блоке.

В СУБД Tantor минимальный размер заголовка строки, так же как и в PostgreSQL 24 байта. Этот размер может увеличиваться сразу на 8 байт за счет размера карты пустых значений. Карта пустых значений (NULL) хранится в заголовке строки. Для каждого столбца используется 1 бит.

В Astralinux PostgeSQL в заголовке строки хранится служебное поле `t_maclabel` **(**а также биты t_infomaskpgac, t_hasmac**) и и минимальный размер заголовка каждой строки 32 байта.** Функция `heap_page_items(...)` показывает поле `t_maclabel`.

Часть 5. Выравнивание строк в блоках таблиц

1) Пересоздайте таблицы:

```
postgres=#
BEGIN;
drop table if exists t1;
drop table if exists t2;
create table t1 (c1 serial, c2 timestamp);
create table t2 (c2 timestamp , c1 serial);
insert into t1 (c2) values(current_timestamp);
insert into t1 (c2) values(current_timestamp);
insert into t2 (c2) values(current_timestamp);
insert into t2 (c2) values(current_timestamp);
COMMIT;
BEGIN
```

```
DROP TABLE
DROP TABLE
CREATE TABLE
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
COMMIT
```

В таблицы вставлено по две строки, чтобы можно было вычислить реальный размер строк с выравниванием всей строки.

2) Выполните запрос к таблице t1:

```
postgres=# SELECT lp, t_ctid, lp_off,lp_len,t_hoff,t_data FROM
heap_page_items(get_raw_page('t1','main',0));
 lp | t_ctid | lp_off | lp_len | t_hoff |                t_data
----+--------+--------+--------+--------+-----------------------------------
  1 | (0,1)  |   8136 |     40 |     24 | \x01000000000000008cb97ce24ecc0200
  2 | (0,2)  |   8096 |     40 |     24 | \x02000000000000008cb97ce24ecc0200
(2 rows)
```

`lp` - (line pointer) номер слота в заголовке блока (размер слота 4 байта), в котором хранится значение `lp_off`.
`t_ctid` - адрес строки. Состоит из двух чисел: порядковый номер блока в слое main, номер слота (`lp`) в заголовке блока.
`lp_off` - адрес начала заголовка строки (смещение в байтах от начала блока).
`lp_len` - длинна строки в байтах без учёта выравнивания всей строки.
`t_hoff` - размер заголовка строки в байтах. Заголовок строки всегда выравнивается по 8 байт (`t_hoff` кратен 8). В примере `t_hoff` имеет минимальный размер 24 байта. В Astralinux PostgreSQL минимальный размер 32 байта. Размер заголовка строки может увеличиваться за счт карты пустых значений в полях строки. Карта пустых значений (NULL) хранится в заголовке строки. Для каждого столбца используется 1 бит.

Размер второй строки: `8136-8096=40` байт.
Размер первой строки точно такой же. Как его вычислить? Адрес специальной области в результате запроса не видна. Для СУБД Tanator SE это 16 байт. Размер блока 8Кб=`8192` байт. Формула: `8192` (размер блока)`-16` (размер специальной области в конце блока)`-8136` (`lp_off` первой строки)`=40` байт.

3) Выполните запрос к таблице t2:

```
postgres=# SELECT lp, t_ctid, lp_off,lp_len,t_hoff,t_data FROM
heap_page_items(get_raw_page('t2','main',0));
 lp | t_ctid | lp_off | lp_len | t_hoff |           t_data
----+--------+--------+--------+--------+----------------------------
  1 | (0,1)  |   8136 |     36 |     24 | \x8cb97ce24ecc020001000000
  2 | (0,2)  |   8096 |     36 |     24 | \x8cb97ce24ecc020002000000
(2 rows)
```

Размер второй строки: `8136-8096=40` байт. Размер, занимаемый строкой такой же, как у таблицы t1. Мы выяснили, с точки зрения места под хранение разницы между таблицами нет.

4) Убедимся, что разницы действительно нет. Вставьте по 600 строк в каждую таблицу и посмотрите максимальный ctid:

```
postgres=#
```

```
DO
$$
BEGIN
FOR i IN 1 .. 600 LOOP
insert into t1 (c2) values(current_timestamp);
insert into t2 (c2) values(current_timestamp);
END LOOP;
END;
$$
;
DO
postgres=# select max(ctid) from t1;
  max
--------
 (3,47)
(1 row)

postgres=# select max(ctid) from t2;
  max
--------
 (3,47)
(1 row)
```

В обеих таблицах максимальный ctid одинаков. Это означает, что в первые два блока поместилось одинаковое число строк.

5) Сколько строк поместилось в первый блок обеих таблиц? Выполните команду:

```
postgres=# SELECT count(lp) FROM heap_page_items(get_raw_page('t1','main',0));
 count
-------
   185
(1 row)
postgres=# SELECT count(lp) FROM heap_page_items(get_raw_page('t1','main',0));
 count
-------
   185
(1 row)
```

В первый блок каждой таблицы поместилось 185 строк.

6) Сколько места занимает заголовок блока?

```
postgres=# select * from page_header(get_raw_page('t2','main',0));
    lsn     | checksum | flags | lower | upper | special | pagesize | version | prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+-----------
 A/3D28EAE0 |        0 |     0 |   764 |   776 |    8176 |     8192 |       5 |         0
(1 row)

postgres=# select * from page_header(get_raw_page('t1','main',0));
    lsn     | checksum | flags | lower | upper | special | pagesize | version | prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+-----------
 A/3D28EA98 |        0 |     0 |   764 |   776 |    8176 |     8192 |       5 |         0
(1 row)
```

В обеих таблицах заголовок блока занимает 764 байта, что равняется 9,3% от размера блока. Между заголовком и областью данных осталось неиспользуемым 776-764=12 байт, что немного. Первый блок (второй блок тоже) таблиц полностью заполнен.

В этой части практики вы изучили функции, которыми можно посмотреть как в реальности располагаются строки в блоках. Проверив блок таблицы, которая занимает много места,

возможно, удастся оптимизировать её хранение. Не только перестановкой строк, а, например, изменением размерности, типов или числа полей.
Эта часть главы полезна тем, кто проектирует или реорганизует схемы хранения данных в PostgreSQL. Это небольшая часть того, что стоит знать при проектировании схем. Дальше в курсе будет рассматриваться хранение пустых значений, вытеснение полей в TOAST, выбор типов данных, хранение записей в индексах.

Часть 6. Хранение пустых (NULL) значений в строках таблиц

Если в строке хотя бы одно из полей пусто (имеет значение NULL), то в заголовке этой строки выделяется место под битовую карту. В битовой карте один бит означает один столбец. Если столбцов тысяча, то в битовой карте будет тысяча бит. Размер заголовка увеличится на 1000 бит и будет после этого выровнен до 8 байт.
Минимальный размер заголовка 24 байта. Хочется определить: сколько столбцов может быть в таблице, чтобы размер заголовка при появлении пустого значения в любом столбце не увеличился до 32 байт.

1) Создайте таблицы с числом столбцов 8,9,24,25 и вставьте в каждую таблицу по две строки. В первой строке все поля непустые, во второй строке хотя бы одно поле пустое:

```
postgres=#
drop table if exists t1;
drop table if exists t2;
drop table if exists t3;
drop table if exists t4;
create table t1(c1 int4, c2 int4, c3 int4, c4 int4, c5 int4, c6 int4, c7 int4,
c8 int4);
create table t2(c1 int4, c2 int4, c3 int4, c4 int4, c5 int4, c6 int4, c7 int4,
c8 int4,c9 int4);
create table t3(c1 int4, c2 int4, c3 int4, c4 int4, c5 int4, c6 int4, c7 int4,
c8 int4, c9 int4, c10 int4, c11 int4, c12 int4, c13 int4, c14 int4, c15 int4,
c16 int4, c17 int4, c18 int4, c19 int4, c20 int4, c21 int4, c22 int4, c23 int4,
c24 int4);
create table t4(c1 int4, c2 int4, c3 int4, c4 int4, c5 int4, c6 int4, c7 int4,
c8 int4, c9 int4, c10 int4, c11 int4, c12 int4, c13 int4, c14 int4, c15 int4,
c16 int4, c17 int4, c18 int4, c19 int4, c20 int4, c21 int4, c22 int4, c23 int4,
c24 int4, c25 int4);
INSERT INTO t1 VALUES (1,2,3,4,5,6,7,8);
INSERT INTO t1 VALUES (1,NULL,3,4,5,6,7,8);
INSERT INTO t2 VALUES (1,2,3,4,5,6,7,8,9);
INSERT INTO t2 VALUES (1,NULL,3,4,5,6,7,8,9);
INSERT INTO t3 VALUES
(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24);
INSERT INTO t3 VALUES
(1,NULL,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24);
INSERT INTO t4 VALUES
(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25);
INSERT INTO t4 VALUES
(1,NULL,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25);

DROP TABLE
DROP TABLE
DROP TABLE
DROP TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
INSERT 0 1
```

```
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
```

2) Выполните запросы, чтобы определить размер заголовка каждой из строк во всех четырёх таблицах:

```
postgres=#
select lp, lp_off, lp_len, t_ctid, t_hoff,t_bits from
heap_page_items(get_raw_page('t1','main',0));
select lp, lp_off, lp_len, t_ctid, t_hoff,t_bits from
heap_page_items(get_raw_page('t2','main',0));
select lp, lp_off, lp_len, t_ctid, t_hoff,t_bits from
heap_page_items(get_raw_page('t3','main',0));
select lp, lp_off, lp_len, t_ctid, t_hoff,t_bits from
heap_page_items(get_raw_page('t4','main',0));
```

```
 lp | lp_off | lp_len | t_ctid | t_hoff |  t_bits
----+--------+--------+--------+--------+----------
  1 |   8120 |     56 | (0,1)  |     24 |
  2 |   8064 |     52 | (0,2)  |     24 | 10111111
(2 rows)

 lp | lp_off | lp_len | t_ctid | t_hoff |      t_bits
----+--------+--------+--------+--------+------------------
  1 |   8112 |     60 | (0,1)  |     24 |
  2 |   8048 |     64 | (0,2)  |     32 | 1011111110000000
(2 rows)

 lp | lp_off | lp_len | t_ctid | t_hoff |          t_bits
----+--------+--------+--------+--------+--------------------------
  1 |   8056 |    120 | (0,1)  |     24 |
  2 |   7928 |    124 | (0,2)  |     32 | 10111111111111111111111
(2 rows)

 lp | lp_off | lp_len | t_ctid | t_hoff |              t_bits
----+--------+--------+--------+--------+----------------------------------
  1 |   8048 |    124 | (0,1)  |     24 |
  2 |   7920 |    128 | (0,2)  |     32 | 10111111111111111111111110000000
(2 rows)
```

Если в таблице до 8 столбцов включительно, то размер заголовка строки 24 байта.
Если в таблице 9 столбцов и больше, то размер заголовка строки в случае если хоть в одном поле присутвтует пустое значение (NULL) резко увеличивается в размере и становится 32 байта.

Битовая карта выравнивается по одному байту. Это означает, что если в таблице до 9 столбцов, то битовая карта занимает 1 байт (8 бит). Если в таблице 9-16 столбца, то битовая карта занимает 2 байта (16 бит). Если 17-24, то 3 байта. Начиная с 25 столбцов - 4 байта и так далее.
Сколько столбцов в таблице должно быть минимально, чтобы заголовок строки из-за битовой карты пустых значений увеличился с 32 байт до 40 байт? 8*8 (64 столбца это 8 байт в карте)+8 (битовая карта на 8 столбцов помещается в заголовке размером 24 байта)+1 (лишний столбец из-за которого размер заголовка строки увеличится на 8 байт)=**73 столбца**.

Не нужно переоценивать увеличение размера заголовка строки. Дело в том, что хранение NULL не занимает ни байта в области данных, то есть высокоэффективно. Например, если в поле

вместо NULL хранить ноль и это поле выравнивается по 8 байт, то использование NULL вместо любого другого значения сэкономит 8 байт. **Это видно по столбцу `lp_len`: 52 меньше, чем 56 в таблице t1.** В других таблицах нужно сделать пустым ещё один столбец, чтобы размер строки стал меньше на 4 байта:

```
postgres=# INSERT INTO t3 VALUES
(1,NULL,3,NULL,5,NULL,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24);
INSERT 0 1
postgres=# select lp, lp_off, lp_len, t_ctid, t_hoff,t_bits from
heap_page_items(get_raw_page('t3','main',0));
 lp | lp_off | lp_len | t_ctid | t_hoff |          t_bits
----+--------+--------+--------+--------+-------------------------
  1 |   8056 |    120 | (0,1)  |     24 |
  2 |   7928 |    124 | (0,2)  |     32 | 101111111111111111111111
  3 |   7808 |    116 | (0,3)  |     32 | 101010111111111111111111
(3 rows)
```

Строка с тремя пустыми полями имеет размер 116 байт, что меньше, чем 120 байт у строки где нет пустых значений. `t_hoff` входит в `lp_len`. Реальный размер строк (с учетом выравнивания всей строки) одинаков: 120 байт. Это легко проверить по столбцу **`lp_off`**: первая строка занимает 8192-8056-16=120 байт; третья строка занимает 7928-7808=120 байт.

Использование NULL неудобно для написания запросов, но при этом в PostgreSQL NULL экономит место под хранение. Индексы типа btree в PostgreSQL индексируют NULL.

Часть 7. Число строк в блоке таблицы

Посмотрим сколько строк может хранится в блоке. Представление сколько строк может быть и обычно бывает полезно для того чтобы представлять себе как организовано хранение строк в таблицах.

1) В зависимости от размера строки, при полном заполнении блока число строк в блоке Tantor SE будет:

```
postgres=# select string_agg(a::text,',') rows from (SELECT distinct
trunc(2038/(2*generate_series(0, 1015)+7)) a order by a desc) a;
                                rows
--------------------------------------------------------------------------
291,226,185,156,135,119,107,97,88,81,75,70,65,61,58,55,52,49,47,45,43,41,39,38,37,35,34,33,32,31,30,29,28,27,26,
25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1
(1 row)
```

В сборках с 32-разрядным счетчиком транзакций специальная область в конце блока размером 16 байт отсутствует. Это повлияет только на три числа: вместо `156,135,119` в таких сборках будет помещаться `157,136,120` строк.

2) Проверьте, сколько строк поместится в блоке у таблицы с одним столбцом типа **`serial`** (**`int4, integer`**):

```
postgres=#
drop table if exists t;
create table t(c serial);
insert into t select * from generate_series(1, 600);
select count(*) from t where (ctid::text::point)[0] = 0;
select lp, lp_off, lp_len, t_ctid, t_hoff, t_data from
heap_page_items(get_raw_page('t','main',0)) where t_ctid::text like '(0,%' order
by lp_off limit 2;
select count(*) from heap_page_items(get_raw_page('t','main',0)) where
t_ctid::text like '(0,%';
select * from page_header(get_raw_page('t','main',0));
```

```
DROP TABLE
CREATE TABLE
INSERT 0 600
 count
-------
   226
(1 row)
 lp  | lp_off | lp_len | t_ctid  | t_hoff |   t_data
-----+--------+--------+---------+--------+------------
 226 |    944 |     28 | (0,226) |     24 | \xe2000000
 225 |    976 |     28 | (0,225) |     24 | \xe1000000
(2 rows)

 count
-------
   226
(1 row)

    lsn     | checksum | flags | lower | upper | special | pagesize | version |prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+---------
 A/55E5F568 |        0 |     0 |   928 |   944 |    8176 |     8192 |       5 |        0
(1 row)
```

В блоке такой таблицы поместится 226 строк. Заменяя определение таблицы можно проверить сколько строк поместится в блоке.

Как получилось 226 байт? Формула для вычисления максимального числа строк в блоке Tantor SE:
8192 (размер блока) = 24 (заголовок блока) + 4*x + 24 (заголовок строки)*x + 8 (длина строки кратна 8)*x + 16 (специальная область).
8152 = 36*x + 16
x = 226

3) В Astralinux PostgreSQL в блоке помещается 185 строк, а не 226 строк. Проверьте это, подсоединившись на порт 5435:

```
postgres@tantor:~$ psql -p 5435
psql (16.2, server 15.6 (Debian 15.6-astra.se2))
Type "help" for help.

postgres=#
drop table if exists t;
create table t(c serial);
insert into t select * from generate_series(1, 600);
select count(*) from t where (ctid::text::point)[0] = 0;
select lp, lp_off, lp_len, t_ctid, t_hoff, t_data from
heap_page_items(get_raw_page('t','main',0)) where t_ctid::text like '(0,%' order
by lp_off limit 2;
select * from page_header(get_raw_page('t','main',0));
DROP TABLE
CREATE TABLE
INSERT 0 600
 count
-------
   185
(1 row)

 lp  | lp_off | lp_len | t_ctid  | t_hoff |   t_data
-----+--------+--------+---------+--------+------------
 185 |    792 |     36 | (0,185) |     32 | \xb9000000
 184 |    832 |     36 | (0,184) |     32 | \xb8000000
(2 rows)

    lsn     | checksum | flags | lower | upper | special | pagesize | version |prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+---------
```

```
 0/8D9C5A8 |         0 |      0 |    764 |    792 |     8192 |      8192 |        4 |         0
(1 row)
```

4) Замените serial на **bigserial** (**int8**) и проверьте сколько строк помещается в блоке:

```
postgres=#
drop table if exists t;
create table t(c bigserial);
insert into t select * from generate_series(1, 600);
select count(*) from t where (ctid::text::point)[0] = 0;
select lp, lp_off, lp_len, t_ctid, t_hoff, t_data from
heap_page_items(get_raw_page('t','main',0)) where t_ctid::text like '(0,%' order
by lp_off limit 2;
select * from page_header(get_raw_page('t','main',0));
DROP TABLE
CREATE TABLE
INSERT 0 600
 count
-------
   226
(1 row)

 lp  | lp_off | lp_len | t_ctid  | t_hoff |       t_data
-----+--------+--------+---------+--------+--------------------
 226 |    944 |     32 | (0,226) |     24 | \xe200000000000000
 225 |    976 |     32 | (0,225) |     24 | \xe100000000000000
(2 rows)

    lsn     | checksum | flags | lower | upper | special | pagesize | version |prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+---------
 A/55E20128 |        0 |     0 |   928 |   944 |    8176 |     8192 |       5 |         0
(1 row)
```

Число строк также 226. Почему? Потому что строка таблицы со столбцом serial выравнивается по 8 байт.
Это не означает, serial не отличается от bigserial. Можно добавить столбцы к таблицам и в части случаев будет разница.
Однако что важно: индексы типа b-tree по столбцам int4 (integer, serial) и int8 (bigserial) всегда имеют одинаковый размер.

5) Число строк при максимальном размере данных в строке:

```
postgres=# SELECT trunc(2038/(2*generate_series+7)) rows , max(generate_series*8)
size FROM generate_series(0, 1015) group by rows order by 1 desc;
 rows | size
------+------
  291 |    0
  226 |    8
  185 |   16
  156 |   24
  135 |   32
  119 |   40
  107 |   48
   97 |   56
   88 |   64
   81 |   72
   75 |   80
   70 |   88
   65 |   96
   61 |  104
   58 |  112
   55 |  120
   52 |  128
   49 |  136
   47 |  144
   45 |  152
   43 |  160
```

```
  41 |  168
  39 |  176
  38 |  184
  37 |  192
  35 |  200
  34 |  208
  33 |  216
  32 |  224
  31 |  232
  30 |  240
  29 |  248
  28 |  256
  27 |  272
  26 |  280
  25 |  296
  24 |  304
  23 |  320
  22 |  336
  21 |  360
  20 |  376
  19 |  400
  18 |  424
  17 |  448
  16 |  480
  15 |  512
  14 |  552
  13 |  592
  12 |  648
  11 |  712
  10 |  784
   9 |  872
   8 |  984
   7 | 1136
   6 | 1328
   5 | 1600
   4 | 2008
   3 | 2688
   2 | 4048
   1 | 8120
(60 rows)
```

6) Проверим значение 185, которое следует за 266:

```
postgres=#
drop table if exists t;
create table t(c bigserial, c1 text default 'a');
insert into t select *, 'a' from generate_series(1, 600);
select count(*) from t where (ctid::text::point)[0] = 0;
select lp, lp_off, lp_len, t_ctid, t_hoff, t_data from
heap_page_items(get_raw_page('t','main',0)) where t_ctid::text like '(0,%' order
by lp_off limit 2;
select count(*) from heap_page_items(get_raw_page('t','main',0)) where
t_ctid::text like '(0,%';
select * from page_header(get_raw_page('t','main',0));
DROP TABLE
CREATE TABLE
INSERT 0 600
 count
-------
   185
(1 row)

 lp  | lp_off | lp_len | t_ctid  | t_hoff |         t_data
-----+--------+--------+---------+--------+-------------------------
 185 |    776 |     34 | (0,185) |     24 | \xb9000000000000000561
 184 |    816 |     34 | (0,184) |     24 | \xb8000000000000000561
(2 rows)

 count
-------
   185
(1 row)

    lsn     | checksum | flags | lower | upper | special | pagesize | version |prune_xid
```

```
------------+----------+-------+-------+-------+---------+----------+---------+---------
 A/55E8F8B0 |        0 |     0 |   764 |   776 |    8176 |     8192 |       5 |       0
(1 row)
```

Для двух столбцов bigserial и столбца переменной ширины, хранящего один символ и выровненного по 4 байта максимальное число строк в блоке 185.
Формула:
8192 = 24 (заголовок блока) + 4*x + 24 (заголовок строки)*x + 16 (длина строки кратна 8)*x + 16 (специальная область)
8152 = 44*x + 12
x = 185

7) Наиболее оптимальный полный размер строки кратен 64 байтам (размер cache line). Число строк и размер области данных строки для Tantor SE:

```
postgres=# SELECT distinct trunc(2038/(16*generate_series+4)) rows,
max(generate_series*64-24) size FROM generate_series(1, 64) group by rows order
by 1 desc;
 rows | size
------+------
  101 |   40
   56 |  104
   39 |  168
   29 |  232
   24 |  296
   20 |  360
   17 |  424
   15 |  488
   13 |  552
   12 |  616
   11 |  680
   10 |  744
    9 |  808
    8 |  936
    7 | 1064
    6 | 1256
    5 | 1576
    4 | 1960
    3 | 2664
    2 | 4008
    1 | 4072
(21 rows)
```

При размере области данных как в столбце size потерь на padding нет и строки выровнены по cache line, то есть обрабатываются с минимальными накладными расходами.

Практика к главе 7

Часть 1. Методы доступа

1) Посмотрите какие имеются методы доступа к данным:

```
postgres=# \dA+
                                 List of access methods
  Name   | Type  |       Handler        |                Description
---------+-------+----------------------+---------------------------------------
 brin    | Index | brinhandler          | block range index (BRIN) access method
 btree   | Index | bthandler            | b-tree index access method
 gin     | Index | ginhandler           | GIN index access method
 gist    | Index | gisthandler          | GiST index access method
 hash    | Index | hashhandler          | hash index access method
 heap    | Table | heap_tableam_handler | heap table access method
 spgist  | Index | spghandler           | SP-GiST index access method
(7 rows)
```

Есть два типа методов (способов) доступа: табличные и индексные.

2) Установите расширения `pg_columnar` и `bloom`:

```
postgres=#
create extension pg_columnar;
create extension bloom;
CREATE EXTENSION
CREATE EXTENSION
```

Эти расширения добавляют методы доступа. Можно добавлять и табличные и индексные методы доступа. Табличные методы доступа определяют способ хранения данных в таблицах.

3) Посмотрите, какие методы доступа добавились:

```
postgres=# \dA+
                                   List of access methods
   Name    | Type  |      Handler         |                Description
-----------+-------+----------------------+-------------------------------
 bloom     | Index | blhandler            | bloom index access method
 brin      | Index | brinhandler          | block range index (BRIN) access method
 btree     | Index | bthandler            | b-tree index access method
 columnar  | Table | columnar.columnar_handler |
 gin       | Index | ginhandler           | GIN index access method
 gist      | Index | gisthandler          | GiST index access method
 hash      | Index | hashhandler          | hash index access method
 heap      | Table | heap_tableam_handler | heap table access method
 spgist    | Index | spghandler           | SP-GiST index access method
(9 rows)

postgres=# select * from pg_am;
  oid   |  amname  |          amhandler          | amtype
--------+----------+-----------------------------+--------
      2 | heap     | heap_tableam_handler        | t
    403 | btree    | bthandler                   | i
    405 | hash     | hashhandler                 | i
    783 | gist     | gisthandler                 | i
   2742 | gin      | ginhandler                  | i
   4000 | spgist   | spghandler                  | i
   3580 | brin     | brinhandler                 | i
```

```
 544775 | columnar | columnar.columnar_handler | t
 544924 | bloom    | blhandler                 | i
(9 rows)
```

4) При создании индекса указывается тип индекса и класс операторов для каждого столбца индекса. Пример:

```
postgres=# drop table if exists t;
NOTICE:  table "t" does not exist, skipping
DROP TABLE
postgres=# create table t (id int8, s text);
CREATE TABLE
postgres=# create index t_idx on t using btree (id int8_ops, s text_pattern_ops);
CREATE INDEX
```

Если не указать тип индекса (индексный метод доступа), то используется `btree`.
Если не указать класс операторов, то используется класс операторов по умолчанию, установленный для типа данных, по которому создаётся индекс.

5) Посмотрите, какие имеются классы операторов для метода btree. В процессе набора команды можно нажать два раза клавишу <TAB> и будет показан список слов, которыми можно продолжить команду:

```
postgres=# \dA<TAB><TAB>
\dA   \dAc  \dAf  \dAo  \dAp
postgres=# \dAc+ btree int<TAB><TAB>
int2             int2vector      int4             int4multirange  int4range
int8             int8multirange  int8range        integer         internal
postgres=# \dAc+ btree int8
                                List of operator classes
  AM   | Input type | Storage type | Operator class | Default? | Operator family |  Owner
-------+------------+--------------+----------------+----------+-----------------+----------
 btree | bigint     |              | int8_ops       | yes      | integer_ops     | postgres
(1 row)
postgres=# \dAc+ btree bigint
                                List of operator classes
  AM   | Input type | Storage type | Operator class | Default? | Operator family |  Owner
-------+------------+--------------+----------------+----------+-----------------+----------
 btree | bigint     |              | int8_ops       | yes      | integer_ops     | postgres
(1 row)
```

Для типа данных `int8 (bigint)` имеется один класс операторов `int8_ops`, входящий в семейство `integer_ops`.
Для int4 (integer) имеется один класс операторов `int4_ops`, входящий в то же семейство.
Семейство зависит от наличия метода доступа и связано с ним. Методы доступа не зависят от наличия семейства. Методы доступа используют не семейства, а классы операторов.

Вместо команд psql можно использовать запросы к таблицам системного каталога. Например, для команды `\dAc+ * bigint` эквивалентом будет запрос:

```
postgres=# SELECT am.amname "AM", format_type(c.opcintype, NULL) "type",
 c.opcname "op_class", c.opcdefault "d", of.opfname "op_family"
FROM pg_opclass c JOIN pg_am am on am.oid = c.opcmethod JOIN pg_opfamily of ON
of.oid = c.opcfamily where format_type(c.opcintype, NULL)='bigint'
ORDER BY 1, 2, 4;
  AM   |  type  |       op_class        | d |        op_family
-------+--------+-----------------------+---+-------------------------
 brin  | bigint | int8_minmax_multi_ops | f | integer_minmax_multi_ops
 brin  | bigint | int8_bloom_ops        | f | integer_bloom_ops
 brin  | bigint | int8_minmax_ops       | t | integer_minmax_ops
 btree | bigint | int8_ops              | t | integer_ops
```

```
 hash   | bigint | int8_ops                | t | integer_ops
(5 rows)
```

6) Посмотрите, какое семейство операторов имеется для целочисленных типов:

```
postgres=# \dAf btree int4
              List of operator families
  AM   | Operator family |     Applicable types
-------+-----------------+--------------------------
 btree | integer_ops     | smallint, integer, bigint
(1 row)

postgres=# \dAf btree int8
              List of operator families
  AM   | Operator family |     Applicable types
-------+-----------------+--------------------------
 btree | integer_ops     | smallint, integer, bigint
(1 row)
```

Для целочисленных типов имеется одно и то же семейство: `integer_ops`
Классы операторов объединяются в семейство, чтобы можно было создавать планы выполнения с выражениями разных (но совместимых, приводимых друг к другу) типов без использования явного приведения типов.

7) Посмотрите (выполнять команды не нужно) следующие примеры, которые показывают, что при использовании типов данных, не включенных в семейство затруднено и требуется явное приведение типов:

Выберем значения столбца `backend_xmin`:

```
postgres=# select backend_xmin from pg_stat_activity where backend_xmin is not null limit
1;
 backend_xmin
--------------
       422122
(1 row)
```

Значение счётчика транзакций похоже на целое число.
Целые числа можно сравнивать и сортировать. Это значение отсортировать нельзя:

```
postgres=# select backend_xmin from pg_stat_activity where backend_xmin is not
null order by 1 limit 1;
ERROR:  could not identify an ordering operator for type xid
LINE 1: ..._activity where backend_xmin is not null order by 1 limit 1;
                                                             ^
HINT:  Use an explicit ordering operator or modify the query.
```

Функция max не работает с этим типом:

```
postgres=# select max(backend_xmin) from pg_stat_activity where backend_xmin is
not null order by 1 limit 1;
ERROR:  function max(xid) does not exist
LINE 1: select max(backend_xmin) from pg_stat_activity where backend...
               ^
HINT:  No function matches the given name and argument types. You might need to
add explicit type casts.
```

Привести к целому числу тип данных xid нельзя, так как нет приведения типов (приведения типов создаются командой CREATE CAST), как и функций приведения:

```
postgres=# select backend_xmin::int8 from pg_stat_activity where backend_xmin is
not null order by 1 limit 1;
ERROR:  cannot cast type xid to bigint
LINE 1: select backend_xmin::int8 from pg_stat_activity where backen...
                           ^

postgres=# select backend_xmin::text from pg_stat_activity where backend_xmin is
not null order by 1 limit 1;
 backend_xmin
--------------
 422122
(1 row)
```

К типу text приведение есть. С типом text работает сортировка.

8) После предыдущего пункта вы могли забыть, какие классы операторов есть для целых чисел и в какое семейство они входят. Можно перечитать 5 и 6 пункты этой части практики. Вы вспомните, что семейства операторов нужны для того, чтобы запросы выполнялись без явного приведения совместимых типов.
Посмотрите, какие функции используются для поддержки метода доступа btree:

```
postgres=# \dAp+ btree integer_ops
                                  List of support functions of operator families
  AM   | Operator family | Registered left type | Registered right type | Number |      Function
-------+-----------------+----------------------+-----------------------+--------+-------------------------
 btree | integer_ops     | bigint               | bigint                |      1 | btint8cmp(bigint,bigint)
 btree | integer_ops     | bigint               | bigint                |      2 | btint8sortsupport(internal)
 btree | integer_ops     | bigint               | bigint                |      3 | in_range(bigint,bigint,
                                                                                    bigint,boolean,boolean)
 btree | integer_ops     | bigint               | bigint                |      4 | btequalimage(oid)
 btree | integer_ops     | integer              | integer               |      1 | btint4cmp(integer,integer)
 btree | integer_ops     | integer              | integer               |      2 | btint4sortsupport(internal)
 btree | integer_ops     | integer              | integer               |      3 | in_range(integer,integer,
                                                                                    integer,boolean,boolean)
 btree | integer_ops     | integer              | integer               |      4 | btequalimage(oid)
 btree | integer_ops     | smallint             | smallint              |      1 | btint2cmp(smallint,smallint)
 btree | integer_ops     | smallint             | smallint              |      2 | btint2sortsupport(internal)
 btree | integer_ops     | smallint             | smallint              |      3 | in_range(smallint,smallint,
                                                                                    smallint,boolean,boolean)
 btree | integer_ops     | smallint             | smallint              |      4 | btequalimage(oid)
 btree | integer_ops     | bigint               | integer               |      1 | btint84cmp(bigint,integer)
 btree | integer_ops     | bigint               | smallint              |      1 | btint82cmp(bigint,smallint)
 btree | integer_ops     | integer              | bigint                |      1 | btint48cmp(integer,bigint)
 btree | integer_ops     | integer              | bigint                |      3 | in_range(integer,integer,
                                                                                    bigint,boolean,boolean)
 btree | integer_ops     | integer              | smallint              |      1 | btint42cmp(integer,smallint)
 btree | integer_ops     | integer              | smallint              |      3 | in_range(integer,integer,
                                                                                    smallint,boolean,boolean)
 btree | integer_ops     | smallint             | bigint                |      1 | btint28cmp(smallint,bigint)
 btree | integer_ops     | smallint             | bigint                |      3 | in_range(smallint,smallint,
                                                                                    bigint,boolean,boolean)
 btree | integer_ops     | smallint             | integer               |      1 | btint24cmp(smallint,integer)
 btree | integer_ops     | smallint             | integer               |      3 | in_range(smallint,smallint,
                                                                                    integer,boolean,boolean)
(22 rows)
```

Функций много, так как есть варианты функций, принимающие разные типы данных: `integer`, `bigint`, `smallint`.

Для индексации методом btree достаточно, чтобы тип данных был сравним. Для этого в классе операторов имелась поддерживающая ("опорная") функция **`Number=1`**, которая могла бы сравнить два значения типа данных для которых создан класс операторов. Функция с таким номером есть и называется **`btint8cmp(..)`**. Результат функции : отрицательное, положительное значение или ноль если значения равны.
Для эффективной сортировки (ORDER BY) желательно чтобы класс операторов имел вторую (**`Number=2`**) функцию быстрой сортировки значений. Функция с таким номером есть и называется **`btint8sortsupport(..)`**

Для возможности использования планировщиком индекса в выражениях "RANGE" оконных функций нужна третья (`Number=3`) функция.
Для поддержки дедупликации нужна четвертая (`Number=4`) функция.

9) Класс операторов связывает операторы, которые будут играть роли (стратегии) в методах которые использует логика индекса для упорядочивания (сравнения, сортировки, измерения расстояний, ассоциаций и т.п.) данных. Посмотрите какие операторы связаны со "стратегиями" методов доступа:

```
postgres=# \dAo btree integer_ops
              List of operators of operator families
  AM   | Operator family |       Operator        | Strategy | Purpose
-------+-----------------+-----------------------+----------+---------
 btree | integer_ops     | <(bigint,bigint)      |        1 | search
 btree | integer_ops     | <=(bigint,bigint)     |        2 | search
 btree | integer_ops     | =(bigint,bigint)      |        3 | search
 btree | integer_ops     | >=(bigint,bigint)     |        4 | search
 btree | integer_ops     | >(bigint,bigint)      |        5 | search
 btree | integer_ops     | <(integer,integer)    |        1 | search
 btree | integer_ops     | <=(integer,integer)   |        2 | search
 btree | integer_ops     | =(integer,integer)    |        3 | search
 btree | integer_ops     | >=(integer,integer)   |        4 | search
 btree | integer_ops     | >(integer,integer)    |        5 | search
 btree | integer_ops     | <(smallint,smallint)  |        1 | search
 btree | integer_ops     | <=(smallint,smallint) |        2 | search
 btree | integer_ops     | =(smallint,smallint)  |        3 | search
 btree | integer_ops     | >=(smallint,smallint) |        4 | search
 btree | integer_ops     | >(smallint,smallint)  |        5 | search
 btree | integer_ops     | <(bigint,integer)     |        1 | search
 btree | integer_ops     | <=(bigint,integer)    |        2 | search
 btree | integer_ops     | =(bigint,integer)     |        3 | search
 btree | integer_ops     | >=(bigint,integer)    |        4 | search
 btree | integer_ops     | >(bigint,integer)     |        5 | search
 btree | integer_ops     | <(bigint,smallint)    |        1 | search
 btree | integer_ops     | <=(bigint,smallint)   |        2 | search
 btree | integer_ops     | =(bigint,smallint)    |        3 | search
 btree | integer_ops     | >=(bigint,smallint)   |        4 | search
 btree | integer_ops     | >(bigint,smallint)    |        5 | search
 btree | integer_ops     | <(integer,bigint)     |        1 | search
 btree | integer_ops     | <=(integer,bigint)    |        2 | search
 btree | integer_ops     | =(integer,bigint)     |        3 | search
 btree | integer_ops     | >=(integer,bigint)    |        4 | search
 btree | integer_ops     | >(integer,bigint)     |        5 | search
 btree | integer_ops     | <(integer,smallint)   |        1 | search
 btree | integer_ops     | <=(integer,smallint)  |        2 | search
 btree | integer_ops     | =(integer,smallint)   |        3 | search
 btree | integer_ops     | >=(integer,smallint)  |        4 | search
 btree | integer_ops     | >(integer,smallint)   |        5 | search
 btree | integer_ops     | <(smallint,bigint)    |        1 | search
 btree | integer_ops     | <=(smallint,bigint)   |        2 | search
 btree | integer_ops     | =(smallint,bigint)    |        3 | search
 btree | integer_ops     | >=(smallint,bigint)   |        4 | search
 btree | integer_ops     | >(smallint,bigint)    |        5 | search
 btree | integer_ops     | <(smallint,integer)   |        1 | search
 btree | integer_ops     | <=(smallint,integer)  |        2 | search
 btree | integer_ops     | =(smallint,integer)   |        3 | search
 btree | integer_ops     | >=(smallint,integer)  |        4 | search
 btree | integer_ops     | >(smallint,integer)   |        5 | search
(45 rows)
```

В классе операторов указаны названия "опорных" (supporting, поддерживающих) функций, которые **если встретятся в запросе, то может быть задействован индексный метод доступа** при поиске (Purpose=search) или сортировке (Purpose=ordering) данных.
Числа в Strategy предопределены в коде, реализующем методы доступа. Например, для btree определены стратегии `1 (<),2 (<=),3 (=),4 (>=),5 (>)`.
Для хэш-индексов одна стратегия с номером `1 (=)`:

```
postgres=# \dAo+ hash integer_ops
                      List of operators of operator families
  AM   | Operator family |       Operator       | Strategy | Purpose | Sort opfamily
-------+-----------------+----------------------+----------+---------+---------------
 hash | integer_ops     | =(bigint,bigint)     |        1 | search  |
 hash | integer_ops     | =(integer,integer)   |        1 | search  |
 hash | integer_ops     | =(smallint,smallint) |        1 | search  |
```

```
 hash | integer_ops      | =(bigint,integer)     |         1 | search  |
 hash | integer_ops      | =(bigint,smallint)    |         1 | search  |
 hash | integer_ops      | =(integer,bigint)     |         1 | search  |
 hash | integer_ops      | =(integer,smallint)   |         1 | search  |
 hash | integer_ops      | =(smallint,bigint)    |         1 | search  |
 hash | integer_ops      | =(smallint,integer)   |         1 | search  |
(9 rows)
```

Хэш индексы могут использоваться только с запросами на точное совпадение: равенство (=).
В куре нет заданий типа "давайте посмотрим всё, что только есть". Наоборот, в курсе не изучаются индексы gist, gin, brin чтобы не было лишней информации.
Приводятся примеры для одного наиболее часто используемого типа индексов: btree. Поняв как функционирует индекс btree можно по аналогии разобраться с тем, как работают все остальные типы индексов. Списки стратегий для методов доступа есть в документации https://docs.tantorlabs.ru/tdb/ru/16_4/se/xindex.html
Мы подробно останавливаемся на операторах, так как это позволит понять, какие типы индексов можно создавать для обслуживания запросов. Если в запросе есть оператор, то смогут использоваться индексы, у которых этот оператор есть в списке стратегий этого типа индексов и типа данных.

Часть 2. Использование индексов ограничениями целостности

1) Создайте таблицу с первичным ключом:

```
postgres=# drop table if exists t3;
DROP TABLE
postgres=# create table t3 (n int4 primary key, m int4);
CREATE TABLE
```

При создании таблицы был создан индекс с названием `t3_pkey` для поддержки первичного ключа:

```
postgres=# \d+ t3
                                        Table "public.t3"
 Column |  Type   | Collation | Nullable | Default | Storage |Compression|Stats target|
--------+---------+-----------+----------+---------+---------+-----------+-------------
 n      | integer |           | not null |         | plain   |           |
 m      | integer |           |          |         | plain   |           |
Indexes:
    "t3_pkey" PRIMARY KEY, btree (n)
Access method: heap
```

Табличный метод доступа к таблице: **heap**.

2) Создайте составной индекс по столбцам **m,n**:

```
postgres=# create unique index concurrently t3_pkey1 on t3 (m,n);
CREATE INDEX
```

Индекс может создаваться с опцией CONCURRENTLY. Эта опция устанавливает блокировку SHARE UPDATE EXCLUSIVE на таблицу на время своей работы (то есть на время создания индекса). Опция позволяет выполняться командам SELECT, WITH, INSERT, UPDATE, DELETE, MERGE и позволяет использовать быстрый путь блокирования объектов.
Создание индекса может занимать долго времени. Без CONCURRENTLY таблица сканируется один раз, с CONCURRENTLY таблица сканируется два раза и используются три транзакции, чтобы уменьшить длительность удержания горизонта базы данных.
Автовакуум несовместим с созданием, удалением, пересозданием индексов независимо от того используется CONCURRENTLY или не используется. Автовакуум попускает таблицы, если не может немедленно получить блокировку.
Несколько индексов с опцией CONCURRENTLY на одну и ту же таблицу не могут одновременно создаваться. Несколько индексов без опции CONCURRENTLY могут одновременно создаваться

на одну и ту же таблицу и на разные таблицы. Команды создания индексов можно запустить в разных сессиях.

3) Замените ограничение целостности одной командой:

```
postgres=# ALTER TABLE t3 DROP CONSTRAINT t3_pkey, ADD CONSTRAINT t3_pkey PRIMARY
KEY USING INDEX t3_pkey1;
NOTICE:  ALTER TABLE / ADD CONSTRAINT USING INDEX will rename index "t3_pkey1" to
"t3_pkey"
ALTER TABLE

postgres=# \d+ t3
                                          Table "public.t3"
 Column |  Type   | Collation | Nullable | Default | Storage |Compression|Stats target|
--------+---------+-----------+----------+---------+---------+-----------+-------------
 n      | integer |           | not null |         | plain   |           |
 m      | integer |           | not null |         | plain   |           |
Indexes:
    "t3_pkey" PRIMARY KEY, btree (m, n)
Access method: heap
```

Сравните с результатом вывода этой же команды, выполненной ранее. На столбец m было добавлено ограничение целостности `not null` и для первичного ключа стал использоваться новый индекс. Старый индекс по столбцу n был удалён и его место освобождено. Индекс **t3_pkey1** был переименован в индекс `t3_pkey`.
Время выполнения команды складывается из:
a) ожидания получения **монопольной** блокировки на таблицу
b) ожидания освобождения (буфера будут добавлены в список свободных) **всех** буферов в кэше буферов, которые использовались блоками удаляемого индекса. При использовании кэша буферов неоправданно большого размера поиск каждого блока в кэше буферов может занять долгое время. Неоправданное увеличение кэша буферов провоцируется увеличением числа hash bucket slot в Shared Buffer Lookup Table в некоторых форках PostgreSQL. СУБД Tantor не увеличивает это число (NUM_BUFFER_PARTITIONS).

4) Создайте индекс на столбец n:

```
postgres=# create index concurrently t3_pkey1 on t3 (n);
CREATE INDEX

postgres=# \d t3
                 Table "public.t3"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 n      | integer |           | not null |
 m      | integer |           | not null |
Indexes:
    "t3_pkey" PRIMARY KEY, btree (m, n)
    "t3_pkey1" btree (n)
```

Индекс создан. На таблице есть два индекса. Оба индекса будут одновременно обновляться, если менять строки в таблице. Дополнительные индексы снижают производительность изменения строк в таблицах. Зачем тогда создаются дополнительные индексы?
Дополнительные индексы создаются, если время выполнения команд, использующих дополнительные индексы, существенно меньше, чем команд без использования этих дополнительных индексов. Меньшее время коррелирует со стоимостью плана выполнения и ресурсами процессоров и ввода-вывода.

5) Попробуйте заменить ограничение целостности одной командой:

```
postgres=# alter table t3 DROP CONSTRAINT t3_pkey, ADD CONSTRAINT t3_pkey PRIMARY
KEY USING INDEX t3_pkey1;
ERROR:  "t3_pkey1" is not a unique index
LINE 1: ALTER TABLE t3 DROP CONSTRAINT t3_pkey, ADD CONSTRAINT t3_pk...
                                                ^
DETAIL:  Cannot create a primary key or unique constraint using such an index.
```

Созданный индекс не может использоваться ограничениями целостности. Почему? Потому, что индекс неуникальный. Неуникальные индексы не могут использоваться ограничениями целостности.

6) Попробуйте добавить ограничение целостности в состоянии NOT VALID:

```
postgres=# alter table t3 ADD CONSTRAINT t3_pkey PRIMARY KEY USING INDEX t3_pkey1
NOT VALID;
ERROR:  PRIMARY KEY constraints cannot be marked NOT VALID
```

В PostgreSQL у `PRIMARY KEY` нет состояния `NOT VALID`. В СУБД других производителей ограничения целостности могут иметь состояние `NOT VALID`.

7) Если индекс не пригоден для использования ограничениями целостности и не ускоряет выполнение запросов, его стоит удалить. Удалите индекс:

```
postgres=# drop index t3_pkey1;
DROP INDEX
```

8) Попробуйте удалить индекс, использующийся ограничением целостности:

```
postgres=# drop index t3_pkey;
ERROR: cannot drop index t3_pkey because constraint t3_pkey on table t3 requires it
HINT:  You can drop constraint t3_pkey on table t3 instead.
```

Индекс, использующийся ограничением целостности нельзя удалить. Можно удалить ограничение целостности.

9) Попробуйте создайте внешний ключ командой:

```
postgres=# alter table t3 add constraint fk foreign key (m) references t3(n) not valid;
ERROR:  there is no unique constraint matching given keys for referenced table "t3"
```

Внешний ключ может быть создан только на ограничения целостности PRIMARY KEY или UNIQUE.

10) Создайте внешний ключ командой:

```
postgres=# alter table t3 add constraint fk foreign key (n,m) references t3(m,n) not
valid;
ALTER TABLE
postgres=# \d+ t3
                                        Table "public.t3"
 Column |  Type   |Collation| Nullable |Default| Storage | Compression |Stats target|
--------+---------+---------+----------+-------+---------+-------------+------------
 n      | integer |         | not null |       | plain   |             |            |
 m      | integer |         | not null |       | plain   |             |            |
Indexes:
    "t3_pkey" PRIMARY KEY, btree (m, n)
Foreign-key constraints:
    "fk" FOREIGN KEY (n, m) REFERENCES t3(m, n) NOT VALID
Referenced by:
    TABLE "t3" CONSTRAINT "fk" FOREIGN KEY (n, m) REFERENCES t3(m, n) NOT VALID
Access method: heap
```

Внешний ключ создан без проверки существующих строк.
Индекс на столбцы внешнего ключа отсутствует.

11) При этом внешний ключ работает и выполняют проверку при вставке, удалении, изменении строк в таблице. Выполните вставку в таблицу:

```
postgres=# insert into t3 values (2,3);
ERROR:  insert or update on table "t3" violates foreign key constraint "fk"
DETAIL:  Key (n, m)=(2, 3) is not present in table "t3".
postgres=# insert into t3 values (1,1);
INSERT 0 1
```

Вставка строки, нарушающая внешний ключ не может быть выполнена.
Если после внесения изменений не нарушаются ограничения целостности, то команда выполняется.

12) Выполните проверку строк на соответствие ограничению целостности:

```
postgres=# alter table t3 validate constraint fk;
ALTER TABLE
```

При валидации строк запрашивается блокировка ShareRowExclusive на таблицы:
a) на которой создан FOREIGN KEY (дочерняя таблица)
b) на родительскую таблицу, в которой есть PRIMARY KEY или UNIQUE, на который ссылается FOREIGN KEY.
В примере таблица одна и та же.
Проверка может быть долгой и зависит от числа строк в дочерней таблице. На время проверки удерживается блокировка ShareRowExclusive.

Если используются секционированные таблицы, то статус проверки ограничения целостности может использоваться планировщиком для исключения секций из сканирования.

Индекс на столбцы внешнего ключа при проверке не был создан.
Индекс на FK создают если:
1) в мастер-таблице (там где PK) часто обновляется значение столбца PK или удаляются строки. Эти действия нежелательны и при проектировании приложений их избегают.
2) когда используются соединения таблиц связанных PK-FK. Это используется очень часто, та как для этого FK и нужен: определяет связь (join) между таблицами.

Часть 3. Характеристики btree индексов

1) Создайте таблицу:

```
postgres=#
drop table if exists t;
create table t (id int8, s text storage plain);
create index t_idx on t (s text_ops) include (id);
insert into t values (1, repeat('a',2700));
DROP TABLE
CREATE TABLE
CREATE INDEX
```

2) Попробуйте вставить строку с полем с 2700 символов в таблицу:

```
postgres=# insert into t values (1, repeat('a',2700));
ERROR:  index row size 2720 exceeds btree version 4 maximum 2704 for index "t_idx"
DETAIL:  Index row references tuple (0,1) in relation "t".
HINT:  Values larger than 1/3 of a buffer page cannot be indexed.
Consider a function index of an MD5 hash of the value, or use full text indexing.
```

Максимальный размер индексной записи 2704 байта, то есть примерно треть размера блока без заголовка блока: 2704*3=8112 байт.
Значения, больше чем 1/3 блока не могут индексироваться.

Для вычисления размера строки в таблице можно использовать функцию:

```
postgres=# select pg_column_size(row(1::int4, repeat('a',2700))),
pg_column_size(row(repeat('a',2700)));
 pg_column_size | pg_column_size
----------------+----------------
           2732 |           2728
(1 row)
```

Функций для вычисления размера записи в блоке индекса нет.

3) Посмотрите, сколько записей помещается в промежуточный блок индекса:

```
postgres=# drop table if exists t3;
create table t3 (id bigserial primary key, s int4) with (autovacuum_enabled=off);
insert into t3 (s) select * from generate_series(1, 1000000);
select pg_indexes_size('t3');
select pg_relation_size('t3_pkey', 'main');
select * from bt_page_stats('t3_pkey',3);
DROP TABLE
CREATE TABLE
INSERT 0 1000000
 pg_indexes_size
-----------------
        22487040
(1 row)
 pg_relation_size
------------------
        22487040
(1 row)
 blkno | type | live_items | dead_items | avg_item_size | free_size | btpo_prev | btpo_next | btpo_level |
-------+------+------------+------------+---------------+-----------+-----------+-----------+------------+
     3 | i    |        286 |          0 |            15 |      2436 |         0 |       411 |          1 |
(1 row)
```

Число записей в промежуточном (внутреннем) блоке индекса 286.
Процент заполнения промежуточных блоков индекса btree 70% и не меняется.
Размер основного слоя индекса `22487040` байт.
В таблицу вставляется большое число строк, чтобы в индексе были уровни: листовой, промежуточный и корневой.

4) По умолчанию процент заполнения листовых блоков 90% и может меняться. Посмотрите сколько записей в листовом блоке индекса:

```
postgres=# select * from bt_page_stats('t3_pkey',274);
 blkno | type | live_items | dead_items | avg_item_size | free_size | btpo_prev | btpo_next | btpo_level |
-------+------+------------+------------+---------------+-----------+-----------+-----------+------------+
   274 | l    |        367 |          0 |            16 |       808 |       273 |       275 |          0 |
(1 row)
```

При заполнении (fillfactor=90) на 90% число записей в блоке индекса 286*90%/70%=367.
Во всех листовых блоках свободно 808 байт, кроме самого правого блока. Проиндексированный столбец заполняется монотонно возрастающей последовательностью. Новые значения добавляются в самый правый листовой блок. У самого правого блока на каждом уровне **`btpo_next=0`**. У листовых блоков `type=l`, также нумерация уровней начинается с листового уровня и поэтому у листовых блоков `btpo_level=0`. Данные по листовому блоку:

```
postgres=# select * from bt_page_stats('t3_pkey',275);
 blkno | type | live_items | dead_items | avg_item_size | free_size | btpo_prev | btpo_next | btpo_level |
-------+------+------------+------------+---------------+-----------+-----------+-----------+------------+
   275 | l    |         82 |          0 |            16 |      6508 |       274 |         0 |          0 |
```

В правом листовом блоке 82 записи, в блоке свободно 6508 байт.

5) Посмотрите, сколько записей поместится в листовой блок, если установить **fillfactor=100**:

```
postgres=# drop table if exists t3;
create table t3 (id bigserial primary key, s int4) with (autovacuum_enabled=off,
fillfactor=100);
alter index t3_pkey set (fillfactor=100);
insert into t3 (s) select * from generate_series(1, 1000000);
select pg_indexes_size('t3');
select pg_relation_size('t3_pkey', 'main');
select * from bt_page_stats('t3_pkey',3);
select * from bt_page_stats('t3_pkey',4);
 pg_indexes_size
-----------------
        20275200
(1 row)
 pg_relation_size
------------------
        20275200
(1 row)
 blkno | type | live_items | dead_items | avg_item_size | free_size | btpo_prev | btpo_next | btpo_level
-------+------+------------+------------+---------------+-----------+-----------+-----------+-----------+
     3 | i    |        286 |          0 |            15 |      2436 |         0 |       411 |          1 |
(1 row)

 blkno | type | live_items | dead_items | avg_item_size | free_size | btpo_prev | btpo_next | btpo_level |
-------+------+------------+------------+---------------+-----------+-----------+-----------+------------+
     4 | l    |        407 |          0 |            16 |         8 |         2 |         5 |          0 |
(1 row)
```

Число записей в промежуточных блоках индекса не поменялось и осталось 286, так как процент заполнения промежуточных блоков индекса btree 70% и не меняется. 30% будут использоваться в случае, если листовые блоки будут делиться. При вставке строк в таблицу t3 (заполнение проиндексированного столбца монотонно возрастающей последовательностью) делятся только правые блоки листового и промежуточных уровней.
Число записей, поместившихся в листовые блоки увеличилось на ~11% с 367 до 407, что соответствует увеличению **fillfactor** с 90% до 100%.
Размер файла индекса уменьшился на ~10% с 22487040 до 20275200 байт, что соответствует увеличению **fillfactor** с 90% до 100%.

Длинна записи в листовом блоке 16 байт.

6) Посмотрите данные из заголовка полностью заполненного листового блока:

```
postgres=# select * from page_header(get_raw_page('t3_pkey', 4));
    lsn      | checksum | flags | lower | upper | special | pagesize | version | prune_xid
-------------+----------+-------+-------+-------+---------+----------+---------+----------
 B/3DD2D598 |        0 |     0 |  1652 |  1664 |    8176 |     8192 |       5 |         0
(1 row)
```

Область данных занимает 407*16 = 6152 байт. 8192-6152=1680 это накладные расходы. Основная часть накладных расходов это переменная часть заголовка блока: 4 байта на индексную запись. 407*4 байта (размер слота в заголовке блока)=1628 байт. 1680-1628=52 байт, которые используются фиксированной частью заголовка блока 24 байта, незанятым местом 1664-1652=12 байт, специальной областью размером 8192-8176=16 байт.

7) Посмотрите, сколько блоков разных типов есть в индексе:

```
postgres=# select type, count(*) from bt_multi_page_stats('t3_pkey',1,-1) group
by type order by 2;
 type | count
------+-------
 r    |     1
 i    |     9
 l    |  2464
(3 rows)
```

99,6%, то есть подавляющее большинство блоков - листовые.
Процент листовых блоков уменьшится, если длинна проиндексированных полей будет больше. В таком случае в листовом и промежуточных блоках будет меньше записей и число уровней индекса возрастёт. Дедупликация увеличит число записей в листовых блоках индекса.

8) Выполните удобный запрос, который выводит размеры индексов и команду создания индексов:

```
postgres=# select i.relname "table", indexrelname "index",
pg_INDEXES_size(relid) "indexes_size",
pg_RELATION_size(relid) "table_size ",
pg_TOTAL_RELATION_size(relid) "total",
pg_RELATION_size(indexrelid) "index_size",
reltuples::bigint "rows",
ii.indexdef ddl
from pg_stat_all_indexes i join pg_class c on (i.relid = c.oid)
join pg_indexes ii on (i.indexrelname = ii.indexname)
where i.schemaname not like 'pg_%'  -- не выводить служебные объекты
order by pg_INDEXES_size(relid) desc, pg_RELATION_size(indexrelid) desc limit 1;
-[ RECORD 1 ]+-----------------------------------------------------------
table        | t3
index        | t3_pkey
indexes_size | 20275200
table_size   | 44285952
total        | 64593920
index_size   | 20275200
rows         | -1
ddl          | CREATE UNIQUE INDEX t3_pkey ON public.t3 USING btree (id) WITH (fillfactor='100')
```

Почему число строк в таблице `rows=-1`?
Потому, что не собрана статистика.

9) Соберите статистику по таблице t3:

```
postgres=# analyze t3;
ANALYZE
```

10) Повторите запрос:

```
postgres=# select i.relname "table", indexrelname "index",
pg_INDEXES_size(relid) "indexes_size",
pg_RELATION_size(relid) "table_size ",
pg_TOTAL_RELATION_size(relid) "total",
pg_RELATION_size(indexrelid) "index_size",
reltuples::bigint "rows",
ii.indexdef ddl
from pg_stat_all_indexes i join pg_class c on (i.relid = c.oid)
join pg_indexes ii on (i.indexrelname = ii.indexname)
where i.schemaname not like 'pg_%'  -- не выводить служебные объекты
order by pg_INDEXES_size(relid) desc, pg_RELATION_size(indexrelid) desc limit 1
\gx
-[ RECORD 1 ]+-----------------------------------------------------------
```

```
table         | t3
index         | t3_pkey
indexes_size  | 20275200
table_size    | 44285952
total         | 64593920
index_size    | 20275200
rows          | 1000000
ddl           | CREATE UNIQUE INDEX t3_pkey ON public.t3 USING btree (id) WITH
                (fillfactor='100')
```

Сейчас число строк в таблице `rows=1000000`.
Если из запроса убрать `limit 1`, то запрос удобен для быстрой оценки размеров таблиц и индексов; узнать, на какие индексы обратить внимание. Например, на индексы большого размера, на таблицы с большим числом индексов по одной таблице.

Часть 4. Навигация по структуре btree индексов

1) В расширении pageinspect есть функции для просмотра структуры блоков индексов. Функции для индексов типа btree имеют префикс `bt_` . Число уровней в дереве индекса типа btree отсутствует в таблицах статистики. Число уровней хранится в блоке метаданных и его можно посмотреть функцией **`bt_metap`**. Посмотрите метаданные об индексе `t3_pkey`,который был создан в предыдущей части практики:

```
postgres=# select * from bt_metap('t3_pkey');
 magic  | version | root | level | fastroot | fastlevel |last_cleanup_|last_cleanup|allequalimage
        |         |      |       |          |           |num_delpages | _num_tuples|
--------+---------+------+-------+----------+-----------+-------------+------------+-------------
 340322 |       4 |  412 |     2 |      412 |         2 |           0 |         -1 | t
(1 row)
```

В результатах функции интерес представляют только **`level`** и **`root`**.
Число уровней **`level=2`**. Нумерация начинается с нуля. Дерево индекса растёт снизу вверх и нуль соотвествует листовым блокам.
Поля magic и version используются для быстрой проверки того, что объект является индексом btree поддерживаемой версии. "Магическое" число для индексов типа btree равно 340322 (0x0531162).
Начиная с PostgreSQL версии 12 используется 4 версия индексов. Более старая версия индекса может встретиться, если СУБД обновлялась со старых версий. Индексы старых версий стоит перестроить, иначе новшества, появившиеся в новых версиях, не будут использоваться.
Корневой блок индекса **`root=412`**. Число - порядковый номер блока с начала первого файла слоя данных индекса.
fastroot и fastlevel используются для неважной оптимизации поиска по индексу. На эти столбцы можно не обращать внимание. Если в таблице удалить все строки, то число уровней индекса не уменьшается. При этом у корневого блока останется один наследник. В таком случае fastroot станет блок, у которого несколько наследников и с которого можно будет начинать поиск. В примере fastroot указывает на root.

2) Функция `bt_page_stats` необходима для навигации по структуре индекса и выдаёт одну строку для каждого блока индекса. Посмотрите данные по корневому блоку индекса `t3_pkey`:

```
postgres=# select * from bt_page_stats('t3_pkey',412);
blkno|type|live_items|dead_items|avg_item_size|free_size| btpo_prev | btpo_next | btpo_level | btpo_flags
-----+----+----------+----------+-------------+---------+-----------+-----------+------------+-----------
 412 | r  |        9 |        0 |          15 |    7976 |         0 |         0 |          2 |         2
(1 row)
```

Для навигации используются поля `btpo_prev` и `btpo_next`. В этих полях указаны номера блоков левее и правее на том же уровне. `btpo_prev=0` означает, что блок самый левый, `btpo_next=0` означает, что блок самый правый на своём уровне. Корневой блок единственный на своём уровне, поэтому значения нули.

Тип блока указан в поле `type`. Значения в этом столбце: r - корневой (root); i - внутренний (internal); l - листовой (list), e - (ignored), d - удалёный листовой (deleted leaf), D - удалённый внутренний (deleted internal).
`avg_item_size` показывает вычисленное значение среднего размера индексной записи в этом блоке. Записи выравниваются по 8 байт.
`live_items` - сколько записей есть в этом блоке.

3) Для просмотра индексных записей используется функция `bt_page_items`. Посмотрите индексные записи в корневом блоке индекса `t3_pkey`:

```
postgres=# select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, data from
bt_page_items('t3_pkey', 412) order by 1;
 itemoffset |   ctid   | itemlen | nulls | vars | dead | htid | tids |          data
------------+----------+---------+-------+------+------+------+------+------------------------
          1 | (3,0)    |       8 | f     | f    |      |      |      |
          2 | (411,1)  |      16 | f     | f    |      |      |      | ff c3 01 00 00 00 00 00
          3 | (698,1)  |      16 | f     | f    |      |      |      | fd 87 03 00 00 00 00 00
          4 | (984,1)  |      16 | f     | f    |      |      |      | fb 4b 05 00 00 00 00 00
          5 | (1270,1) |      16 | f     | f    |      |      |      | f9 0f 07 00 00 00 00 00
          6 | (1556,1) |      16 | f     | f    |      |      |      | f7 d3 08 00 00 00 00 00
          7 | (1842,1) |      16 | f     | f    |      |      |      | f5 97 0a 00 00 00 00 00
          8 | (2128,1) |      16 | f     | f    |      |      |      | f3 5b 0c 00 00 00 00 00
          9 | (2414,1) |      16 | f     | f    |      |      |      | f1 1f 0e 00 00 00 00 00
(9 rows)
```

В корневом блоке 9 записей, которые указывают на блоки индекса. В столбце `ctid` могут храниться ссылки на блоки индекса или строки таблицы. При хранении ссылок на блоки индекса идентификатор строки не играет роли: в приведённом примере идентификатор строки имеет значения 0 или 1. Если 1, то в поле data хранится минимальное значение, которое присутствует в дочернем листовом блоке. Если ноль, то поле `data` пусто (трактуется как "минус бесконечность", то есть граница неизвестна) и эта ссылка ведёт на самый левый дочерний блок.
Так как блок корневой, то в первой записи (`itemoffset=1`) не хранится HighKey. HighKey не хранится в самых правых блоках каждого уровня, так как нет смысла обозначать правую границу - блок и так самый правый. HighKey используется для проверки того не нужно ли считывать блок правее.
Поле `data` текущей и следующей (`itemoffset+1`) записи задаёт диапазон, в который должно попасть значение, по которому выполняется поиск в индексе.

Индексные записи хранятся упорядоченно, порядок выдаётся в столбце `itemoffset`. Первая строка `itemoffset=1`, `ctid=(3,0)` указывает на блок нижнего уровня, который будет самым левым на своём уровне. Последняя строка `itemoffset=9`, `ctid=(2414,1)` указывает на блок нижнего уровня, который будет самым правым на своём уровне.
В индексе используются оптимизация suffix truncation и усечение проиндексированных столбцов в поле data её следствие. Из-за этой оптимизации индекс btree используемый в PostgreSQL можно называть "Simple Prefix B-Tree". Простое (Simple) потому, что усекаются целые поля (whole "attribute" truncation). Для индекса по одному столбцу остаётся пустота, трактуемая как минус бесконечность.
Длинна первой записи за счёт отсутствия значения в поле `data` 8 байт: `itemlen=8`.

4) Посмотрите, что выдаёт функция **`bt_page_stats`** по блокам с `itemoffset=1,9,2`:

```
postgres=# select blkno, type, live_items live, dead_items dead, avg_item_size
size, free_size free, btpo_prev, btpo_next, btpo_level l, btpo_flags f from
bt_page_stats('t3_pkey', 3);
 blkno | type | live | dead | size | free | btpo_prev | btpo_next | l | f
-------+------+------+------+------+------+-----------+-----------+---+---
     3 | i    |  286 |    0 |   15 | 2436 |         0 |       411 | 1 | 0
(1 row)
```

3 блок самый левый на своём уровне, на это указывает `btpo_prev=0`.

В 3 блоке 286 записей. Блок наполнен на 70%, что является значением процента заполнения по умолчанию для промежуточных блоков.

```
postgres=# select blkno, type, live_items live, dead_items dead, avg_item_size
size, free_size free, btpo_prev, btpo_next, btpo_level l, btpo_flags f from
bt_page_stats('t3_pkey', 2414);
 blkno | type | live | dead | size | free | btpo_prev | btpo_next | l | f
-------+------+------+------+------+------+-----------+-----------+---+---
  2414 | i    |  184 |    0 |   15 | 4476 |      2128 |         0 | 1 | 0
(1 row)
```

2414 блок самый правый на своём уровне, на это указывает btpo_next=0.
В 2414 блоке 184 записи. Это самый правый блок своего уровня, он заполнен не на 70%, а меньше. Это произошло из-за того, что при заполнении монотонно возрастающей последовательностью, вставки выполняются в правый листовой блок и именно он делится. Вставка записи со ссылкой на новый листовой блок выполняется в самый правый блок вышестоящего уровня. Делятся самые правые блоки каждого уровня. После деления записи перераспределяются между двумя блоками. В блоке "левее" остаётся 70% записей для промежуточных блоков. При делении листового блока, в блоке "левее" остаётся fillfactor записей. Оставшиеся записи как у промежуточного, так и у листового блока остаются в "правом" блоке и их меньше, чем в блоке "левее" от него. Именно поэтому в "правом" 2414 блоке 184 записи, что меньше, чем 286.

```
postgres=# select blkno, type, live_items live, dead_items dead, avg_item_size
size, free_size free, btpo_prev, btpo_next, btpo_level l, btpo_flags f from
bt_page_stats('t3_pkey', 411);
 blkno | type | live | dead | size | free | btpo_prev | btpo_next | l | f
-------+------+------+------+------+------+-----------+-----------+---+---
   411 | i    |  286 |    0 |   15 | 2436 |         3 |       698 | 1 | 0
(1 row)
```

411 блок стоит справа от блока 3, на это указывает btpo_prev=3.
Значения btpo_prev и btpo_next соответствуют порядку следования записей столбца itemoffset. вышестоящего блока. В примере btpo_next=698, что соответствует записи с itemoffset=3 в вышестоящем блоке.
Во всех промежуточных блоках, кроме "правых" число записей (286) и свободное место (free) соответствует 70% заполнению потому, что вставки были в правые блоки. В результате деления правых блоков (когда в них не оставалось места для вставки) левые блоки заполнялись на 70% (промежуточные) или до fillfactor (листовые).

5) Блок 2414 самый правый на промежуточном уровне. В блоке 184 записи. Посмотрите первые и последние записи:

```
postgres=# select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, data
from bt_page_items('t3_pkey', 2414) order by 1 limit 2;
 itemoffset |   ctid   | itemlen | nulls | vars | dead | htid | tids |          data
------------+----------+---------+-------+------+------+------+------+-------------------------
          1 | (2290,0) |       8 | f     | f    |      |      |      |
          2 | (2291,1) |      16 | f     | f    |      |      |      | 87 21 0e 00 00 00 00 00
(2 rows)
postgres=# select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, data
from bt_page_items('t3_pkey', 2414) order by 1 desc limit 2;
 itemoffset |   ctid   | itemlen | nulls | vars | dead | htid | tids |          data
------------+----------+---------+-------+------+------+------+------+-------------------------
        184 | (2474,1) |      16 | f     | f    |      |      |      | 2b 42 0f 00 00 00 00 00
        183 | (2473,1) |      16 | f     | f    |      |      |      | 95 40 0f 00 00 00 00 00
(2 rows)
```

В блоке три уровня. Какой номер у самого правого листового блока? 2474.

6) Посмотрите статистику и содержимое самого правого листового блока:

```
postgres=# select blkno, type, live_items live, dead_items dead, avg_item_size size, free_size free, btpo_prev, btpo_next, btpo_level l, btpo_flags f from bt_page_stats('t3_pkey',2474);
 blkno | type | live | dead | size | free | btpo_prev | btpo_next | l | f
-------+------+------+------+------+------+-----------+-----------+---+---
  2474 | l    |   22 |    0 |   16 | 7708 |      2473 |         0 | 0 | 1
(1 row)
```

В статистике `type=l`, это означает, что блок листовой. `btpo_next=0` означает, что блок самый правый.

```
postgres=# select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, data from bt_page_items('t3_pkey', 2474);
 itemoffset |   ctid    | itemlen | nulls | vars | dead |   htid    | tids |          data
------------+-----------+---------+-------+------+------+-----------+------+-------------------------
          1 | (5405,54) |      16 | f     | f    | f    | (5405,54) |      | 2b 42 0f 00 00 00 00 00
          2 | (5405,55) |      16 | f     | f    | f    | (5405,55) |      | 2c 42 0f 00 00 00 00 00
          3 | (5405,56) |      16 | f     | f    | f    | (5405,56) |      | 2d 42 0f 00 00 00 00 00
          4 | (5405,57) |      16 | f     | f    | f    | (5405,57) |      | 2e 42 0f 00 00 00 00 00
          5 | (5405,58) |      16 | f     | f    | f    | (5405,58) |      | 2f 42 0f 00 00 00 00 00
          6 | (5405,59) |      16 | f     | f    | f    | (5405,59) |      | 30 42 0f 00 00 00 00 00
          7 | (5405,60) |      16 | f     | f    | f    | (5405,60) |      | 31 42 0f 00 00 00 00 00
          8 | (5405,61) |      16 | f     | f    | f    | (5405,61) |      | 32 42 0f 00 00 00 00 00
          9 | (5405,62) |      16 | f     | f    | f    | (5405,62) |      | 33 42 0f 00 00 00 00 00
         10 | (5405,63) |      16 | f     | f    | f    | (5405,63) |      | 34 42 0f 00 00 00 00 00
         11 | (5405,64) |      16 | f     | f    | f    | (5405,64) |      | 35 42 0f 00 00 00 00 00
         12 | (5405,65) |      16 | f     | f    | f    | (5405,65) |      | 36 42 0f 00 00 00 00 00
         13 | (5405,66) |      16 | f     | f    | f    | (5405,66) |      | 37 42 0f 00 00 00 00 00
         14 | (5405,67) |      16 | f     | f    | f    | (5405,67) |      | 38 42 0f 00 00 00 00 00
         15 | (5405,68) |      16 | f     | f    | f    | (5405,68) |      | 39 42 0f 00 00 00 00 00
         16 | (5405,69) |      16 | f     | f    | f    | (5405,69) |      | 3a 42 0f 00 00 00 00 00
         17 | (5405,70) |      16 | f     | f    | f    | (5405,70) |      | 3b 42 0f 00 00 00 00 00
         18 | (5405,71) |      16 | f     | f    | f    | (5405,71) |      | 3c 42 0f 00 00 00 00 00
         19 | (5405,72) |      16 | f     | f    | f    | (5405,72) |      | 3d 42 0f 00 00 00 00 00
         20 | (5405,73) |      16 | f     | f    | f    | (5405,73) |      | 3e 42 0f 00 00 00 00 00
         21 | (5405,74) |      16 | f     | f    | f    | (5405,74) |      | 3f 42 0f 00 00 00 00 00
         22 | (5405,75) |      16 | f     | f    | f    | (5405,75) |      | 40 42 0f 00 00 00 00 00
(22 rows)
```

Это содержимое самого правого листового блока. Во всех ctid самого правого листового блока хранятся ссылки на строки таблицы. В столбце **htid** (heap tuple id) хранится то же самое значение, что и в `ctid`. Почему дублируется значение в ctid и htid? Алгоритм работы btree оптимизирован для работы в условиях минимума блокировок и для минимизации расщеплений блоков. В процессе чтения блоков индекса одним процессом, другие процессы могут менять его структуру. При навигации по блокам нет единой картины ("целостности по чтению"). Дополнительные поля используются для выявления противоречий.

7) Проверьте, что ctid ссылается на строку таблицы. Выполните запрос к таблице t3 по служебному столбец `ctid`:

```
postgres=# select * from t3 where ctid='(5405,54)';
   id   |   s
--------+--------
 999979 | 999979
(1 row)

postgres=# select * from t3 where ctid='(5405,75)';
   id    |    s
---------+---------
 1000000 | 1000000
(1 row)
```

Строка **ctid='(5405,75)'** была добавлена самой последней в таблицу.

8) Вставьте строку в таблицу и проверьте, что в правый листовой блок была вставлена запись:

```
postgres=# insert into t3 values(default);
```

```
INSERT 0 1
postgres=# select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, data
from bt_page_items('t3_pkey', 2474) where itemoffset>21;
 itemoffset |   ctid    | itemlen | nulls | vars | dead |   htid    | tids |          data
------------+-----------+---------+-------+------+------+-----------+------+------------------------
         22 | (5405,75) |      16 | f     | f    | f    | (5405,75) |      | 40 42 0f 00 00 00 00 00
         23 | (5405,76) |      16 | f     | f    | f    | (5405,76) |      | 41 42 0f 00 00 00 00 00
(2 rows)
```

9) Посмотрите первые записи в любом листовом блоке, кроме самого правого:

```
postgres=# select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, data
from bt_page_items('t3_pkey', 2473) limit 3;
 itemoffset |   ctid    | itemlen | nulls | vars | dead |   htid    | tids |          data
------------+-----------+---------+-------+------+------+-----------+------+----------------------
          1 | (5405,1)  |      16 | f     | f    |      |           |      | 2b 42 0f 00 00 00 00 00
          2 | (5403,18) |      16 | f     | f    | f    | (5403,18) |      | 95 40 0f 00 00 00 00 00
          3 | (5403,19) |      16 | f     | f    | f    | (5403,19) |      | 96 40 0f 00 00 00 00 00
(3 rows)
```

Первая строка (`itemoffset=1`) в листовых блоках кроме самого "правого" всегда хранит служебное значение, называемое "High key". При вставке в структуру индекса нового блока обновляются High keys и ссылки на соседние блоки того же уровня. **High key хранит наибольшее значение, которое встречается в этом блоке индекса**.

High key (первая строка листового блока кроме самого правого листового блока) всегда проверяется при поиске по индексу. Зачем? В процессе спуска с предыдущего уровня до листового другой процесс мог уже расщепить блок, на который спускаются и перераспределить ссылки на строки таблиц, а это значит, что искомое значение находится в блоке (или даже блоках) правее того, на который спустились. Если значение High key листового блока **отличается** от значения в ссылке на следующий листовой блок, то процесс должен двигаться вправо по листовому уровню и проверять, нет ли там искомого значения. Ссылка на следующий листовой блок находится в вышестоящем блоке в записи `itemoffset`+1 (в примере 184).

Часть 5. Дедупликация в btree индексах

Дедупликация появилась в PostgreSQL 13 версии и использует для хранения информации поля, которые без дедупликации предназначены для других целей. Это позволило радикально не менять структуру индекса и не требовать перестройки индексов при миграции на новую версию PostgreSQL.

1) Создайте таблицу с одним столбцом типа int4 или int8. Вставьте в таблицу 407 строк с одинаковым значением 1. Посмотрите содержимое блока индекса:

```
postgres=# drop table if exists td;
create table td(id int8) with (autovacuum_enabled=off);
create index td_idx on td (id);
insert into td select 1 from generate_series(1, 407);
select magic, root, level, fastroot, fastlevel, allequalimage from
bt_metap('td_idx');
select blkno, type, live_items live, dead_items dead, avg_item_size size,
free_size free, btpo_prev, btpo_next, btpo_level l, btpo_flags f from
bt_page_stats('td_idx',1);
select itemoffset o, ctid, itemlen, htid, data, substring(tids::text for 34) tids
from bt_page_items('td_idx',1) limit 3;
DROP TABLE
CREATE TABLE
CREATE INDEX
INSERT 0 407
 magic  | root | level | fastroot | fastlevel | allequalimage
--------+------+-------+----------+-----------+---------------
 340322 |    1 |     0 |        1 |         0 | t
(1 row)
```

Корневым блоком указан блок номер 1. Для поддержки дедупликации в классе операторов для типа данных, который индексируется, должна быть определена функция номер 4 (`BTEQUALIMAGE_PROC`). Если **`allequalimage=t`**, то функция определена и дедупликация поддерживается.
Дедупликация не поддерживается с типами данных: `numeric, jsonb, float4, float8`, массивами, составными, диапазонными типами. Индексы со столбцами INCLUDE не поддерживают дедупликацию. У таких индексов **`allequalimage=f`**.

```
 blkno | type | live | dead | size | free | btpo_prev | btpo_next | l | f
-------+------+------+------+------+------+-----------+-----------+---+---
     1 | l    |  407 |    0 |   16 |    8 |         0 |         0 | 0 | 3
(1 row)
```

Тип блока номер 1 в самом блоке указан как листовой. Левее и правее этого блока блоков нет (`btpo_prev=0` и `btpo_next=0`), блок единственный на своём уровне. В блоке 407 записи (`live=407`). В блоке свободно 8 байт (`free=8`).
Индекс состоит из двух блоков: нулевого блока с метаданными и единственного блока.

```
 o | ctid  | itemlen | htid  |          data           |   tids
---+-------+---------+-------+-------------------------+-----------
 1 | (0,1) |      16 | (0,1) | 01 00 00 00 00 00 00 00 |
 2 | (0,2) |      16 | (0,2) | 01 00 00 00 00 00 00 00 |
 3 | (0,3) |      16 | (0,3) | 01 00 00 00 00 00 00 00 |
(3 rows)
```

Блок имеет формат правого листового блока. В каждой записи в поле `ctid` хранится ссылка на строку таблицы, в поле `data` хранится проиндексированное значение. Так как блок правый, то он заполнен полностью так, что в нём нет места для вставки новой записи.

2) Вставьте одну строку:

```
postgres=# insert into td values(1);
INSERT 0 1
```

3) Посмотрите как поменялось содержимое блока 1 индекса:

```
postgres=# select blkno, type, live_items live, dead_items dead, avg_item_size
size, free_size free, btpo_prev, btpo_next, btpo_level l, btpo_flags f from
bt_page_stats('td_idx',1);
 select itemoffset o, ctid, itemlen, htid, data, substring(tids::text for 34)
tids from bt_page_items('td_idx',1);
```

```
 blkno | type | live | dead | size | free | btpo_prev | btpo_next | l | f
-------+------+------+------+------+------+-----------+-----------+---+--
     1 | l    |    3 |    0 |  832 | 5640 |         0 |         0 | 0 | 3
(1 row)
```

Была выполнена дедупликация. Дедупликация выполняется, если блок должен был бы делиться: в блоке нет места или превышается fillfactor.
В блоке было 407 записи, а стало 3 записи.
В блоке свободно 5640 байт.

```
 o |   ctid     | itemlen |  htid   |          data           |              tids
---+------------+---------+---------+-------------------------+-----------------------------------
 1 | (16,8414)  |    1352 | (0,1)   | 01 00 00 00 00 00 00 00 | {"(0,1)","(0,2)","(0,3)","(0,4)","
 2 | (16,8377)  |    1128 | (0,223) | 01 00 00 00 00 00 00 00 | {"(0,223)","(0,224)","(0,225)","(0
 3 | (1,182)    |      16 | (1,182) | 01 00 00 00 00 00 00 00 |
(3 rows)
```

В поле data хранится значение проиндексированных столбцов. В примере это целое число 1.

Значения проиндексированных столбцов хранятся в записи индекса, а ссылки на строки таблицы хранится в столбце tids (tuple ids, идентификаторы строк таблицы) в виде отсортированного массива значений типа ctid.
В строке itemoffset=3 нет дедупликации (tids пуст) и на строку таблицы указывает ctid=(1,182).
В первой и второй строках на строки таблицы указывает tids.
В htid сохраняется первый tid из tids.
В ctid первой и второй записи хранятся **не** ссылки на блоки, а служебные данные о tids. Например, ctid промежуточных блоков будет хранить номер блока в индексе, ссылающийся на нижестоящий уровень, а вторая часть ctid будет хранить число элементов в tids.

Размер массива tids указан в столбце `itemlen`:

```
postgres=# select itemoffset o, ctid, itemlen, htid, pg_column_size(tids) size,
cardinality(tids) from bt_page_items('td_idx',1);
 o |   ctid    | itemlen |  htid   | size | cardinality
---+-----------+---------+---------+------+-------------
 1 | (16,8414) |    1352 | (0,1)   | 1356 |         222
 2 | (16,8377) |    1128 | (0,223) | 1134 |         185
 3 | (1,182)   |      16 | (1,182) |      |
(3 rows)
```

Размер типа ctid 6 байт. Число элементов в массиве выдаёт функция **`cardinality(tids)`**. Вычисление размера массива в столбце tids: 222*6=1332. 185*6=1110.
Это соответствует значениям, выдаваемым функцией `pg_column_size(tids)`. Функция выдаёт значения, увеличенные на 24 байта (1332+24=1356 и 1110+24=1134), так как предназначена для выдачи размера строки таблицы. 24 байта это минимальный размер заголовка строки.

4) Посмотрим как определить поддерживает ли индекс дедупликацию.
У индексов не поддерживающих дедупликацию **`allequalimage=f`**.
Дедупликация не поддерживается с типами данных: `numeric, jsonb, float4, float8,` массивами, составными типами, диапазонными типами:

```
postgres=# drop table if exists td;
create table td(id float8);
create index td_idx on td (id);
select allequalimage from bt_metap('td_idx');
DROP TABLE
CREATE TABLE
CREATE INDEX
 allequalimage
---------------
 f
(1 row)

postgres=# drop table if exists td;
create table td(id int8[]);
create index td_idx on td (id);
select allequalimage from bt_metap('td_idx');
DROP TABLE
CREATE TABLE
CREATE INDEX
 allequalimage
---------------
 f
(1 row)

postgres=# drop table if exists td;
create table td(id jsonb);
create index td_idx on td (id);
```

```
select allequalimage from bt_metap('td_idx');
DROP TABLE
CREATE TABLE
CREATE INDEX
 allequalimage
---------------
 f
(1 row)

postgres=# drop table if exists td;
create table td(n timestamp, n1 date, n2 integer, n3 char, n4 text, n5 varchar);
create index td_idx on td (n,n1,n2,n3,n4,n5);
select allequalimage from bt_metap('td_idx');
DROP TABLE
CREATE TABLE
CREATE INDEX
 allequalimage
---------------
 t
(1 row)
```

Составной индекс поддерживает дедупликацию, если типы ключевые данных ее поддерживают.

```
postgres=# create index td1_idx on td (n) include (n1);
select allequalimage from bt_metap('td1_idx');
CREATE INDEX
 allequalimage
---------------
 f
(1 row)
```

Индексы со столбцами INCLUDE не поддерживают дедупликацию даже если типы данных поддерживают.

Часть 6. Индексы в убывающем порядке

По умолчанию индекс строится в возрастающем порядке, то есть "слева" меньшие значения, "справа" большие. При создании индекса можно указать обратный порядок: DESC. Не стоит это делать для индексов заполняемых возрастающей последовательностью. Свойство ASC и DESC при создании индекса не влияет на эффективность использования индекса планировщиком (например, ORDER BY ASC или DESC). Это свойство влияет на заполнение индекса: правые блоки в индексе отличаются от остальных тем, что оптимизированы для вставок. Желательно, чтобы вставки выполнялись преимущественно в правый листовой блок индекса.

1) Включите измерение времени выполнения команд:

```
postgres=# \timing
Timing is on.
```

2) Создайте таблицу и индекс в прямом и обратном порядке сортировки:

```
postgres=# drop table if exists t3;
 create table t3 (id bigserial, s int4) with (autovacuum_enabled=off,
fillfactor=100);
 create unique index if not exists t3_pkey on t3 using btree (id) include (s)
with (fillfactor=100, deduplicate_items=off);
 insert into t3 (s) select * from generate_series(1, 1000000);
 select pg_relation_size('t3_pkey', 'main');

 drop table if exists t3;
```

```
 create table t3 (id bigserial, s int4) with (autovacuum_enabled=off,
fillfactor=100);
 create unique index if not exists t3_pkey on t3 using btree (id DESC nulls
first) include (s) with (fillfactor=100, deduplicate_items=off);
 insert into t3 (s) select * from generate_series(1, 1000000);
 select pg_relation_size('t3_pkey', 'main');

 DROP TABLE
 Time: 29.181 ms
 CREATE TABLE
 Time: 5.667 ms
 CREATE INDEX
 Time: 15.279 ms
 INSERT 0 1000000
 Time: 4351.432 ms (00:04.351)
  pg_relation_size
 ------------------
          28467200
 (1 row)
 Time: 0.229 ms

 DROP TABLE
 Time: 23.951 ms
 CREATE TABLE
 Time: 7.504 ms
 CREATE INDEX
 Time: 11.651 ms
 INSERT 0 1000000
 Time: 5740.328 ms (00:05.740)
  pg_relation_size
 ------------------
          56401920
 (1 row)
 Time: 0.307 ms
```

Изменение порядка существенно повлияло на скорость вставки и размер индекса.
Размер индекса увеличился в 2 раза. Скорость вставки снизилась на 32%.
Если бы значения заполнялись убывающей последовательностью, то индекс DESC был бы оптимальнее, чем ASC.

3) Перестройте индекс и посмотрите уменьшился ли размер индекса:

```
postgres=# reindex index t3_pkey;
REINDEX
postgres=# select pg_relation_size('t3_pkey', 'main');
 pg_relation_size
------------------
         28475392
(1 row)
Time: 1077.214 ms (00:01.077)
```

Размер индекса уменьшился. Перестройка индекса была эффективна. Индекс при вставке строк в таблицу обновлялся неэффективно. Обновления происходили в левых блоках индекса, которые не оптимизированы для вставок, в отличие от правых блоков индекса btree.

При вставке строк в таблицу значениями, которые попадают не в правый блок структура индекса заполняется неэффективно и вставки замедляются.
Причина замедления вставок не столько в излишнем делении блоков, сколько в отсутствии оптимизации fastpath. Процесс, который выполнил вставку в правый листовой блок, запоминает

ссылку на него и при последующей вставке, если новое значение больше предыдущего (или пусто) и не проходит путь от корня до листового блока. Оптимизация fastpath также используется при вставке в датавременной столбец, заполняемый по DEFAULT временем вставки. Процесс забывает адрес блока и снова начинает поиск с корня, если по какой-либо причине выполнил вставку (в индекс записи только вставляются, они не меняются, а удаляются только вакуумом) не в самый правый блок. Fastpath применяется при числе уровней в индексе 2 и больше.

Помимо значений, заполняющихся последовательностью, нужно помнить и о пустых значениях.
По умолчанию пустые значения сохраняются "справа" (в правых блоках индекса). Это можно переопределить указав NULLS FIRST.
При переопределении порядка, разработчики приложений обычно исходят из того, что должно выдаваться преимущественно первым при сортировке. Эта идея неверна, так как серверный процесс скользит по блокам в обе стороны с одинаковой эффективностью.
Использование NULLS FIRST может повлиять на производительность: если при вставке строк в таблицу в индекс вставляется NULL (при вставке строки в таблицу значение индексированного столбца не задаётся, а обновляется позже и обновления распределены по времени, а не массовые), то оптимизация fastpath перестает работать, так как NULL будут в самом левом листовом блоке, а fastpath работает только с правым блоком. Вставки строк с NULL замедлятся при использовании NULLS FIRST. Деградация производительности такая же, как при использовании DESC в приведённом примере, поэтому пример для NULLS FIRST не приводится.

4) Отключите измерение времени:

```
postgres=# \timing
Timing is off.
```

Часть 7. Покрывающие индексы и Index Only Scan

Индекс t3_pkey был создан с опцией **`include (s)`** означающей, что в листовых блоках индекса сохраняются значения столбца s. Хранение значений столбцов увеличивает размер индекса.
Такие индексы используются, чтобы можно было использовать метод доступа Index Only Scan. Этот метод используется, если упоминаемые в запросе столбцы присутствуют в индексе. Можно сказать, что запрос "покрывается" индексом. Тогда индекс называют "покрывающим" для запроса. Трудоёмкость выполнения команды методом Index Only Scan существенно уменьшается, так как не нужно обращаться к блокам таблицы. Все нужные запросу значения берутся из индекса.
Столбцы, добавленные **`include (..)`** не влияют на структуру индекса.
Почему бы не добавить столбец в ключевые? Если тип данных столбца не поддерживает операцию сравнения, то такой тип данных нельзя добавить в ключевые столбцы. Также **`include (..)`** столбцы отсутствуют в промежуточных блоках, что немного уменьшает размер индекса.

1) Посмотрите план выполнения команд:

```
postgres=# explain select * from t3 where id=1;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Only Scan using t3_pkey on t3  (cost=0.42..8.44 rows=1 width=12)
   Index Cond: (id = 1)
(2 rows)

postgres=# explain select * from t3 where id=4 and s>3;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Only Scan using t3_pkey on t3  (cost=0.42..8.45 rows=1 width=12)
   Index Cond: (id = 4)
   Filter: (s > 3)
(3 rows)
```

Используется метод **Index Only Scan**.

2) Посмотрите план выполнения команды:

```
postgres=# explain select s from t3 where id>4 and id < 10;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on t3  (cost=123.67..5717.65 rows=5000 width=4)
   Recheck Cond: ((id > 4) AND (id < 10))
   ->  Bitmap Index Scan on t3_pkey  (cost=0.00..122.42 rows=5000 width=0)
         Index Cond: ((id > 4) AND (id < 10))
(4 rows)
```

Index Only Scan не используется. Планировщик сильно ошибается в числе строк: по его оценке запрос выдает 5000 строк.

3) Соберите статистику:

```
postgres=# analyze t3;
ANALYZE
```

4) Повторите команду:

```
postgres=# explain select s from t3 where id>4 and id < 10;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Only Scan using t3_pkey on t3  (cost=0.42..8.53 rows=5 width=4)
   Index Cond: ((id > 4) AND (id < 10))
(2 rows)
```

Стал использоваться Index Only Scan.
Оценка числа строк становится правильной **rows=5**.

5) В **include** могут присутствовать только столбцы, но не выражения:

```
postgres=# create unique index if not exists t3_pkey1 on t3 using btree (id)
include (UPPER(s));
ERROR:  expressions are not supported in included columns
```

Часть 8. Частичные (partial) индексы

Частичные (partial) индексы создаются по части строк таблицы. Часть строк определяется предикатом WHERE, который указывается при создании индекса и делает индекс частичным.

Частичные индексы полезны тем, что позволяют избежать индексирования наиболее часто встречающихся значений. Наиболее часто встречающееся значение - это значение, которое содержится в значительном проценте всех строк таблицы. При поиске наиболее часто встречающихся значений индекс всё равно не будет использоваться, так как более эффективным будет сканирование всех строк таблицы. Индексировать строки с наиболее часто встречающимися значениями нет смысла. Исключив такие строки из индекса, можно уменьшить размер индекса, что ускорит вакуумирование таблицы. Также ускоряется внесение изменений в строки таблицы, если индекс не затрагивается.

Вторая причина, по которой используется частичный индекс это когда отсутствуют обращения к части строк таблицы. Если обращения присутствуют, то используется не индексный доступ, а полное сканирование таблицы.

Частичный индекс может быть уникальным.

Создавать большое число частичных индексов, которые индексируют разные строки не стоит. Чем больше индексов на таблице, тем ниже производительность команд, изменяющих данные; автовакуума; вероятность использования быстрого пути блокировок уменьшается.

1) Посмотрите какой размер у индекса t3_pkey:

```
postgres=# select pg_relation_size('t3_pkey', 'main');
 pg_relation_size
------------------
         28475392
(1 row)
```

2) Удалите индекс и создайте частичный индекс по строкам в которых s<1000:

```
postgres=# drop index t3_pkey;
DROP INDEX
postgres=# create unique index t3_pkey on t3 using btree (id) include (s) WHERE
s<1000;
CREATE INDEX
```

3) Посмотрите какой размер у индекса:

```
postgres=# select pg_relation_size('t3_pkey', 'main');
 pg_relation_size
------------------
            49152
(1 row)
```

Размер индекса существенно меньше, чем индекса по всем строкам.

4) Выполните запрос:

```
postgres=# explain select * from t3 where id=4 and s>3;
                              QUERY PLAN
-----------------------------------------------------------------------
 Gather  (cost=1000.00..12656.10 rows=1 width=12)
   Workers Planned: 2
   ->  Parallel Seq Scan on t3  (cost=0.00..11656.00 rows=1 width=12)
         Filter: ((s > 3) AND (id = 4))
(4 rows)
```

Хотя запрос и выдаёт строки, на которые есть ссылки в индексе, но планировщик об этом не знает.

5) Выполните запрос с явно заданным условием s<3:

```
postgres=# explain select * from t3 where id=4 and s<3;
                               QUERY PLAN
------------------------------------------------------------------------
 Index Only Scan using t3_pkey on t3  (cost=0.28..8.29 rows=1 width=12)
   Index Cond: (id = 4)
   Filter: (s < 3)
(3 rows)
```

Частичный индекс используется.

Часть 9. Изучение структуры индекса типа btree

В главе приводился пример структуры индекса.

1) Создайте таблицу, индекс, вставьте три строки:

```
postgres=# drop table if exists t;
create table t(s text storage plain) with (autovacuum_enabled=off,
fillfactor=10);
create index t_idx on t (s) with (fillfactor=10, deduplicate_items = off);
insert into t values (repeat('a',2500));
insert into t values (repeat('b',2500));
insert into t values (repeat('c',2500));
```

Размер полей выбран так, чтобы в блок таблицы помещалась одна строка. В блок индекса индекса помещались 3-4 строки.

2) Адрес корневого блока индекса и число уровней можно посмотреть запросом:

```
postgres=# select root, level, fastroot, fastlevel, allequalimage from
bt_metap('t_idx');
 root | level | fastroot | fastlevel | allequalimage
------+-------+----------+-----------+---------------
    1 |     0 |        1 |         0 | t
(1 row)
```

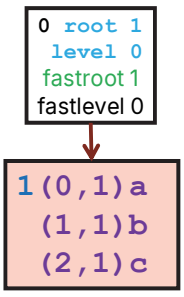
Статистику по всем блокам индекса можно посмотреть запросом:

```
postgres=# select blkno, type, live_items live, avg_item_size size, free_size
free, btpo_prev prev,  btpo_next next, btpo_level level, btpo_flags fl from
bt_multi_page_stats('t_idx',1,-1);
 blkno | type | live | size | free | prev | next | level | fl
-------+------+------+------+------+------+------+-------+----
     1 | l    |    3 | 2512 |  600 |    0 |    0 |     0 |  3
(1 row)
```

Содержимое блока индекса можно посмотреть запросом:

```
postgres=# select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',1);
 o | ctid  | itemlen | htid  |        data        | c
---+-------+---------+-------+--------------------+---
 1 | (0,1) |    2512 | (0,1) | 20 27 00 00 61 61  | a
 2 | (1,1) |    2512 | (1,1) | 20 27 00 00 62 62  | b
 3 | (2,1) |    2512 | (2,1) | 20 27 00 00 63 63  | c
(3 rows)
```

На основе этих запросов можно нарисовать структуру индекса:



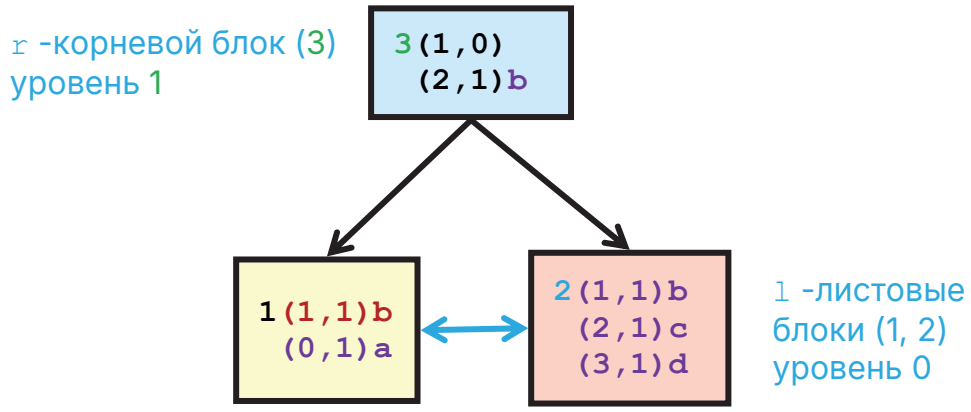
В индексе два блока: нулевой всегда содержит метаданные; блок номер 1 содержит три строки. Для компактности значение отображено одной буквой, а не 2500 буквами. Буква a в шестнадцатеричном виде 61, буква b - 62, буква c - 63.

Дальше, вставляя по одной строке, будут выполнены запросы и на основе запросов нарисована структура индекса. Запросы в следующих пунктах не обязательно выполнять, достаточно

просматривать результаты запросов и сопоставлять их с рисунком. Это полезно, чтобы понять как растёт индекс.

4) После вставки четвёртой строки можно выполнить следующие запросы и на основе их результатов нарисовать структуру индекса:

```
postgres=# insert into t values (repeat('d',2500));
select ctid, left(s, 24) from t;
select blkno, type, live_items live, avg_item_size size, free_size free, btpo_prev prev,  btpo_next next,
btpo_level level, btpo_flags fl from bt_multi_page_stats('t_idx',1,-1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',2);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',3);
INSERT 0 1
 ctid  |           left
-------+--------------------------
 (0,1) | aaaaaaaaaaaaaaaaaaaaaaaa
 (1,1) | bbbbbbbbbbbbbbbbbbbbbbbb
 (2,1) | cccccccccccccccccccccccc
 (3,1) | dddddddddddddddddddddddd
(4 rows)

 blkno | type | live | size | free | prev | next | level | fl
-------+------+------+------+------+------+------+-------+----
     1 | l    |    2 | 2512 | 3116 |    0 |    2 |     0 |  1
     2 | l    |    3 | 2512 |  600 |    1 |    0 |     0 |  1
     3 | r    |    2 | 1260 | 5620 |    0 |    0 |     1 |  2
(3 rows)

 o | ctid  | itemlen | htid  |        data         | c
---+-------+---------+-------+---------------------+---
 1 | (1,1) |    2512 |       | 20 27 00 00 62 62   | b
 2 | (0,1) |    2512 | (0,1) | 20 27 00 00 61 61   | a
(2 rows)

 o | ctid  | itemlen | htid  |        data         | c
---+-------+---------+-------+---------------------+---
 1 | (1,1) |    2512 | (1,1) | 20 27 00 00 62 62   | b
 2 | (2,1) |    2512 | (2,1) | 20 27 00 00 63 63   | c
 3 | (3,1) |    2512 | (3,1) | 20 27 00 00 64 64   | d
(3 rows)

 o | ctid  | itemlen | htid |        data         | c
---+-------+---------+------+---------------------+---
 1 | (1,0) |       8 |      |                     |
 2 | (2,1) |    2512 |      | 20 27 00 00 62 62   | b
(2 rows)
```

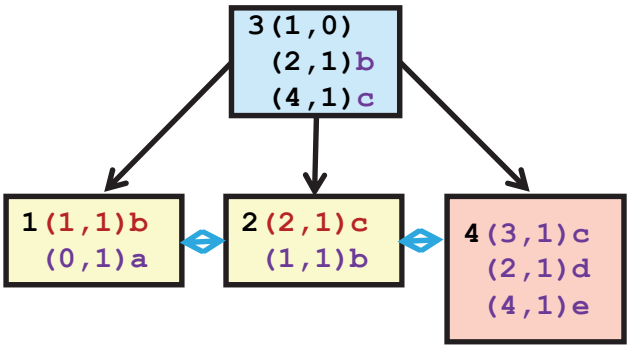


Корневым стал 3 блок. При дальнейших вставках корневой блок будет также меняться.

5) После вставки четвёртой строки можно повторить запросы, добавив к ним запрос по 4 блоку. Блоки в индекс добавляются один за другим и в индекс будет добавлен 4 блок.

```
postgres=# insert into t values (repeat('e',2500));
```

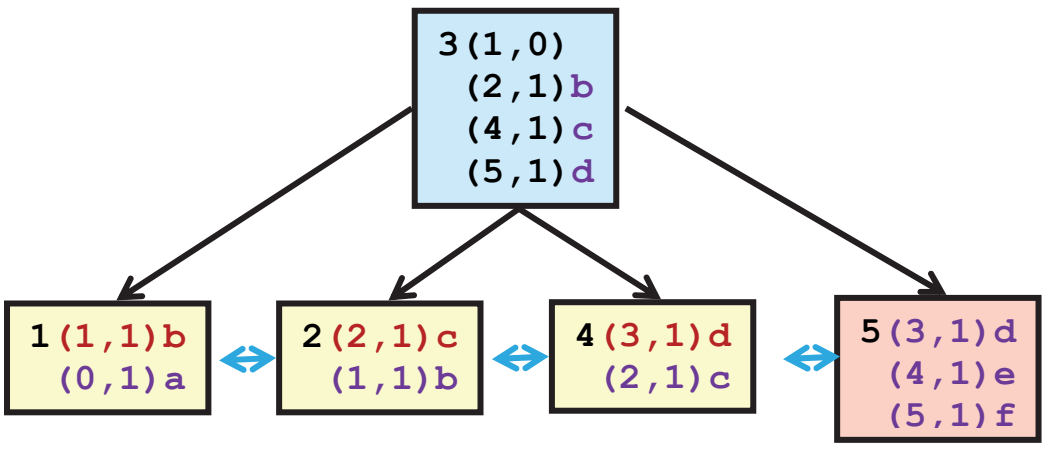
```
 select ctid, left(s, 24) from t;
 select blkno, type, live_items live, avg_item_size size, free_size free, btpo_prev prev,  btpo_next next,
btpo_level level, btpo_flags fl from bt_multi_page_stats('t_idx',1,-1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',2);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',3);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',4);
 INSERT 0 1
  ctid  |           left
 -------+--------------------------
  (0,1) | aaaaaaaaaaaaaaaaaaaaaaa
  (1,1) | bbbbbbbbbbbbbbbbbbbbbbb
  (2,1) | ccccccccccccccccccccccc
  (3,1) | ddddddddddddddddddddddd
  (4,1) | eeeeeeeeeeeeeeeeeeeeeee
 (5 rows)

  blkno | type | live | size | free | prev | next | level | fl
 -------+------+------+------+------+------+------+-------+----
      1 | l    |    2 | 2512 | 3116 |    0 |    2 |     0 |  1
      2 | l    |    2 | 2512 | 3116 |    1 |    4 |     0 |  1
      3 | r    |    3 | 1677 | 3104 |    0 |    0 |     1 |  2
      4 | l    |    3 | 2512 |  600 |    2 |    0 |     0 |  1
 (4 rows)

  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
  1 | (1,1) |    2512 |       | 20 27 00 00 62 62   | b
  2 | (0,1) |    2512 | (0,1) | 20 27 00 00 61 61   | a
 (2 rows)

  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
  1 | (2,1) |    2512 |       | 20 27 00 00 63 63   | c
  2 | (1,1) |    2512 | (1,1) | 20 27 00 00 62 62   | b
 (2 rows)

  o | ctid  | itemlen | htid |        data         | c
 ---+-------+---------+------+---------------------+---
  1 | (1,0) |       8 |      |                     |
  2 | (2,1) |    2512 |      | 20 27 00 00 62 62   | b
  3 | (4,1) |    2512 |      | 20 27 00 00 63 63   | c
 (3 rows)

  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
  1 | (2,1) |    2512 | (2,1) | 20 27 00 00 63 63   | c
  2 | (3,1) |    2512 | (3,1) | 20 27 00 00 64 64   | d
  3 | (4,1) |    2512 | (4,1) | 20 27 00 00 65 65   | e
 (3 rows)
```



6) Повтор для 6 строки:

```
postgres=# insert into t values (repeat('f',2500));
 select ctid, left(s, 24) from t;
 select blkno, type, live_items live, avg_item_size size, free_size free, btpo_prev prev,  btpo_next next,
btpo_level level, btpo_flags fl from bt_multi_page_stats('t_idx',1,-1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',2);
```

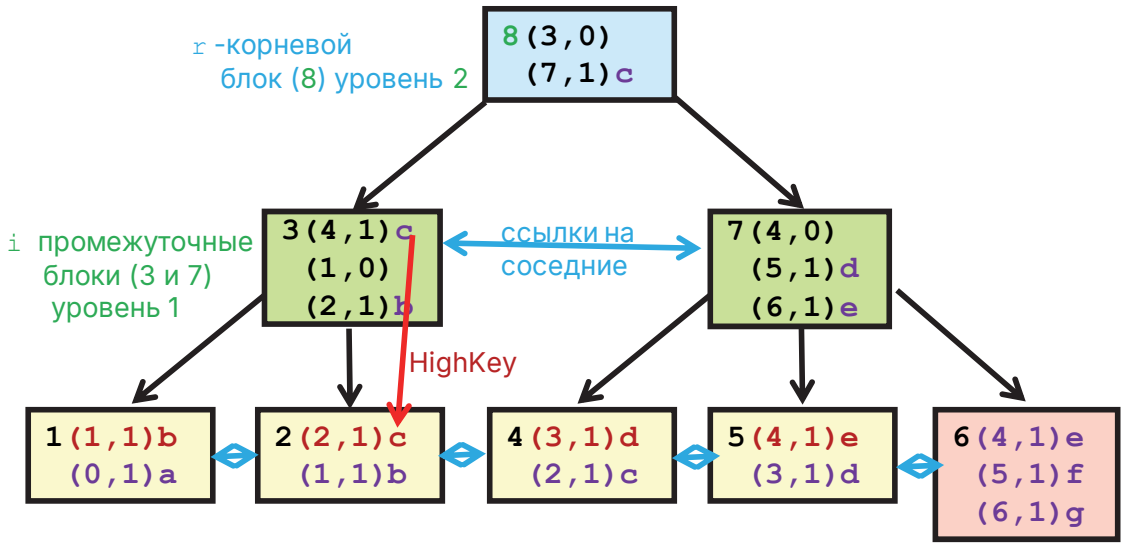
```
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',3);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',4);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',5);
 INSERT 0 1
  ctid  |           left
-------+--------------------------
  (0,1) | aaaaaaaaaaaaaaaaaaaaaaa
  (1,1) | bbbbbbbbbbbbbbbbbbbbbbb
  (2,1) | ccccccccccccccccccccccc
  (3,1) | ddddddddddddddddddddddd
  (4,1) | eeeeeeeeeeeeeeeeeeeeeee
  (5,1) | fffffffffffffffffffffff
 (6 rows)

 blkno | type | live | size | free | prev | next | level | fl
-------+------+------+------+------+------+------+-------+----
     1 | l    |    2 | 2512 | 3116 |    0 |    2 |     0 |  1
     2 | l    |    2 | 2512 | 3116 |    1 |    4 |     0 |  1
     3 | r    |    4 | 1886 |  588 |    0 |    0 |     1 |  2
     4 | l    |    2 | 2512 | 3116 |    2 |    5 |     0 |  1
     5 | l    |    3 | 2512 |  600 |    4 |    0 |     0 |  1
 (5 rows)


 o | ctid  | itemlen | htid  |        data         | c
---+-------+---------+-------+---------------------+---
 1 | (1,1) |    2512 |       | 20 27 00 00 62 62   | b
 2 | (0,1) |    2512 | (0,1) | 20 27 00 00 61 61   | a
(2 rows)

 o | ctid  | itemlen | htid  |        data         | c
---+-------+---------+-------+---------------------+---
 1 | (2,1) |    2512 |       | 20 27 00 00 63 63   | c
 2 | (1,1) |    2512 | (1,1) | 20 27 00 00 62 62   | b
(2 rows)

 o | ctid  | itemlen | htid |        data         | c
---+-------+---------+------+---------------------+---
 1 | (1,0) |       8 |      |                     |
 2 | (2,1) |    2512 |      | 20 27 00 00 62 62   | b
 3 | (4,1) |    2512 |      | 20 27 00 00 63 63   | c
 4 | (5,1) |    2512 |      | 20 27 00 00 64 64   | d
(4 rows)

 o | ctid  | itemlen | htid  |        data         | c
---+-------+---------+-------+---------------------+---
 1 | (3,1) |    2512 |       | 20 27 00 00 64 64   | d
 2 | (2,1) |    2512 | (2,1) | 20 27 00 00 63 63   | c
(2 rows)

 o | ctid  | itemlen | htid  |        data         | c
---+-------+---------+-------+---------------------+---
 1 | (3,1) |    2512 | (3,1) | 20 27 00 00 64 64   | d
 2 | (4,1) |    2512 | (4,1) | 20 27 00 00 65 65   | e
 3 | (5,1) |    2512 | (5,1) | 20 27 00 00 66 66   | f
(3 rows)
```



7) Повтор для 7 строки. В индекс будет добавлено сразу три блока, так как увеличится число уровней индекса и на двух уровнях блоки разделятся:

```
postgres=# insert into t values (repeat('g',2500));
 select blkno, type, live_items live, avg_item_size size, free_size free, btpo_prev prev,  btpo_next next,
btpo_level level, btpo_flags fl from bt_multi_page_stats('t_idx',1,-1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',2);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',3);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',4);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',5);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',6);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',7);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',8);
 INSERT 0 1
  blkno | type | live | size | free | prev | next | level | fl
 -------+------+------+------+------+------+------+-------+----
      1 | l    |    2 | 2512 | 3116 |    0 |    2 |     0 |  1
      2 | l    |    2 | 2512 | 3116 |    1 |    4 |     0 |  1
      3 | i    |    3 | 1677 | 3104 |    0 |    7 |     1 |  0
      4 | l    |    2 | 2512 | 3116 |    2 |    5 |     0 |  1
      5 | l    |    2 | 2512 | 3116 |    4 |    6 |     0 |  1
      6 | l    |    3 | 2512 |  600 |    5 |    0 |     0 |  1
      7 | i    |    3 | 1677 | 3104 |    3 |    0 |     1 |  0
      8 | r    |    2 | 1260 | 5620 |    0 |    0 |     2 |  2
 (8 rows)

  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
  1 | (1,1) |    2512 |       | 20 27 00 00 62 62   | b
  2 | (0,1) |    2512 | (0,1) | 20 27 00 00 61 61   | a
 (2 rows)

  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
  1 | (2,1) |    2512 |       | 20 27 00 00 63 63   | c
  2 | (1,1) |    2512 | (1,1) | 20 27 00 00 62 62   | b
 (2 rows)

  o | ctid  | itemlen | htid |        data         | c
 ---+-------+---------+------+---------------------+---
  1 | (4,1) |    2512 |      | 20 27 00 00 63 63   | c
  2 | (1,0) |       8 |      |                     |
  3 | (2,1) |    2512 |      | 20 27 00 00 62 62   | b
 (3 rows)

  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
  1 | (3,1) |    2512 |       | 20 27 00 00 64 64   | d
  2 | (2,1) |    2512 | (2,1) | 20 27 00 00 63 63   | c
 (2 rows)

  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
  1 | (4,1) |    2512 |       | 20 27 00 00 65 65   | e
  2 | (3,1) |    2512 | (3,1) | 20 27 00 00 64 64   | d
 (2 rows)

  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
  1 | (4,1) |    2512 | (4,1) | 20 27 00 00 65 65   | e
  2 | (5,1) |    2512 | (5,1) | 20 27 00 00 66 66   | f
  3 | (6,1) |    2512 | (6,1) | 20 27 00 00 67 67   | g
 (3 rows)

  o | ctid  | itemlen | htid |        data         | c
 ---+-------+---------+------+---------------------+---
  1 | (4,0) |       8 |      |                     |
  2 | (5,1) |    2512 |      | 20 27 00 00 64 64   | d
  3 | (6,1) |    2512 |      | 20 27 00 00 65 65   | e
 (3 rows)

  o | ctid  | itemlen | htid |        data         | c
```

```
---+-------+---------+------+---------------------+---
 1 | (3,0) |       8 |      |                     |
 2 | (7,1) |    2512 |      | 20 27 00 00 63 63   | c
(2 rows)
```



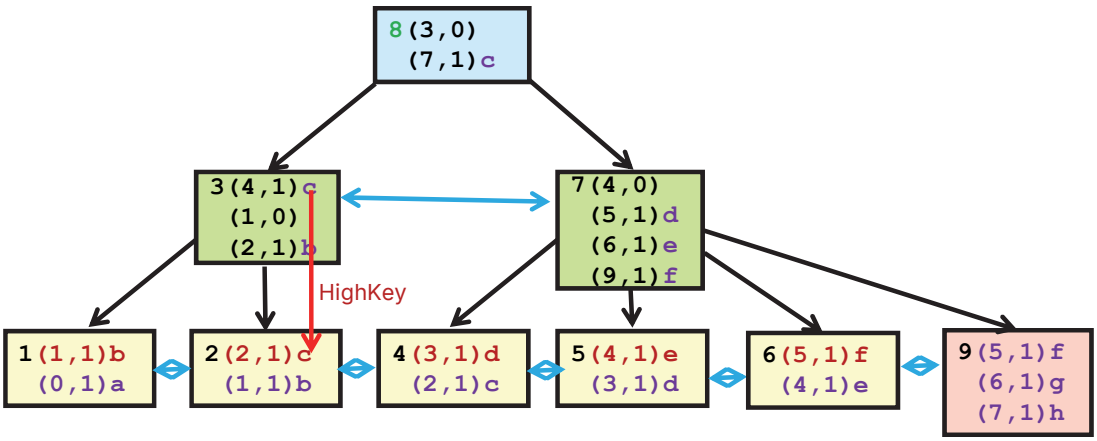
Корневым стал 8 блок вместо 3.

Добавление нового уровня привело к особенности в 3 блоке. Вторую запись в 3 блоке "логичнее" было бы сделать первой. Однако, она идёт второй и это не является ошибкой. Также на 4 блок есть два указателя из двух блоков вышестоящего уровня. Это также не является ошибкой (поиск значений по индексу не нарушается).

8) Повтор для 8 строки:

```
postgres=# insert into t values (repeat('h',2500));
 select blkno, type, live_items live, avg_item_size size, free_size free, btpo_prev prev,  btpo_next next,
btpo_level level, btpo_flags fl from bt_multi_page_stats('t_idx',1,-1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',2);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',3);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',4);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',5);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',6);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',7);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',8);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',9);
 INSERT 0 1
  blkno | type | live | size | free | prev | next | level | fl
 -------+------+------+------+------+------+------+-------+----
      1 | l    |    2 | 2512 | 3116 |    0 |    2 |     0 |  1
      2 | l    |    2 | 2512 | 3116 |    1 |    4 |     0 |  1
      3 | i    |    3 | 1677 | 3104 |    0 |    7 |     1 |  0
      4 | l    |    2 | 2512 | 3116 |    2 |    5 |     0 |  1
      5 | l    |    2 | 2512 | 3116 |    4 |    6 |     0 |  1
      6 | l    |    2 | 2512 | 3116 |    5 |    9 |     0 |  1
      7 | i    |    4 | 1886 |  588 |    3 |    0 |     1 |  0
      8 | r    |    2 | 1260 | 5620 |    0 |    0 |     2 |  2
      9 | l    |    3 | 2512 |  600 |    6 |    0 |     0 |  1
 (9 rows)

  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
  1 | (1,1) |    2512 |       | 20 27 00 00 62 62   | b
  2 | (0,1) |    2512 | (0,1) | 20 27 00 00 61 61   | a
 (2 rows)

  o | ctid  | itemlen | htid  |        data         | c
```

```
 1 | (2,1) |    2512 |       | 20 27 00 00 63 63  | c
 2 | (1,1) |    2512 | (1,1) | 20 27 00 00 62 62  | b
(2 rows)

 o | ctid  | itemlen | htid |        data         | c
---+-------+---------+------+---------------------+---
 1 | (4,1) |    2512 |      | 20 27 00 00 63 63   | c
 2 | (1,0) |       8 |      |                     |
 3 | (2,1) |    2512 |      | 20 27 00 00 62 62   | b
(3 rows)

 o | ctid  | itemlen | htid  |        data         | c
---+-------+---------+-------+---------------------+---
 1 | (3,1) |    2512 |       | 20 27 00 00 64 64   | d
 2 | (2,1) |    2512 | (2,1) | 20 27 00 00 63 63   | c
(2 rows)

 o | ctid  | itemlen | htid  |        data         | c
---+-------+---------+-------+---------------------+---
 1 | (4,1) |    2512 |       | 20 27 00 00 65 65   | e
 2 | (3,1) |    2512 | (3,1) | 20 27 00 00 64 64   | d
(2 rows)

 o | ctid  | itemlen | htid  |        data         | c
---+-------+---------+-------+---------------------+---
 1 | (5,1) |    2512 |       | 20 27 00 00 66 66   | f
 2 | (4,1) |    2512 | (4,1) | 20 27 00 00 65 65   | e
(2 rows)

 o | ctid  | itemlen | htid |        data         | c
---+-------+---------+------+---------------------+---
 1 | (4,0) |       8 |      |                     |
 2 | (5,1) |    2512 |      | 20 27 00 00 64 64   | d
 3 | (6,1) |    2512 |      | 20 27 00 00 65 65   | e
 4 | (9,1) |    2512 |      | 20 27 00 00 66 66   | f
(4 rows)

 o | ctid  | itemlen | htid |        data         | c
---+-------+---------+------+---------------------+---
 1 | (3,0) |       8 |      |                     |
 2 | (7,1) |    2512 |      | 20 27 00 00 63 63   | c
(2 rows)

 o | ctid  | itemlen | htid  |        data         | c
---+-------+---------+-------+---------------------+---
 1 | (5,1) |    2512 | (5,1) | 20 27 00 00 66 66   | f
 2 | (6,1) |    2512 | (6,1) | 20 27 00 00 67 67   | g
 3 | (7,1) |    2512 | (7,1) | 20 27 00 00 68 68   | h
(3 rows)
```



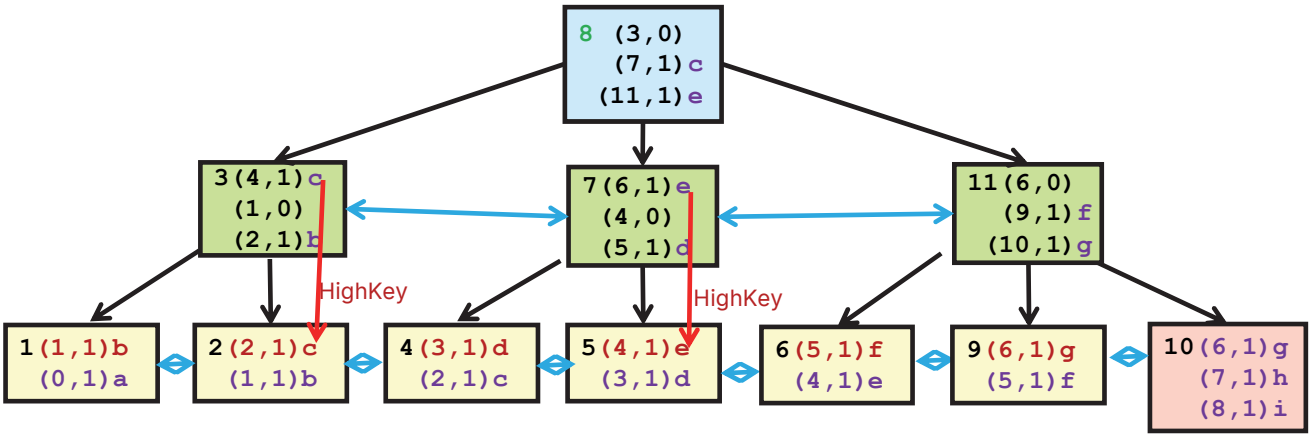
В индекс добавится один блок. Содержимое блоков 6 и 7 поменялось.

9) Повтор для 9 строки.

```
postgres=# insert into t values (repeat('i',2500));
 select blkno, type, live_items live, avg_item_size size, free_size free, btpo_prev prev,  btpo_next next,
btpo_level level, btpo_flags fl from bt_multi_page_stats('t_idx',1,-1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',2);
```

```
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',3);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',4);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',5);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',6);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',7);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',8);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',9);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',10);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',11);
 INSERT 0 1
  blkno | type | live | size | free | prev | next | level | fl
 -------+------+------+------+------+------+------+-------+----
      1 | l    |    2 | 2512 | 3116 |    0 |    2 |     0 |  1
      2 | l    |    2 | 2512 | 3116 |    1 |    4 |     0 |  1
      3 | i    |    3 | 1677 | 3104 |    0 |    7 |     1 |  0
      4 | l    |    2 | 2512 | 3116 |    2 |    5 |     0 |  1
      5 | l    |    2 | 2512 | 3116 |    4 |    6 |     0 |  1
      6 | l    |    2 | 2512 | 3116 |    5 |    9 |     0 |  1
      7 | i    |    3 | 1677 | 3104 |    3 |   11 |     1 |  0
      8 | r    |    3 | 1677 | 3104 |    0 |    0 |     2 |  2
      9 | l    |    2 | 2512 | 3116 |    6 |   10 |     0 |  1
     10 | l    |    3 | 2512 |  600 |    9 |    0 |     0 |  1
     11 | i    |    3 | 1677 | 3104 |    7 |    0 |     1 |  0
 (11 rows)

  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
  1 | (1,1) |    2512 |       | 20 27 00 00 62 62   | b
  2 | (0,1) |    2512 | (0,1) | 20 27 00 00 61 61   | a
 (2 rows)
  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
  1 | (2,1) |    2512 |       | 20 27 00 00 63 63   | c
  2 | (1,1) |    2512 | (1,1) | 20 27 00 00 62 62   | b
 (2 rows)
  o | ctid  | itemlen | htid |        data         | c
 ---+-------+---------+------+---------------------+---
  1 | (4,1) |    2512 |      | 20 27 00 00 63 63   | c
  2 | (1,0) |       8 |      |                     |
  3 | (2,1) |    2512 |      | 20 27 00 00 62 62   | b
 (3 rows)
  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
  1 | (3,1) |    2512 |       | 20 27 00 00 64 64   | d
  2 | (2,1) |    2512 | (2,1) | 20 27 00 00 63 63   | c
 (2 rows)
  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
  1 | (4,1) |    2512 |       | 20 27 00 00 65 65   | e
  2 | (3,1) |    2512 | (3,1) | 20 27 00 00 64 64   | d
 (2 rows)
  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
  1 | (5,1) |    2512 |       | 20 27 00 00 66 66   | f
  2 | (4,1) |    2512 | (4,1) | 20 27 00 00 65 65   | e
 (2 rows)
  o | ctid  | itemlen | htid |        data         | c
 ---+-------+---------+------+---------------------+---
  1 | (6,1) |    2512 |      | 20 27 00 00 65 65   | e
  2 | (4,0) |       8 |      |                     |
  3 | (5,1) |    2512 |      | 20 27 00 00 64 64   | d
 (3 rows)
  o |  ctid  | itemlen | htid |        data         | c
 ---+--------+---------+------+---------------------+---
  1 | (3,0)  |       8 |      |                     |
  2 | (7,1)  |    2512 |      | 20 27 00 00 63 63   | c
  3 | (11,1) |    2512 |      | 20 27 00 00 65 65   | e
 (3 rows)
  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
  1 | (6,1) |    2512 |       | 20 27 00 00 67 67   | g
  2 | (5,1) |    2512 | (5,1) | 20 27 00 00 66 66   | f
 (2 rows)
  o | ctid  | itemlen | htid  |        data         | c
 ---+-------+---------+-------+---------------------+---
```

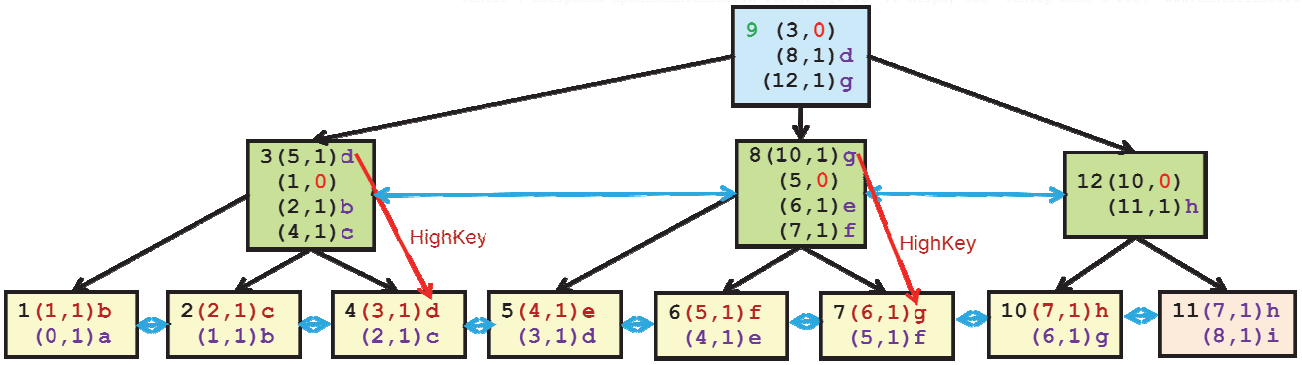
```
 1 | (6,1) |     2512 | (6,1) | 20 27 00 00 67 67   | g
 2 | (7,1) |     2512 | (7,1) | 20 27 00 00 68 68   | h
 3 | (8,1) |     2512 | (8,1) | 20 27 00 00 69 69   | i
(3 rows)
 o |  ctid  | itemlen | htid |        data         | c
---+--------+---------+------+---------------------+---
 1 | (6,0)  |       8 |      |                     |
 2 | (9,1)  |    2512 |      | 20 27 00 00 66 66   | f
 3 | (10,1) |    2512 |      | 20 27 00 00 67 67   | g
(3 rows)
```



10) Посмотрите как изменится структура индекса после перестройки командой REINDEX:

```
postgres=# reindex index t_idx;
 select blkno blk, type, live_items live, btpo_prev prev,  btpo_next next, btpo_level level from
bt_multi_page_stats('t_idx',1,-1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',2);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',3);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',4);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',5);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',6);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',7);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',8);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',9);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',10);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',11);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',12);
 REINDEX
  blk | type | live | prev | next | level
 -----+------+------+------+------+-------
    1 | l    |    2 |    0 |    2 |     0
    2 | l    |    2 |    1 |    4 |     0
    3 | i    |    4 |    0 |    8 |     1
    4 | l    |    2 |    2 |    5 |     0
    5 | l    |    2 |    4 |    6 |     0
    6 | l    |    2 |    5 |    7 |     0
    7 | l    |    2 |    6 |   10 |     0
    8 | i    |    4 |    3 |   12 |     1
    9 | r    |    3 |    0 |    0 |     2
   10 | l    |    2 |    7 |   11 |     0
   11 | l    |    2 |   10 |    0 |     0
   12 | i    |    2 |    8 |    0 |     1
 (12 rows)

 o | ctid  | itemlen | htid  |        data         | c
---+-------+---------+-------+---------------------+---
 1 | (1,1) |    2512 |       | 20 27 00 00 62 62   | b
 2 | (0,1) |    2512 | (0,1) | 20 27 00 00 61 61   | a
(2 rows)

 o | ctid  | itemlen | htid  |        data         | c
```

```
---+-------+---------+-------+---------------------+---
 1 | (2,1) |    2512 |       | 20 27 00 00 63 63   | c
 2 | (1,1) |    2512 | (1,1) | 20 27 00 00 62 62   | b
(2 rows)

 o | ctid  | itemlen | htid |         data         | c
---+-------+---------+------+---------------------+---
 1 | (5,1) |    2512 |      | 20 27 00 00 64 64   | d
 2 | (1,0) |       8 |      |                     |
 3 | (2,1) |    2512 |      | 20 27 00 00 62 62   | b
 4 | (4,1) |    2512 |      | 20 27 00 00 63 63   | c
(4 rows)

 o | ctid  | itemlen | htid  |         data         | c
---+-------+---------+-------+---------------------+---
 1 | (3,1) |    2512 |       | 20 27 00 00 64 64   | d
 2 | (2,1) |    2512 | (2,1) | 20 27 00 00 63 63   | c
(2 rows)

 o | ctid  | itemlen | htid  |         data         | c
---+-------+---------+-------+---------------------+---
 1 | (4,1) |    2512 |       | 20 27 00 00 65 65   | e
 2 | (3,1) |    2512 | (3,1) | 20 27 00 00 64 64   | d
(2 rows)

 o | ctid  | itemlen | htid  |         data         | c
---+-------+---------+-------+---------------------+---
 1 | (5,1) |    2512 |       | 20 27 00 00 66 66   | f
 2 | (4,1) |    2512 | (4,1) | 20 27 00 00 65 65   | e
(2 rows)

 o | ctid  | itemlen | htid  |         data         | c
---+-------+---------+-------+---------------------+---
 1 | (6,1) |    2512 |       | 20 27 00 00 67 67   | g
 2 | (5,1) |    2512 | (5,1) | 20 27 00 00 66 66   | f
(2 rows)

 o |  ctid  | itemlen | htid |         data         | c
---+--------+---------+------+---------------------+---
 1 | (10,1) |    2512 |      | 20 27 00 00 67 67   | g
 2 | (5,0)  |       8 |      |                     |
 3 | (6,1)  |    2512 |      | 20 27 00 00 65 65   | e
 4 | (7,1)  |    2512 |      | 20 27 00 00 66 66   | f
(4 rows)

 o |  ctid  | itemlen | htid |         data         | c
---+--------+---------+------+---------------------+---
 1 | (3,0)  |       8 |      |                     |
 2 | (8,1)  |    2512 |      | 20 27 00 00 64 64   | d
 3 | (12,1) |    2512 |      | 20 27 00 00 67 67   | g
(3 rows)

 o | ctid  | itemlen | htid  |         data         | c
---+-------+---------+-------+---------------------+---
 1 | (7,1) |    2512 |       | 20 27 00 00 68 68   | h
 2 | (6,1) |    2512 | (6,1) | 20 27 00 00 67 67   | g
(2 rows)

 o | ctid  | itemlen | htid  |         data         | c
---+-------+---------+-------+---------------------+---
 1 | (7,1) |    2512 | (7,1) | 20 27 00 00 68 68   | h
 2 | (8,1) |    2512 | (8,1) | 20 27 00 00 69 69   | i
(2 rows)

 o |  ctid  | itemlen | htid |         data         | c
---+--------+---------+------+---------------------+---
 1 | (10,0) |       8 |      |                     |
 2 | (11,1) |    2512 |      | 20 27 00 00 68 68   | h
(2 rows)
```



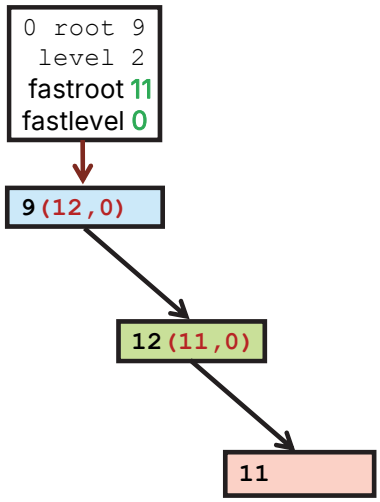
11) Посмотрите, что после удаления всех строк в таблице в индексе число уровней не меняется, а fasroot укажет на листовой блок:

```
postgres=# delete from t;
select root, level, fastroot, fastlevel,
last_cleanup_num_delpages, allequalimage from bt_metap('t_idx');
vacuum t;
select version, root, level, fastroot, fastlevel,
last_cleanup_num_delpages delpages,
last_cleanup_num_tuples tuples, allequalimage from bt_metap('t_idx');
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',9);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',10);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',11);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',12);
DELETE 9
 version | root | level | fastroot | fastlevel | delpages | tuples | allequalimage
---------+------+-------+----------+-----------+----------+--------+---------------
       4 |    9 |     2 |        9 |         2 |        0 |     -1 | t
(1 row)
VACUUM
 version | root | level | fastroot | fastlevel | delpages | tuples | allequalimage
---------+------+-------+----------+-----------+----------+--------+---------------
       4 |    9 |     2 |       11 |         0 |        9 |     -1 | t
(1 row)
 blk | type | live | prev | next | level
-----+------+------+------+------+-------
   1 | d    |    0 |    0 |    2 |     0
   2 | d    |    0 |    0 |    4 |     0
   3 | D    |    0 |    0 |    8 |     1
   4 | d    |    0 |    0 |    5 |     0
   5 | d    |    0 |    0 |    6 |     0
   6 | d    |    0 |    0 |    7 |     0
   7 | d    |    0 |    0 |   10 |     0
   8 | D    |    0 |    0 |   12 |     1
   9 | r    |    1 |    0 |    0 |     2
  10 | d    |    0 |    0 |   11 |     0
  11 | l    |    0 |    0 |    0 |     0
  12 | i    |    1 |    0 |    0 |     1
(12 rows)

 o |  ctid  | itemlen | htid | data
---+--------+---------+------+------
 1 | (12,0) |       8 |      |
(1 row)

NOTICE:  page from block 10 is deleted
 o | ctid | itemlen | htid | data
```

```
---+------+---------+------+------
(0 rows)

 o | ctid | itemlen | htid | data
---+------+---------+------+------
(0 rows)

 o |  ctid  | itemlen | htid | data
---+--------+---------+------+------
 1 | (11,0) |       8 |      |
(1 row)
```

После удаления всех строк fastroot не менялся и блоки не были исключены их структуры индекса.

После вакуумирования 9 блоков было исключено из структуры индекса, осталось 3 правых блока с каждого уровня. `fastroot` указывает на листовой блок 11. С этого блока будет начинаться поиск в индексе.



Часть 10. Очистка блоков индекса при его сканировании

При удалении строки в табличном блоке удаляющая транзакция проставляет свой номер в заголовке строки в поле xmax. При этом признак фиксации не проставляется. При следующем обращении к строке обычно другой процесс сверяется со статусом транзакции в буфере "XLOG Ctl" (устаревшее название "CLOG") и иногда буфере статуса мультитранзакций "Shared MultiXact State". По результатам сверки в заголовке строки таблицы выставляются биты-подсказки о том, что транзакция, удалившая строку была зафиксирована или отменена. При этом на строку могут ссылаться записи индексов, созданных на таблицу. Если доступ к строке таблицы был не через индекс, то индекс никак не обновляется, как и все остальные индексы, созданные на таблицу.

Однако, если процесс при доступе к строке использовал индекс (Index Scan) и процесс обнаружит, что строка (или цепочка строк, на которую ссылается индексная запись) удалена и вышла за горизонт базы, то в индексной записи листового блока (leaf page) в lp_flags процесс установит бит-подсказку LP_DEAD (называют known dead, killed tuple).

LP_DEAD устанавливается в записи того индекса, через который был получен доступ к строке.

LP_DEAD устанавливается независимо от того была ли строка (или цепочка строк, на которую ссылается индексная запись) удалена ранее или этим же процессом.

Вставление этого бита-подсказки называют внутристраничной очисткой в индексах.

В этой части практики вы посмотрите как сильно влияет это на время выполнения запроса.

1) Создайте таблицу, вставьте 1млн.строк, удалите большую часть строк таблицы и соберите статистику для планировщика:

```
postgres=# drop table if exists t;
create table t(id int4 primary key, val text not null) with
(autovacuum_enabled=off);
```

```
insert into t select i,'text number '||i from generate_series(1,1000000) i;
analyze t;
delete from t where id between 501 and 799500;
analyze t;
DROP TABLE
CREATE TABLE
INSERT 0 1000000
ANALYZE
DELETE 799000
ANALYZE
```

Сбор статистики не играет роли, а иллюстрирует то, что планировщик не вводится в заблуждение.

2) Посмотрите данные из блоков индекса:

```
postgres=# select blkno, type, live_items, dead_items, avg_item_size size,
free_size free, btpo_prev prev,  btpo_next next, btpo_level level, btpo_flags fl
from bt_multi_page_stats('t_pkey',1,-1) where type='l' limit 5;
 select dead_items, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by
dead_items;
 select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
 blkno | type | live_items | dead_items | size | free | prev | next | level | fl
-------+------+------------+------------+------+------+------+------+-------+----
     1 | l    |        367 |          0 |   16 |  808 |    0 |    2 |     0 |  1
     2 | l    |        367 |          0 |   16 |  808 |    1 |    4 |     0 |  1
     4 | l    |        367 |          0 |   16 |  808 |    2 |    5 |     0 |  1
     5 | l    |        367 |          0 |   16 |  808 |    4 |    6 |     0 |  1
     6 | l    |        367 |          0 |   16 |  808 |    5 |    7 |     0 |  1
(5 rows)

 dead_items | count
------------+-------
          0 |  2744
(1 row)

 type | count
------+-------
 r    |     1
 l    |  2733
 i    |    10
(3 rows)
```

В индексе нет **LP_DEAD (столбец `dead_items`)** записей.
Число листовых блоков 2733, блоков на промежуточном уровне 10.

2) Чтобы индексные записи были помечены как **`dead_items`** нужно обратиться к строкам с помощью индекса (индексным методом доступа `Index * Scan`). Для этого подходит команда `explain analyze`. Полезно видеть сколько блоков будет прочитано, для этого используется опция `buffers`. Чтобы время выполнения команды было реальным используется опция **`timing off`**. Выполните команду:

```
postgres=# explain (buffers,timing off) select * from t where id between 1 and
800000;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Index Scan using t_pkey on t  (cost=0.42..220.56 rows=2007 width=22)
   Index Cond: ((id >= 1) AND (id <= 800000))
 Planning:
   Buffers: shared hit=27
(4 rows)
```

В разделе `Planning:` строка **`Buffers:`** показывает сколько блоков было прочитано при построении палана.

Выполните команду:

```
postgres=# explain (analyze,buffers,timing off) select * from t where id between 1 and 800000;
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Index Scan using t_pkey on t (cost=0.42..220.58 rows=2008 width=22) (actual rows=1000 loops=1)
   Index Cond: ((id >= 1) AND (id <= 800000))
   Buffers: shared hit=9501
 Planning:
   Buffers: shared hit=8
 Planning Time: 0.173 ms
 Execution Time: 187.850 ms
(7 rows)
```

Число буферов, прочитанных для построения плана уменьшилось с 27 до 8 так как данные из таблиц системного каталога были закэшированы в локальной памяти серверного процесса в кэше таблиц системного каталога (структура памяти называется CatalogCache).

Биты-подсказки в любом случае устанавливаются и грязнятся. Если немного подождать перед выполнением команды `explain`, то результат будет следующим (число блоков **`dirtied`** будет *произвольным*):

```
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Index Scan using t_pkey on t  (cost=0.42..220.58 rows=2008 width=22) (actual rows=1000 loops=1)
   Index Cond: ((id >= 1) AND (id <= 800000))
   Buffers: shared hit=9501 dirtied=7306
 Planning:
   Buffers: shared hit=20 dirtied=1
 Planning Time: 0.167 ms
 Execution Time: 223.348 ms
(7 rows)
```

Использовалось сканирование индекса. Табличные методы доступа (`Seq Scan`) и `Bitmap Index Scan` не очищают блоки индекса, очищает только индексное сканирование.

Число прочитанных блоков **9501**. Из них большая часть относится к таблице (в индексе всего `2744` блоков, а читалась только часть листовых блоков) - запрос проверял по таблице удалена ли строка в блоке таблицы. Если индексная ссылка вела на удаленную строку в таблице, то индексная запись помечалась как LP_DEAD.

Время выполнения запроса **187** миллисекунд.

После выполнения этого запроса записи в блоках индекса были помечены битом-подсказкой LP_DEAD (known dead). Этот бит позволяет игнорировать эту индексную запись и не перепроверять действительно ли надо игнорировать запись путем обращения к блоку таблицы.

3) Проверьте появились ли **`dead_items`** записи:

```
postgres=# select blkno, type, live_items, dead_items, avg_item_size size,
free_size free, btpo_prev prev,  btpo_next next, btpo_level level, btpo_flags fl
from bt_multi_page_stats('t_pkey',1,-1) where type='l' limit 5;
 select dead_items, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by
dead_items;
 select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
 blkno | type | live_items | dead_items | size | free | prev | next | level | fl
-------+------+------------+------------+------+------+------+------+-------+----
     1 | l    |        367 |          0 |   16 |  808 |    0 |    2 |     0 |  1
     2 | l    |        135 |        232 |   16 |  808 |    1 |    4 |     0 | 65
     4 | l    |          1 |        366 |   16 |  808 |    2 |    5 |     0 | 65
     5 | l    |          1 |        366 |   16 |  808 |    4 |    6 |     0 | 65
     6 | l    |          1 |        366 |   16 |  808 |    5 |    7 |     0 | 65
(5 rows)
```

```
 dead_items | count
------------+-------
        156 |     1
        232 |     1
        366 |  2182
          0 |   560
(4 rows)

 type | count
------+-------
 r    |     1
 l    |  2733
 i    |    10
(3 rows)
```

Много индексных записей было помечено битом-подсказкой LP_DEAD (known dead). Блоков исключённых из структуры индекса не появилось.

4) Выполните запрос ещё раз:

```
postgres=# explain (analyze,buffers,timing off) select * from t where id between 1 and 800000;
                                    QUERY PLAN
---------------------------------------------------------------------------------------------
 Index Scan using t_pkey on t (cost=0.42..220.58 rows=2008 width=22) (actual rows=1000 loops=1)
   Index Cond: ((id >= 1) AND (id <= 800000))
   Buffers: shared hit=2196
 Planning:
   Buffers: shared hit=8
 Planning Time: 0.114 ms
 Execution Time: 2.522 ms
(7 rows)
```

Время выполнения запроса уменьшилось с 187 до 2 миллисекунд.
Запрос выполнялся 187 миллисекунд потому, что вносил изменения в записи блоков, а также потому что считывал блоки таблицы. Установка флага LP_DEAD происходит однократно на мастере.
На физической реплике индексные записи не помечаются флагом LP_DEAD, так как реплика не может менять содержимое блоков локальных объектов баз данных.
Изменения в битах-подсказках LP_DEAD не журналируются и не передаются на реплику.
Однако, **после удаления или изменения большого числа строк в таблице запросы, которые обычно выполнялись быстро могут один раз (на мастере) выполняться в сотни раз (в примере в 187/2=93 раза) дольше.**
На **реплике запросы будут выполняться медленнее всё время до автовакуума** на мастере.
Если до запроса к таблице успеет обратится автовакуум, то задержку испытает автовакуум.
Существенная задержка выполнения запроса может быть неожиданной для клиентского приложения. чтобы такого не происходило, после массового удаления или изменения строк стоит выполнить вакуумирование таблицы.

При повторных выполнениях запроса читается намного меньше блоков. Если повторить запрос, число прочитанных блоков будет таким же: 2196.

5) Выполните вакуумирование таблицы и её индекса:

```
postgres=# vacuum analyze t;
VACUUM
```

6) Посмотрите статистику по страницам индекса:

```
postgres=# select blkno, type, live_items, dead_items, avg_item_size size,
free_size free, btpo_prev prev,  btpo_next next, btpo_level level, btpo_flags fl
from bt_multi_page_stats('t_pkey',1,-1) where type='l' limit 5;
 select dead_items, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by
dead_items;
 select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
 blkno | type | live_items | dead_items | size | free | prev | next | level | fl
-------+------+------------+------------+------+------+------+------+-------+----
     1 | l    |        367 |          0 |   16 |  808 |    0 |    2 |     0 |  1
     2 | l    |        135 |          0 |   16 | 5448 |    1 |  286 |     0 |  1
   286 | l    |          1 |          0 |   16 | 8128 |    2 | 2194 |     0 |  1
  2194 | l    |        211 |          0 |   16 | 3928 |  286 | 2195 |     0 |  1
  2195 | l    |        367 |          0 |   16 |  808 | 2194 | 2196 |     0 |  1
(5 rows)

 dead_items | count
------------+-------
          0 |  2744
(1 row)
 type | count
------+-------
 D    |     6
 r    |     1
 l    |   552
 i    |     4
 d    |  2181
(5 rows)
```

Помеченные флагом LP_DEAD индексные записи были удалены вакуумом из блоков индекса.
Типы блоков: `d` - удалёный листовой (deleted leaf) блок, `D` - удалённый внутренний (deleted internal) блок.
Блоки в которых **все записи были помечены как удалённые (при вакуумировании или ранее при Index Scan) исключаются из структуры индекса при вакуумировании**. Такие блоки не сканируются запросами, но они считываются при вакуумировании, так как у индексов нет карты видимости и считываются все блоки.

7) Выполните запрос:

```
postgres=# explain (analyze,buffers,timing off) select * from t where id between
1 and 800000;
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Index Scan using t_pkey on t (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
   Index Cond: ((id >= 1) AND (id <= 800000))
   Buffers: shared hit=15
 Planning:
   Buffers: shared hit=16
 Planning Time: 0.108 ms
 Execution Time: 0.284 ms
(7 rows)
```

Вакуумирование очистило блоки индекса. Число читаемых запросом блоков уменьшилось с 2196 до 15. Значение "shared hit=16" это число буферов, прочитанных для создания плана и в них находятся, в основном, блоки таблиц системного каталога. Почему в разделе "Planning:" число буферов увличилось с 8 до 16? Потому, что была собрана статистика командой **vacuum analyze** и планировщик её считал для создания плана.

8) Посмотрите размер индекса:

```
postgres=# select pg_indexes_size('t');
```

Вакуумирование не уменьшило размер индекса, так как в конце файла индекса есть блоки с записями и размер файла индекса не уменьшился.

9) Перестройте индекс и проверьте уменьшится ли размер индекса:

```
postgres=# reindex table t;
analyze t;
select pg_indexes_size('t');
pg_indexes_size
-----------------
        22487040
(1 row)


REINDEX
ANALYZE
 pg_indexes_size
-----------------
         4538368
(1 row)
```

После перестроения индекса размер файла индекса уменьшился.

10) Выполните запрос:

```
postgres=# explain (analyze,buffers,timing off) select * from t where id between
1 and 800000;
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Index Scan using t_pkey on t (cost=0.42..132.56 rows=2007 width=22) (actual rows=1000 loops=1)
   Index Cond: ((id >= 1) AND (id <= 800000))
   Buffers: shared hit=11 read=2
 Planning:
   Buffers: shared hit=17 read=s4
 Planning Time: 0.201 ms
 Execution Time: 0.346 ms
(7 rows)
```

Размер индекса уменьшился в 5 раз. Стоимость того же самого плана уменьшилась с 220.58 до 132.56. При этом время выполнения не улучшилось. Физические чтения двух блоков не влияют на время выполнения запроса. Если несколько раз выполнять запрос до перестройки и после перестройки индекса, то можно увидеть, что среднее время выполнения запроса одинаково, а значит перестройка индекса никак не повлияла на время выполнения запроса.

Почему не повлияло? Файлы индекса могут включать в себя блоки, которые исключены из структуры индекса. Исключенные из структуры индекса блоки учитываются в размере индекса. Последние два запроса с разной стоимостью, но одинаковым временем выполнения скорее всего сканировали одинаковое число листовых блоков индекса, что и определяет время выполнения запроса.

При вакуумировании читаются все блоки файлов индекса по порядку от начала файла до конца каждого файла. Уменьшение размера файлов индекса ускоряет работу автовакуума и уменьшает число пустых блоков индекса, которые вынужден подгружать в буферный кэш автовакуум.

После удаления большого числа строк стоит перестроить индексы на таблице, если нужно ускорить работу автовакуума с этой таблицей.

А при одномоментном удалении большого числа строк более оптимально удалить индексы перед удалением строк, затем удалить строки, потом создать индексы, потом выполнить вакуум и собрать статистику. При отсутствии индексов удаление строк выполниться существенно (на порядки) быстрее.

Часть 11. Медленное выполнение запросов из-за отсутствия очистки блоков индекса на реплике

1) Создайте физическую реплику:

```
postgres@tantor:~$ pg_ctl stop -D /var/lib/postgresql/backup/1 -m immediate
```

```
psql -c "select pg_drop_replication_slot('replica1');"
rm -rf /var/lib/postgresql/backup/1
pg_basebackup -D /var/lib/postgresql/backup/1 -P -R -c fast -C --slot=replica1
echo "port=5433" >> /var/lib/postgresql/backup/1/postgresql.auto.conf
pg_ctl start -D /var/lib/postgresql/backup/1

ERROR:  replication slot "replica1" does not exist
...
server started
postgres@tantor:~$
```

Если появилось сообщение `server started`, то реплика работает на порту 5433.
В терминал будут выдаваться диагностические сообщения, чтобы вернуть терминал достаточно нажать на клавиатуре **<ENTER>.**

2) Запустите psql и выполните команды на экземпляре мастера:

```
postgres@tantor:~$ psql
postgres=# drop table if exists t;
create table t(id int4 primary key, val text not null) with
(autovacuum_enabled=off);
insert into t select i,'text number '||i from generate_series(1,1000000) i;
analyze t;
delete from t where id between 501 and 799500;
analyze t;
DROP TABLE
CREATE TABLE
INSERT 0 1000000
ANALYZE
DELETE 799000
ANALYZE
```

3) В новом окне терминала запустите psql и подсоединитесь к реплике:

```
postgres@tantor:~$ psql -p 5433
```

4) Выполните несколько раз запрос на реплике:

```
postgres=# explain (analyze,buffers,timing off) select * from t where id between
1 and 800000;
explain (analyze,buffers,timing off) select * from t where id between 1 and
800000;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using t_pkey on t  (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
   Index Cond: ((id >= 1) AND (id <= 800000))
   Buffers: shared hit=7317
 Planning:
   Buffers: shared hit=63 read=16
 Planning Time: 4.032 ms
 Execution Time: 341.255 ms
(7 rows)

                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using t_pkey on t  (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
   Index Cond: ((id >= 1) AND (id <= 800000))
   Buffers: shared hit=7317
 Planning:
   Buffers: shared hit=8
 Planning Time: 0.104 ms
 Execution Time: 345.192 ms
(7 rows)
```

При каждом выполнении до первого запроса на мастере запрос на реплике выполняется долго: ~345 миллисекунд.

5) Выполните запрос на мастере:

```
postgres=# explain (analyze,buffers,timing off) select * from t where id between
1 and 800000;
                                              QUERY PLAN
-----------------------------------------------------------------------------------------------
 Index Scan using t_pkey on t  (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
   Index Cond: ((id >= 1) AND (id <= 800000))
   Buffers: shared hit=9501 dirtied=2636
 Planning:
   Buffers: shared hit=26 dirtied=1
 Planning Time: 0.190 ms
 Execution Time: 189.604 ms
(7 rows)
```

Первый запрос на мастере выполнялся `189.604 ms`, второй и последующие запросы выполнятся быстро, за ~2мс. Точно так же как и в отсутствии реплики, как в предыдущей части практики.

6) После того как на мастере выполнится первый раз запрос, на реплике запросы станут выполняться за ~120мс. Выполните на реплике два раза запрос:

```
postgres=# explain (analyze,buffers,timing off) select * from t where id between
1 and 800000;
explain (analyze,buffers,timing off) select * from t where id between 1 and
800000;
                                              QUERY PLAN
-----------------------------------------------------------------------------------------------
 Index Scan using t_pkey on t  (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
   Index Cond: ((id >= 1) AND (id <= 800000))
   Buffers: shared hit=7317
 Planning:
   Buffers: shared hit=8
 Planning Time: 0.086 ms
 Execution Time: 122.248 ms
(7 rows)

                                              QUERY PLAN
-----------------------------------------------------------------------------------------------
 Index Scan using t_pkey on t  (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
   Index Cond: ((id >= 1) AND (id <= 800000))
   Buffers: shared hit=7317
 Planning:
   Buffers: shared hit=8
 Planning Time: 0.132 ms
 Execution Time: 116.896 ms
(7 rows)
```

Первый запрос на мастере выполняется за ~190мс, последующие за ~2мс.
После выполнения первого запроса на мастере, запросы на реплике станут выполняться немного быстрее: за ~120мс. Разница из-за того, что на мастере вносятся изменения в блоки индекса: устанавливаются биты LP_DEAD. На реплике изменения в блоки индекса не вносятся. Большая часть времени запроса тратится на обращения к блокам таблицы.
До вакуумирования запросы на репликах будут выполняться дольше, чем даже на мастере.

7) Выполните на мастере запрос и вакуумирование таблицы t:

```
postgres=# explain (analyze,buffers,timing off) select * from t where id between
1 and 800000;
vacuum analyze t;
select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
explain (analyze,buffers,timing off) select * from t where id between 1 and
800000;

                                              QUERY PLAN
-----------------------------------------------------------------------------------------------
 Index Scan using t_pkey on t  (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
   Index Cond: ((id >= 1) AND (id <= 800000))
```

```
   Buffers: shared hit=2196
 Planning:
   Buffers: shared hit=8
 Planning Time: 0.125 ms
 Execution Time: 2.234 ms
(7 rows)
VACUUM
 type | count
------+-------
 D    |     6
 r    |     1
 l    |   552
 i    |     4
 d    |  2181
(5 rows)
                                          QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Index Scan using t_pkey on t  (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
   Index Cond: ((id >= 1) AND (id <= 800000))
   Buffers: shared hit=15
 Planning:
   Buffers: shared hit=16 dirtied=1
 Planning Time: 0.123 ms
 Execution Time: 0.286 ms
(7 rows)
```

Запрос на мастере нужен, чтобы убедиться, что второй запрос на мастере будет выполнен за 2мс.

После вакуумирования большинство блоков были исключены из структуры индекса и помечены как имеющие тип `type=d` или `D`.

После вакуумирования запрос начнет выполняться на порядок быстрее: за 0.286мс.

8) Выполните на реплике два раза запрос:

```
postgres=# explain (analyze,buffers,timing off) select * from t where id between 1 and 800000;
explain (analyze,buffers,timing off) select * from t where id between 1 and 800000;
select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
                                          QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Index Scan using t_pkey on t  (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
   Index Cond: ((id >= 1) AND (id <= 800000))
   Buffers: shared hit=15
 Planning:
   Buffers: shared hit=19
 Planning Time: 0.164 ms
 Execution Time: 0.291 ms
(7 rows)

                                          QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Index Scan using t_pkey on t  (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
   Index Cond: ((id >= 1) AND (id <= 800000))
   Buffers: shared hit=15
 Planning:
   Buffers: shared hit=8
 Planning Time: 0.071 ms
 Execution Time: 0.307 ms
(7 rows)

 type | count
------+-------
 D    |     6
 r    |     1
 l    |   552
 i    |     4
 d    |  2181
(5 rows)
```

После вакуума на мастере время выполнения запросов на реплике такая же как на мастере: ~0.3мс.

9) Удалите физическую реплику, чтобы освободить место на диске:

```
postgres@tantor:~$ pg_ctl stop -D /var/lib/postgresql/backup/1 -m immediate
psql -c "select pg_drop_replication_slot('replica1');"
rm -rf /var/lib/postgresql/backup/1
```

Время выполнения в примере уменьшилось в тысячу раз: с 345мс до 0.3мс.
Своевременная отработка автовакуума на мастере влияет на время выполнения запросов на репликах. До отработки автовакуума или вакуумирования таблицы, в которой было удалено много строк, запросы на реплике выполняются существенной медленнее, чем могли бы.

Часть 12. Определение числа удаленных строк

Как определить число удалённых строк до вакуумирования? Анализ и автоанализ такую статистику по индексам не собирают.

1) Выполните команды на экземпляре мастера:

```
postgres@tantor:~$ psql
postgres=# drop table if exists t;
create table t(id int4 primary key, val text not null) with
(autovacuum_enabled=off);
insert into t select i,'text number '||i from generate_series(1,1000000) i;
analyze t;
delete from t where id between 501 and 799500;
analyze t;
DROP TABLE
CREATE TABLE
INSERT 0 1000000
ANALYZE
DELETE 799000
ANALYZE
```

Эти команды пересоздают таблицу t и создают большое число удалённых строк.

2) Функция `bt_multi_page_stats('t_pkey',1,-1)` расширения `pageinspect` и функция `pgstatindex('t_pkey')` расширения `pgstattuple` покажут удалённые блоки только после вакуумирования. Пример выполнения функций до вакуумирования и после:

```
postgres=# create extension pgstattuple;
CREATE EXTENSION
postgres=# select tree_level levels, index_size, internal_pages i_pages, leaf_pages
leaf_pages, empty_pages e, deleted_pages, avg_leaf_density dense, leaf_fragmentation frag
FROM pgstatindex('t_pkey');

 levels | index_size | i_pages | leaf_pages | e | deleted_pages | dense | frag
--------+------------+---------+------------+---+---------------+-------+------
      2 |   22487040 |      11 |       2733 | 0 |             0 | 90.06 |    0
(1 row)

postgres=# select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
type | count
-----+-------
r    |     1
l    |  2733
i    |    10
(3 rows)

postgres=# vacuum t;
VACUUM
postgres=# select tree_level levels, index_size, internal_pages i_pages, leaf_pages
leaf_pages, empty_pages e, deleted_pages deleted, avg_leaf_density dense,
```

```
leaf_fragmentation frag  FROM pgstatindex('t_pkey');
 levels | index_size | i_pages | leaf_pages | e | deleted_pages | dense | frag
--------+------------+---------+------------+---+---------------+-------+------
      2 |   22487040 |       5 |        552 | 0 |          2187 | 89.63 |    0
(1 row)

postgres=# select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
type | count
-----+------
D    |     6
r    |     1
l    |   552
i    |     4
d    |  2181
(5 rows)
```

3) Строки в индексе будут помечены удаленными только после первого запроса с методом доступа `Index Scan` на мастере. Такие запросы могут не обратиться ко всем удаленным строкам, трудоёмки и могут отсутствовать на мастере, так как скорее всего перенесены на реплики. Пример после запроса, если его выполнить после команд первого пункта этой части практики (выполнять эти запросы не обязательно):

```
 postgres=# select sum(live_items), sum(dead_items) from
bt_multi_page_stats('t_pkey',1,-1);
    sum    | sum
 ----------+-----
  1005484 |   0
 (1 row)

 postgres=# explain (analyze,buffers,timing off) select * from t where id between
1 and 800000;
 ...
 postgres=# select sum(live_items) live, sum(dead_items) deleted from
bt_multi_page_stats('t_pkey',1,-1);
   live   | deleted
 ---------+--------
  206484 | 799000
 (1 row)
```

4) Как определить число удалённых строк до вакуумирования и без запроса к таблице на мастере? Это можно сделать косвенно по статистике числа удаленных строк в таблице:

```
 postgres=# drop table if exists t;
 create table t(id int4 primary key, val text not null) with
(autovacuum_enabled=off);
 insert into t select i,'text number '||i from generate_series(1,1000000) i;
 delete from t where id between 501 and 799500;
 DROP TABLE
 CREATE TABLE
 INSERT 0 1000000
 ANALYZE
 DELETE 799000
 postgres=# select relname, n_tup_ins, n_tup_upd, n_tup_del, n_live_tup,
n_dead_tup from pg_stat_all_tables where relname='t';
  relname | n_tup_ins | n_tup_upd | n_tup_del | n_live_tup | n_dead_tup
 ---------+-----------+-----------+-----------+------------+------------
  t       |   1000000 |         0 |    799000 |     201000 |     799000
 (1 row)
```

Рабочая статистика собирается серверными процессами и не требует ни анализа, ни вакуума.

Эта часть практики нужна, чтобы запомнить важность столбца `n_dead_tup` рабочей статистики, выдаваемой представлением `pg_stat_all_tables`.

Функции расширения `pgstattuple` имеют многообещающие названия, их пытаются использовать для определения раздувания таблиц (bloat), но по большей части функции этого расширения оказываются бесполезными.

Часть 13. Поиск по структуре индекса btree

В 9 части этой практики рассматривалась структура индекса и его графическое отображение. Чтобы запомнить, как по результатам запроса рисовать структуру индекса, стоит нарисовать структуру самостоятельно. Отрисовка структуры помогает понять (визуально оценить) насколько трудоёмок поиск по индексу для серверных процессов.

1) Если вы успеваете выполнять практические задания, то можете попробовать нарисовать структуру следующего индекса:

```
drop table if exists t;
create table t(s text storage plain) with (autovacuum_enabled=off,
fillfactor=10);
create index t_idx on t (s) with (deduplicate_items = off);
insert into t values (repeat('a',1800));
insert into t values (repeat('b',1800));
insert into t values (repeat('c',1800));
insert into t values (repeat('d',1800));
insert into t values (repeat('e',1800));
insert into t values (repeat('f',1800));
insert into t values (repeat('g',1800));
insert into t values (repeat('h',1800));
insert into t values (repeat('i',1800));
insert into t values (repeat('j',1800));
insert into t values (repeat('k',1800));
insert into t values (repeat('l',1800));
insert into t values (repeat('m',1800));
insert into t values (repeat('n',1800));
insert into t values (repeat('o',1800));
insert into t values (repeat('p',1800));
insert into t values (repeat('q',1800));

select blkno, type, live_items live, avg_item_size size, free_size free,
btpo_prev prev,  btpo_next next, btpo_level level, btpo_flags fl from
bt_multi_page_stats('t_idx',1,-1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',2);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',3);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',4);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',5);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',6);
```

```
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',7);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',8);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',9);
```

По результату первого запроса можно нарисовать каркас структуры (блоки индекса):

```
 blkno | type | live | size | free | prev | next | level | fl
-------+------+------+------+------+------+------+-------+----
     1 | l    |    4 | 1816 |  868 |    0 |    2 |     0 |  1
     2 | l    |    4 | 1816 |  868 |    1 |    4 |     0 |  1
     3 | i    |    4 | 1364 | 2676 |    0 |    8 |     1 |  0
     4 | l    |    4 | 1816 |  868 |    2 |    5 |     0 |  1
     5 | l    |    4 | 1816 |  868 |    4 |    6 |     0 |  1
     6 | l    |    4 | 1816 |  868 |    5 |    7 |     0 |  1
     7 | l    |    2 | 1816 | 4508 |    6 |    0 |     0 |  1
     8 | i    |    3 | 1213 | 4496 |    3 |    0 |     1 |  0
     9 | r    |    2 |  912 | 6316 |    0 |    0 |     2 |  2
(9 rows)
```

Стоит начать с корневого блока (type=r), дальше разметить уровни (level) и нарисовать прямоугольники блоков. Пронумеровать блоки (blkno) по каждому уровню, учитывая какой блок левее (prev) и какой правее (next).

В индексе три уровня, 9 блоков. На промежуточном уровне 2 блока, на листовом уровне 6 блоков.

Содержимое каждого блока:

```
 o | ctid  | itemlen | htid  |        data         | c
---+-------+---------+-------+---------------------+---
 1 | (3,1) |    1816 |       | 30 1c 00 00 64 64   | d
 2 | (0,1) |    1816 | (0,1) | 30 1c 00 00 61 61   | a
 3 | (1,1) |    1816 | (1,1) | 30 1c 00 00 62 62   | b
 4 | (2,1) |    1816 | (2,1) | 30 1c 00 00 63 63   | c
(4 rows)

 o | ctid  | itemlen | htid  |        data         | c
---+-------+---------+-------+---------------------+---
 1 | (6,1) |    1816 |       | 30 1c 00 00 67 67   | g
 2 | (3,1) |    1816 | (3,1) | 30 1c 00 00 64 64   | d
 3 | (4,1) |    1816 | (4,1) | 30 1c 00 00 65 65   | e
 4 | (5,1) |    1816 | (5,1) | 30 1c 00 00 66 66   | f
(4 rows)

 o | ctid  | itemlen | htid |        data         | c
---+-------+---------+------+---------------------+---
 1 | (5,1) |    1816 |      | 30 1c 00 00 6a 6a   | j
 2 | (1,0) |       8 |      |                     |
 3 | (2,1) |    1816 |      | 30 1c 00 00 64 64   | d
 4 | (4,1) |    1816 |      | 30 1c 00 00 67 67   | g
(4 rows)

 o | ctid  | itemlen | htid  |        data         | c
---+-------+---------+-------+---------------------+---
 1 | (9,1) |    1816 |       | 30 1c 00 00 6a 6a   | j
```

```
 2 | (6,1) |    1816 | (6,1) | 30 1c 00 00 67 67  | g
 3 | (7,1) |    1816 | (7,1) | 30 1c 00 00 68 68  | h
 4 | (8,1) |    1816 | (8,1) | 30 1c 00 00 69 69  | i
(4 rows)

 o |  ctid  | itemlen |  htid  |        data         | c
---+--------+---------+--------+---------------------+---
 1 | (12,1) |    1816 |        | 30 1c 00 00 6d 6d   | m
 2 | (9,1)  |    1816 | (9,1)  | 30 1c 00 00 6a 6a   | j
 3 | (10,1) |    1816 | (10,1) | 30 1c 00 00 6b 6b   | k
 4 | (11,1) |    1816 | (11,1) | 30 1c 00 00 6c 6c   | l
(4 rows)

 o |  ctid  | itemlen |  htid  |        data         | c
---+--------+---------+--------+---------------------+---
 1 | (15,1) |    1816 |        | 30 1c 00 00 70 70   | p
 2 | (12,1) |    1816 | (12,1) | 30 1c 00 00 6d 6d   | m
 3 | (13,1) |    1816 | (13,1) | 30 1c 00 00 6e 6e   | n
 4 | (14,1) |    1816 | (14,1) | 30 1c 00 00 6f 6f   | o
(4 rows)

 o |  ctid  | itemlen |  htid  |        data         | c
---+--------+---------+--------+---------------------+---
 1 | (15,1) |    1816 | (15,1) | 30 1c 00 00 70 70   | p
 2 | (16,1) |    1816 | (16,1) | 30 1c 00 00 71 71   | q
(2 rows)

 o | ctid  | itemlen | htid |        data         | c
---+-------+---------+------+---------------------+---
 1 | (5,0) |       8 |      |                     |
 2 | (6,1) |    1816 |      | 30 1c 00 00 6d 6d   | m
 3 | (7,1) |    1816 |      | 30 1c 00 00 70 70   | p
(3 rows)

 o | ctid  | itemlen | htid |        data         | c
---+-------+---------+------+---------------------+---
 1 | (3,0) |       8 |      |                     |
 2 | (8,1) |    1816 |      | 30 1c 00 00 6a 6a   | j
(2 rows)
```

Связи между блоками (обозначаются стрелками) рисуются по столбцу ctid. В листовых блоках ctid указывает на строку в таблице по которой создан индекс.

Во внутренних блоках и корневом блоке первое число в ctid указывает на номер блока индекса, а второе число единица или ноль. Если единица, то в поле data хранится минимальное значение, которое присутствует в дочернем листовом блоке. Если ноль `ctid=(N,0)`, то поле `data` пусто (**трактуется как "минус бесконечность", то есть граница неизвестна**) и эта ссылка ведёт на самый "левый" дочерний блок.

Поле `htid` добавляет "избыточности": в листовых блоках дублирует значение поля `ctid`. Различия могут появляться в процессе внесения изменения в записи индекса и используется для идентификации того, что записи в процессе изменения без введения блокировок (арбитража). Структура индекса btree оптимизирована для параллельной работы процессов при минимуме блокировок, при этом гарантируя непротиворечивость.

Если вы не сможете нарисовать структуру индекса, то можете повторить (уже осмысленно) 9 часть практики.

2) От примера в 9 части этой практики индекс в этой части практики отличается тем, что в листовом блоке помещается до трёх ссылок на строки таблицы, а не одна ссылка. И этот индекс позволяет убедиться, что во внутренних блоках хранятся самые "левые" (меньшие) значения,

которые присутствуют во второй записи (первая запись это HighKey) всех листовых блоков, кроме самого правого блока. Процесс, который ищет значения `e, f, h,i, n, o, q` будет ориентироваться на диапазон ("вилку") между значением "левее" искомого и HighKey, так как эти значения отсутствуют в корневом и внутренних (промежуточных) блоках индекса.

После того как структура индекса нарисована попробуйте найти по структуре значения e, f, h,i, n, o, q. Благодаря простоте структуры индекса вы визуально оцените как процессы выполняют поиск по индексам btree и "трудоёмкость" поиска, представляя себе, что операции сравнения, выборки записей, прохождение по ссылке на соседний блок используют ресурсы ядра процессора.

Практика к главе 8

Часть 1. TOAST

1) TOAST поддерживают типы данных переменной длинны, называемые `varlena` (`pg_type.typlen=-1`). Поля фиксированной длины не могут храниться вне блока таблицы, так как для этих типов данных не написан код, реализующий хранение вне блока таблицы (в TOAST-таблице).
Для большинства типов переменной длинны по умолчанию используется режим EXTENDED, кроме типов:

```
postgres=# select distinct typname, typalign, typstorage, typcategory, typlen
from pg_type where typtype='b' and typlen<0 and typstorage<>'x' order by typname;
   typname    | typalign | typstorage | typcategory | typlen
--------------+----------+------------+-------------+--------
 cidr         | i        | m          | I           |     -1
 gtsvector    | i        | p          | U           |     -1
 inet         | i        | m          | I           |     -1
 int2vector   | i        | p          | A           |     -1
 numeric      | i        | m          | N           |     -1
 oidvector    | i        | p          | A           |     -1
 tsquery      | i        | p          | U           |     -1
(7 rows)
```

Режим EXTENDED сначала пытается сжать поле и только потом решается выносить поле в TOAST или не выносить. Режим EXTERNAL сразу выносит поле в TOAST, оставляя в блоке 18 байт.

2) Вынесенные в TOAST поля делятся на части - "чанки". Деление на чанки происходит после сжатия, если сжатие применялось.
Максимальный размер чанка хранится в управляющем файле кластера, его можно посмотреть утилитой `pg_controldata`:

```
postgres@tantor:~$ pg_controldata | grep TOAST
Maximum size of a TOAST chunk:        1996
```

Если размер поля (после сжатия если оно применяется) превышает `1996` байт, то поле делится на два и более чанка, первый чанк имеет размер 1996 байт. `1996` байт является максимальным размером чанка в СУБД Tantor и PostgreSQL (на 64-разрядных платформах).

Для Astralinux PostgreSQL значение не `1996` байт, а `1988` байт:

```
postgres@tantor:~$ /usr/lib/postgresql/15/bin/pg_controldata -D
/var/lib/postgresql/15/main | grep TOAST
Maximum size of a TOAST chunk:        1988
```

У PostgreSQL на 32-разрядных платформах максимальный размер чанка 2000 байт.

3) Посмотрим, при каком условии поля начинают вытесняться в TOAST.
Создайте таблицу с двумя столбцами, вставьте строки и посмотрите размер строки:

```
postgres=# drop table if exists t;
create table t (id int4, c text storage external);
insert into t VALUES (1, repeat('a',2000));
insert into t VALUES (1, repeat('a',2000));
insert into t VALUES (1, repeat('a',2000));
insert into t VALUES (1, repeat('a',2000));
select pg_column_size(t.*) from t;
```

```
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
DROP TABLE
CREATE TABLE
ALTER TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
 pg_column_size
----------------
           2032
           2032
           2032
           2032
(4 rows)
ERROR:  block number 0 is out of range for relation "pg_toast_546660"
```

Поля не были вынесены в TOAST, так как размер строки не превысил TOAST_TUPLE_THRESHOLD=2032 байта.

Значение TOAST_TUPLE_THRESHOLD=2032 байта установлено в исходном коде PostgreSQL и не переопределяется параметрами. Если формируемая (INSERT, UPDATE, COPY) строка таблицы превышает 2032 байта, то выполняется код алгоритма сжатия и/или выноса полей в TOAST-таблицу.

4) Увеличьте размер текстовых полей на единицу и повторите команды:

```
postgres=# drop table if exists t;
create table t (id int4, c text storage external);
insert into t VALUES (1, repeat('a',2001));
insert into t VALUES (1, repeat('a',2001));
insert into t VALUES (1, repeat('a',2001));
insert into t VALUES (1, repeat('a',2001));
select pg_column_size(t.*) from t;
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
DROP TABLE
CREATE TABLE
ALTER TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
 pg_column_size
----------------
           2033
           2033
           2033
           2033
(4 rows)

 lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+--------
  1 |   6152 |   2032 | (0,1)  |     24
  2 |   6104 |     41 | (0,2)  |     24
  3 |   4072 |   2032 | (0,3)  |     24
  4 |   4024 |     41 | (0,4)  |     24
  5 |   1992 |   2032 | (0,5)  |     24
  6 |   1944 |     41 | (0,6)  |     24
(6 rows)
```

Все четыре текстовых поля четырёх строк были вытеснены в TOAST.

Максимальный размер строки в таблице TOAST не может превысить TOAST_TUPLE_THRESHOLD=2032 байта.

Текстовое поле в этой таблице вытесняется начиная с размера 2001 байт. Критерием начала или продолжения вытеснения полей является то, что pg_column_size(t.*)>2032. При этом значения полей будут сжиматься (если режим хранения поля EXTENDED) и затем резаться на чанки (части) по 1996 байт. В этом примере для текстового поля поля длинной 2001 байта появится чанк небольшого размера в 41 байт, который будет вставлен серверным процессом в блок с чанками 2032 байт. Из-за этого в блоке таблицы TOAST поместится только 3 чанка по 2032 байт и будет много свободного места (1944 - 24 байт заголовка блока - 6 слотов заголовка блока *4 байта=1896 байт).

5) Длинна поля-кандидата на вытеснение зависит от длинны строки. Поля для вытеснения выбираются по алгоритму, описанному в главе.
Рассмотрим ещё пример с таблицей, из одного столбца:

```
postgres=# drop table if exists t;
create table t (c text storage external);
insert into t VALUES (repeat('a',2005));
insert into t VALUES (repeat('a',2005));
insert into t VALUES (repeat('a',2005));
insert into t VALUES (repeat('a',2005));
select pg_column_size(t.*) from t;
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
DROP TABLE
CREATE TABLE
ALTER TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
 pg_column_size
----------------
           2033
           2033
           2033
           2033
(4 rows)

 lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+--------
  1 |   6152 |   2032 | (0,1)  |     24
  2 |   6104 |     45 | (0,2)  |     24
  3 |   4072 |   2032 | (0,3)  |     24
  4 |   4024 |     45 | (0,4)  |     24
  5 |   1992 |   2032 | (0,5)  |     24
  6 |   1944 |     45 | (0,6)  |     24
(6 rows)
```

Текстовое поле в этой таблице вытесняется начиная с размера 2005 байт. Критерием начала вытеснения полей всё также является то, что pg_column_size(t.*)>2032.

Вытесняемое поле всегда режется на чанки по 1996 байт.
В этом примере для каждого текстового поля поля длинной 2005 байта получится два чанка: первый размером 2032 байт и второй чанк размером 45 байт. Маленький чанк будет по возможности вставлен серверным процессом в блок с первым чанком размера 2032 байт. Чанки одного поля вставляются по возможности в один блок исходя из того, чтобы быстрее собирать или удалять чанки относящиеся к одному полю.

Из-за того, что поле делится на 2 чанка в блоке таблицы TOAST поместится только 3 чанка по 2032 байт и будет свободно 1896 байт (1944 - 24 байт заголовка блока - 6 слотов заголовка блока *4 байта=1896 байт). Свободное место может использоваться под чанки от других полей, если чанки поместятся на свободное место.

6) Небольшие чанки не дают поместиться в блоки чанкам большого размера. Посмотрим при каких условиях чанки небольших не появляются.
Пересоздайте таблицу с двумя столбцами размером 8 байт и текстовым столбцом:

```
drop table if exists t;
create table t (id int8, c1 int8, c text storage external);
insert into t VALUES (1, 1, repeat('a',1989));
insert into t VALUES (1, 1, repeat('a',1989));
insert into t VALUES (1, 1, repeat('a',1989));
insert into t VALUES (1, 1, repeat('a',1989));
select pg_column_size(t.*) from t;
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
```

```
DROP TABLE
CREATE TABLE
ALTER TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
 pg_column_size
----------------
           2033
           2033
           2033
           2033
(4 rows)

 lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+--------
  1 |   6152 |   2025 | (0,1)  |     24
  2 |   4120 |   2025 | (0,2)  |     24
  3 |   2088 |   2025 | (0,3)  |     24
  4 |     56 |   2025 | (0,4)  |     24
```

Каждое поле было вытеснено в один чанк, а не в два чанка.
В блок влезли 4 чанка и свободного места в блоке почти не осталось: 56-24-4*4=16 байт.

Однако, если вставить поле размером больше, чем 1989 байт:
`insert into t VALUES (1, 1, repeat('a',1990));`
, то такое поле будет поделено на два чанка.

7) Для лучшего понимания повторите создание таблицы. Пересоздайте таблицу с тремя четырёхбайтными столбцами:

```
drop table if exists t;
create table t (id int4, c1 int4, c2 int4, c text storage external);
insert into t VALUES (1, 1, 1, repeat('a',1993));
insert into t VALUES (1, 1, 1, repeat('a',1993));
insert into t VALUES (1, 1, 1, repeat('a',1993));
insert into t VALUES (1, 1, 1, repeat('a',1993));
insert into t VALUES (1, 1, 1, repeat('a',2640));
insert into t VALUES (1, 1, 1, repeat('a',2640));
insert into t VALUES (1, 1, 1, repeat('a',2640));
select pg_column_size(t.*) from t;
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
```

```
 select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',1));
 DROP TABLE
 CREATE TABLE
 ALTER TABLE
 INSERT 0 1
 INSERT 0 1
 INSERT 0 1
 INSERT 0 1
 INSERT 0 1
 INSERT 0 1
 INSERT 0 1
  pg_column_size
 ----------------
             2033
             2033
             2033
             2033
             2678
             2678
             2678
 (7 rows)

  lp | lp_off | lp_len | t_ctid | t_hoff
 ----+--------+--------+--------+--------
   1 |   6152 |   2029 | (0,1)  |     24
   2 |   4120 |   2029 | (0,2)  |     24
   3 |   2088 |   2029 | (0,3)  |     24
   4 |     56 |   2029 | (0,4)  |     24
 (4 rows)

  lp | lp_off | lp_len | t_ctid | t_hoff
 ----+--------+--------+--------+--------
   1 |   6152 |   2032 | (1,1)  |     24
   2 |   5472 |    678 | (1,2)  |     24
   3 |   3440 |   2032 | (1,3)  |     24
   4 |   2760 |    678 | (1,4)  |     24
   5 |    728 |   2032 | (1,5)  |     24
   6 |     48 |    678 | (1,6)  |     24
 (6 rows)
```

Поля первых 4 строк имеют один чанк и поместились в первый блок TOAST-таблицы.
Поля следующих трёх строк имеют по два чанка и заняли второй блок.

8) Оптимизировать здесь можно разве что: с точки зрения использования места и производительности без разницы хранить без сжатия поля размером от 1997 байт и до 2640 байт. Пример:

```
 truncate table t;
 insert into t VALUES (1, 1, 1, repeat('a',1997));
 insert into t VALUES (1, 1, 1, repeat('a',1997));
 insert into t VALUES (1, 1, 1, repeat('a',1997));
 select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
 TRUNCATE TABLE
 INSERT 0 1
 INSERT 0 1
 INSERT 0 1
  lp | lp_off | lp_len | t_ctid | t_hoff
 ----+--------+--------+--------+--------
   1 |   6152 |   2032 | (0,1)  |     24
   2 |   6112 |     37 | (0,2)  |     24
   3 |   4080 |   2032 | (0,3)  |     24
   4 |   4040 |     37 | (0,4)  |     24
```

```
  5 |   2008 |   2032 | (0,5)  |      24
  6 |   1968 |     37 | (0,6)  |      24
(6 rows)
```

Три поля размером начиная с 1997 байт без сжатия создают два чанка на каждое поле. Три поля займут один блок. В блоке останется 1968-24-6*4=1920 байт, но они вряд ли будут использованы потому, что чанков максимального размера больше, чем "обрезков".

9) Для предыдущей таблицы с полем 2640 байт:

```
truncate table t;
insert into t VALUES (1, 1, 1, repeat('a',2640));
insert into t VALUES (1, 1, 1, repeat('a',2640));
insert into t VALUES (1, 1, 1, repeat('a',2640));
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
TRUNCATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
 lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+--------
  1 |   6152 |   2032 | (0,1)  |      24
  2 |   5472 |    680 | (0,2)  |      24
  3 |   3440 |   2032 | (0,3)  |      24
  4 |   2760 |    680 | (0,4)  |      24
  5 |    728 |   2032 | (0,5)  |      24
  6 |     48 |    680 | (0,6)  |      24
(6 rows)
```

Три поля размером 2636 байт тоже создали по два чанка на поле и также заняли один блок TOAST-таблицы. Свободного места в блоке нет.

10) Для таблицы с двумя столбцами по 8 байт:

```
create table t (id int8, c1 int8, c text);
insert into t VALUES (1, 1, repeat('a',1989));
```

поля размером 1989 байт создают один чанк и хранение оптимально.

11) Для таблицы с двумя столбцами длинной 4 байт:

```
create table t (id int4, c1 int4, c text);
insert into t VALUES (1, 1, repeat('a',1997));
```

поля создадут по два чанка, а значит хранение будет менее оптимально.

В примерах для простоты приводятся таблицы с одним столбцом, вытесняемым без сжатия и небольшим числом невытесняемых столбцов. В таком случае размер первого чанка будет иметь максимальный размер. Если столбцов в таблице много, то первые чанки могут иметь небольшой размер. Если у чанков небольшой размер, то они смогут поместиться в свободное место блоков TOAST-таблицы. При анализе TOAST-таблиц стоит обратить внимание на свободное место в блоках TOAST. Если свободного места "много" (с точки зрения того кто занимается оптимизацией), то стоит обратить внимание на размеры чанков. Если много чанков минимального размера (до ~100 байт), то есть потенциал для оптимизации.

В первую очередь можно попробовать сменить алгоритм сжатия параметром `default_toast_compression`, возможно улучшение сжатия позволит уменьшить число небольших чанков.

Парадоксально, но отключение сжатия может увеличить производительность за счёт исключения траты ресурсов на сжатие-разжатие и при этом не изменить размер TOAST-таблицы. Такое произойдёт, если:

a) поля сжимаются менее, чем в 2 раза
b) размер поля меньше, чем 1996*2=3992 байта
c) после сжатия получается не 1 чанк, а два чанка и без сжатия получается тоже два чанка.
Можно попробовать увеличить (менее вероятно уменьшить) параметр `toast_tuple_target`. Параметр устанавливается только на уровне таблицы:

```
ALTER TABLE t SET (toast_tuple_target = 2032);
```

Изменение параметра влияет на число строк, помещающихся в таблице. При уменьшении значения ниже 2032 (значение по умолчанию) может начать выноситься в TOAST больше полей и работа с такими полями замедлится. Однако эти поля могут заполнить пустые места в блоках TOAST. При увеличении значения больше 2032, большее число полей начнёт оставаться в таблице и размер таблицы увеличится. Увеличит это или уменьшит производительность нельзя предсказать. Например, уменьшение числа строк, помещающихся в блок таблицы, в редких случаях может снизить конкуренцию за доступ к буферам буферного кэша.

Часть 2. Структура таблиц TOAST

1) В таблице TOAST есть два столбца: `chunk_id` (тип OID, уникальный для поля, вынесенного в TOAST), `chunk_seq` (порядковый номер чанка), `chunk_data` (данные поля). Для быстрого доступа к чанкам на TOAST-таблицу создается составной уникальный индекс по `chunk_id` и `chunk_seq`.

```
postgres=# drop table if exists t;
create table t (c text storage external);
insert into t VALUES (repeat('a',2005));
insert into t VALUES (repeat('a',2005));
insert into t VALUES (repeat('a',2005));
insert into t VALUES (repeat('a',2005));
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
DROP TABLE
CREATE TABLE
ALTER TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
 lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+--------
  1 |   6152 |   2032 | (0,1)  |     24
  2 |   6104 |     45 | (0,2)  |     24
  3 |   4072 |   2032 | (0,3)  |     24
  4 |   4024 |     45 | (0,4)  |     24
  5 |   1992 |   2032 | (0,5)  |     24
  6 |   1944 |     45 | (0,6)  |     24
(6 rows)
```

Полный размер строки с длинным чанком 2032 байта (6104-4072).

2) Запрос, который удобен тем, что выдаёт одну строку на блок и полезен для определения свободного места в блоке TOAST и обычных таблиц:

```
postgres=# select lower, upper, special, pagesize from
page_header(get_raw_page((select reltoastrelid::regclass::text from pg_class
where relname='t'),'main',0));
 lower | upper | special | pagesize
-------+-------+---------+----------
    48 |  1944 |    8184 |     8192
(1 row)
```

Пример как рассчитывать место в блоке для 4 строк размера 2032 байт (с 4 чанками):
24 (заголовок)+ 4*4 (заголовок) + 2032*4 + 8 (pagesize-special)=8176. Не используется 16 байт, но они бы и не могли использоваться, так строки выравниваются по 8 байт, а их 4.

3) Запрос для определения названия TOAST-таблицы чтобы использовать название в командах psql:

```
postgres=# select reltoastrelid, reltoastrelid::regclass from pg_class where
relname='t';
 reltoastrelid |      reltoastrelid
---------------+--------------------------
        546160 | pg_toast.pg_toast_546157
(1 row)
```

4) Команда psql для просмотра характеристик TOAST-таблицы

```
postgres=# \d+ pg_toast.pg_toast_546157
TOAST table "pg_toast.pg_toast_546157"
   Column   |  Type   | Storage
------------+---------+---------
 chunk_id   | oid     | plain
 chunk_seq  | integer | plain
 chunk_data | bytea   | plain
Owning table: "public.t"
Indexes:
    "pg_toast_546157_index" PRIMARY KEY, btree (chunk_id, chunk_seq)
Access method: heap
```

5) Запрос к TOAST-таблице для просмотра размеров чанков:

```
postgres=# select chunk_id, chunk_seq, length(chunk_data) from
pg_toast.pg_toast_546157;
 chunk_id | chunk_seq | length
----------+-----------+--------
   546162 |         0 |   1996
   546162 |         1 |      9
   546163 |         0 |   1996
   546163 |         1 |      9
   546164 |         0 |   1996
   546164 |         1 |      9
   546165 |         0 |   1996
   546165 |         1 |      9
(8 rows)
```

length(chunk_data)=1996 соответствует значению "Maximum size of a TOAST chunk: 1996" в управляющем файле.

6) Значение, остающееся в блоке таблицы, после выноса поля в TOAST:

```
postgres=# select lp_len, t_data from heap_page_items(get_raw_page('t', 0));
 lp_len |                 t_data
--------+----------------------------------------
     42 | \x0112d9070000d5070000725508007055080
     42 | \x0112d9070000d5070000735508007055080
     42 | \x0112d9070000d5070000745508007055080
     42 | \x01 12 d9070000 d5070000 75550800 70550800
(4 rows)
```

\x07d5 = 2005 что равно 1996+9
\x07d9 = 2009

В этих значениях хранится указатель на первый chunk **размером 18 байт (независимо от размера поля)**.
В этих 18 байтах хранится структура varatt_external, описанная в varatt.h:
первый байт имеет значение 0x01, это признак того, что поле вынесено в TOAST;
второй байт - длинна этой записи (значение 0x12 = 18 байт);
4 байта длинна поля с заголовком поля до сжатия;
4 байта длинна того, что вынесено в TOAST;
4 байта - указатель на первый чанк в TOAST (столбец chunk_id таблицы TOAST);
4 байта - oid toast-таблицы (pg_class.reltoastrelid)

В столбце chunk_id (тип oid 4 байта) может быть 4млрд. (2 в степени 32) значений. Это значит, что в одной таблице в TOAST может быть вытеснено только 4млрд. **полей** (даже не строк). Это существенно ограничивает количество строк в исходной таблице.

7) Проверим, действительно ли у каждого поля отдельный chunk_id?

```
drop table if exists t3;
create table t3 (a text storage external, b text storage external);
truncate t;
insert into t3 VALUES (repeat('a',2005), repeat('b',2005));
select lp, t_ctid, lp_off,lp_len,t_hoff,t_data from
heap_page_items(get_raw_page('t3', 0));
SELECT reltoastrelid, reltoastrelid::text FROM pg_class WHERE relname='t3';
SELECT reltoastrelid, reltoastrelid::regclass FROM pg_class WHERE relname='t3';
DROP TABLE
CREATE TABLE
ALTER TABLE
ALTER TABLE
TRUNCATE TABLE
INSERT 0 1
 lp | t_ctid | lp_off | lp_len | t_hoff |                    t_data
----+--------+--------+--------+--------+---------------------------------
  1 | (0,1)  |   8112 |     60 |     24 |
\x0112d9070000d5070000885508008355080000112d9070000d5070000895508008355080000
(1 row)

 reltoastrelid | reltoastrelid
---------------+---------------
        546169 | 546169
(1 row)

 reltoastrelid |      reltoastrelid
---------------+--------------------------
        546169 | pg_toast.pg_toast_546166
(1 row)
ostgres=# select chunk_id, chunk_seq, length(chunk_data) from
pg_toast.pg_toast_546166;
 chunk_id | chunk_seq | length
----------+-----------+--------
   546174 |         0 |   1996
   546174 |         1 |      9
   546175 |         0 |   1996
   546175 |         1 |      9
(4 rows)
```

У каждого поля отдельный chunk_id.

Выравнивание по длинне всей строки, если остаётся 18 байт также присутствует.

8) Сколько строк поместится в блоке?

```
drop table if exists t3;
drop table if exists t4;
create table t3 (c1 text storage external);
create table t4 (c1 serial , c2 smallint, c3 text storage external);
truncate t3;
truncate t4;
DO $$ BEGIN
FOR i IN 1 .. 200 LOOP
insert into t3 VALUES (repeat('a',1000000));
insert into t4 VALUES (default, 1, repeat('a',1000000));
END LOOP; END; $$ LANGUAGE plpgsql;
select count(*) from heap_page_items(get_raw_page('t3','main',0)) where
(t_ctid::text::point)[0]=0
union all
select count(*) from heap_page_items(get_raw_page('t4','main',0)) where
(t_ctid::text::point)[0]=0;
select pg_total_relation_size('t3'), pg_total_relation_size('t4');
DROP TABLE
DROP TABLE
CREATE TABLE
CREATE TABLE
ALTER TABLE
ALTER TABLE
TRUNCATE TABLE
TRUNCATE TABLE
DO
 count
-------
   156
   156
(2 rows)
```

Если заменить serial и smallint на bigserial, то размер будет разный (156 и 135 строк в каждом блоке). bigserial бесполезен для таблиц, в которых много полей вытесняется в TOAST, так как в таблицах сможет сохраниться не больше 4млрд (2^32) строк с одним длинным полем или 2млрд. с двумя длинными полями.

```
 pg_total_relation_size | pg_total_relation_size
------------------------+------------------------
              208011264 |              208011264
(1 row)
```

9) Блоки таблиц t3 и t4 содержат одинаковые данные по свободному месту в блоках. Пример для первых блоков таблиц:

```
SELECT * FROM page_header(get_raw_page((SELECT reltoastrelid::regclass::text
FROM pg_class WHERE relname='t3'),'main',0));
    lsn     | checksum | flags | lower | upper | special | pagesize | version | prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+-----------
 8/3205CB70 |        0 |     4 |    40 |    56 |    8184 |     8192 |       5 |         0
(1 row)

SELECT * FROM page_header(get_raw_page((SELECT reltoastrelid::regclass::text
FROM pg_class WHERE relname='t4'),'main',0));
    lsn     | checksum | flags | lower | upper | special | pagesize | version | prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+-----------
 8/25366AA0 |    25187 |     0 |    40 |   168 |    8184 |     8192 |       5 |         0
(1 row)
```

10) В каждом поле миллион байт `repeat('a',1000000)`.б поэтому большинство чанков имеют максимальные размеры и хранение чанков компактное:

```
SELECT lp,lp_off,lp_len,t_ctid,t_hoff FROM heap_page_items(get_raw_page((SELECT
reltoastrelid::regclass::text FROM pg_class WHERE relname='t3'),'main',0));
```

```
 lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+--------
  1 |   6152 |   2032 | (0,1)  |     24
  2 |   4120 |   2032 | (0,2)  |     24
  3 |   2088 |   2032 | (0,3)  |     24
  4 |     56 |   2032 | (0,4)  |     24
(4 rows)
```

Свободного места в блоке между заголовком блока и данными нет: 56 - 24 - 4*4 = 16 байт.
Заполнение блока полное и соответствует константам TOAST_TUPLE_TARGET=TOAST_TUPLE_THRESHOLD=2032.
Заголовок строки 24 байта. Размер области данных строки 2032-24=2008 байт.

11) В строке любой TOAST-таблицы три столбца:

```
postgres=# \d pg_toast.pg_toast_546186
TOAST table "pg_toast.pg_toast_546186"
   Column   |  Type
------------+---------
 chunk_id   | oid
 chunk_seq  | integer
 chunk_data | bytea
Owning table: "public.t3"
Indexes:
    "pg_toast_546186_index" PRIMARY KEY, btree (chunk_id, chunk_seq)
```

12) Содержимое первой строки в TOAST-таблице:

```
postgres=# select * from
heap_page_items(get_raw_page('pg_toast.pg_toast_546186', 0)) limit 1 \gx
-[ RECORD 1 ]---
lp          | 1
lp_off      | 6152
lp_flags    | 1
lp_len      | 2032
t_xmin      | 2
t_xmax      | 0
t_field3    | 0
t_ctid      | (0,1)
t_infomask2 | 3
t_infomask  | 2818
t_hoff      | 24
t_bits      |
t_oid       |
t_data      | \x9c550800 00000000 401f0000616161616161616161
```

Размер поля bytea хранится в первых 4 байтах поля bytea: *401f0000*

13) Посмотрите как выглядит строка, оставшаяся в таблице:

```
postgres=# select lp_len, t_data from heap_page_items(get_raw_page('t3', 0))
limit 2;
 lp_len |                t_data
--------+---------------------------------------
     42 | \x01 12 44420f00 40420f00 9c550800 8d550800
     42 | \x01 12 44420f00 40420f00 9e550800 8d550800
(2 rows)

\x44420f00 = 1000004
```

\x40420f00 = 1000000 что соответствует миллиону символов, вставленных в поле функцией **repeat('a',1000000)**

14) Создайте таблицу со структурой TOAST:

```
postgres=# drop table if exists t5;
create table t5 (a oid, b integer, c bytea);
insert into t5 values (1, 1, '');
insert into t5 values (2, 2, 'a');
insert into t5 values (3, 3, 'aa');
select lp,lp_off,lp_len,t_ctid,t_hoff,t_data from
heap_page_items(get_raw_page('t5', 0));
DROP TABLE
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
 lp | lp_off | lp_len | t_ctid | t_hoff |          t_data
----+--------+--------+--------+--------+--------------------------
  1 |   8136 |     33 | (0,1)  |     24 | \x01000000 01000000 03
  2 |   8096 |     34 | (0,2)  |     24 | \x02000000 02000000 0561
  3 |   8056 |     35 | (0,3)  |     24 | \x03000000 03000000 076161
(3 rows)
```

Длинна каждой из трёх строк: 8136-8096=8096-8056=40 байт.
В 40 байт входит заголовок 24 байта. Область данных: 40-24=16=4 (первый столбец типа oid)+4 (второй столбец типа integer)+ **8 байт**. Почему **8 байт**? Ведь в результате запроса в первой строке 1 байт, во второй 2 байта, в третьей 3 байта. Потому что вся строка всегда выравнивается до 8 байт. В столбце lp_len длинна трёх строк выдана без выравнивания: 33, 34, 35 байт.

Часть 3. Эффективность UPDATE в сравнении с INSERT

Если в одной транзакции много раз обновлять одну и ту же строку, то место, занимаемое порождаемыми версиями строк не сможет очиститься. Такое может произойти при использовании триггеров AFTER.. FOR EACH ROW. Стоит проверять код приложения на присутствие фраз FOR EACH ROW, DEFERRABLE INITIALLY DEFERRED. Второе на что можно обращать внимание - если в логике работы с таблицами преобладают UPDATE вместо INSERT.

1) Выполните команды:

```
postgres=# drop table if exists item;
drop table if exists sum;
drop function if exists add_to_sum();
create table item (id bigserial, value int8);
create table sum (total int8);
insert into sum values (0);
CREATE FUNCTION add_to_sum() RETURNS trigger LANGUAGE plpgsql AS
$$BEGIN
 UPDATE sum SET total = total + NEW.value;
 RETURN NEW;
END;$$;
CREATE CONSTRAINT TRIGGER add_to_sum
AFTER INSERT ON item
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW
EXECUTE FUNCTION add_to_sum();
analyze item, sum;
DROP TABLE
DROP TABLE
DROP FUNCTION
CREATE TABLE
CREATE TABLE
```

```
INSERT 0 1
CREATE FUNCTION
CREATE TRIGGER
ANALYZE
```

Создаются две таблицы item и sum. После вставки строки в таблицу item триггер выполняет обновление строки в таблице sum.

2) Включите измерение времени выполнения команд:

```
postgres=# \timing on
Timing is on.
```

3) Выполните вставку 10, 20, 30 тыс. строк:

```
postgres=# insert into item select * from generate_series(1,10000);
 insert into item select * from generate_series(1,20000);
 insert into item select * from generate_series(1,30000);
INSERT 0 10000
Time: 2998.265 ms (00:02.998)
INSERT 0 20000
Time: 11798.652 ms (00:11.799)
INSERT 0 30000
Time: 26589.966 ms (00:26.590)
```

Каждая команда выполняется в одной транзакции. Обновления строки триггером выполняются в той же транзакции.

При увеличении числа строк, вставляемых в одной транзакции время выполнения команды увеличивается нелинейно.

Следовать идеям "чем реже COMMIT, тем больше производительность", "пакетные обновления быстрее одиночных" не стоит. В PostgreSQL увеличение длительности транзакции приводит к ухудшению производительности из-за невозможности выполнить очистку (HOT и автовакуумом) блоков от старых версий строк.

4) Пересоздайте триггер и триггерную функцию:

```
postgres=# drop trigger add_to_sum on item;
drop function add_to_sum();
CREATE OR REPLACE FUNCTION add_to_sum() RETURNS trigger
 LANGUAGE plpgsql AS
$$BEGIN
 UPDATE sum SET total = total + (SELECT sum(value) FROM new_rows);
 RETURN NULL;
END;$$;
CREATE TRIGGER add_to_sum
 AFTER INSERT ON item
 REFERENCING NEW TABLE AS new_rows
 FOR EACH STATEMENT
 EXECUTE FUNCTION add_to_sum();
DROP TRIGGER
Time: 7.679 ms
DROP FUNCTION
Time: 3.456 ms
CREATE FUNCTION
Time: 4.232 ms
CREATE TRIGGER
Time: 5.288 ms
```

5) Повторите команды вставки 10, 20, 30 тыс. строк:

```
postgres=# insert into item select * from generate_series(1,10000);
 insert into item select * from generate_series(1,20000);
 insert into item select * from generate_series(1,30000);
```

```
INSERT 0 10000
Time: 22.831 ms
INSERT 0 20000
Time: 44.134 ms
INSERT 0 30000
Time: 62.785 ms
```

Длительность выполнения команд уменьшилось в сотни раз с 26589.966 ms до 62.785 ms и стало линейным.

6) Была выполнена замена **FOR EACH ROW** на **FOR EACH STATEMENT** это добавило эффективности, так как триггерная функция вызывается намного реже.

Более интересно сравнить эффективность замены логики UPDATE на INSERT. При проектировании логики приложения может казаться, что обновление строк не увеличивает число строк и более предпочтительно, чем вставка строк (приложение, накапливающее данные) и последующее периодическое удаление ненужных строк.

Пересоздайте триггер и триггерную функцию заменив обновление строки вставкой новой строки:

```
postgres=# drop trigger add_to_sum on item;
drop function if exists add_to_sum();
CREATE FUNCTION add_to_sum() RETURNS trigger LANGUAGE plpgsql AS
$$BEGIN
 insert into sum values (NEW.value);
 RETURN NEW;
END;$$;
CREATE CONSTRAINT TRIGGER add_to_sum
AFTER INSERT ON item
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW
EXECUTE FUNCTION add_to_sum();
DROP TRIGGER
Time: 5.033 ms
DROP FUNCTION
Time: 3.456 ms
CREATE FUNCTION
Time: 5.270 ms
CREATE TRIGGER
Time: 4.590 ms
```

7) Повторите команды вставки 10, 20, 30 тыс. строк:

```
postgres=# insert into item select * from generate_series(1,10000);
 insert into item select * from generate_series(1,20000);
 insert into item select * from generate_series(1,30000);
INSERT 0 10000
Time: 87.187 ms
INSERT 0 20000
Time: 188.129 ms
INSERT 0 30000
Time: 247.266 ms
```

Длительность вставки 30 тыс.строк уменьшилось **в сотню раз** с **26589.966 ms** до **247.266 ms**.

При этом триггерная функция срабатывала по кажой строке.

Эффект от замены логики приложения UPDATE на INSERT может быть существенный.

Замедление срабатывания триггера **FOR EACH ROW** по сравнению со срабатыванием один раз **FOR EACH STATEMENT** ~4 раза: **247.266 ms** в сравнении с **62.785 ms**.

Также мы выяснили эффект оптимизации: замена FOR EACH ROW на FOR EACH STATEMENT ~4 раза.

8) Выключите измерение времени выполнения команд:

```
postgres=# \timing off
Timing is off.
```

9) Удалите созданные объекты:

```
postgres=# drop trigger add_to_sum on item;
drop table if exists item;
drop table if exists sum;
drop function if exists add_to_sum();
DROP TRIGGER
DROP TABLE
DROP TABLE
DROP FUNCTION
```

Часть 4. HOT cleanup

При выполнении запросов SELECT и других команд серверный процесс может удалить dead tuples (версии строк, вышедшие за горизонт видимости базы данных, xmin horizon) и выполнить реорганизацию версий строк внутри блока. Это называется внутристраничной очисткой.

HOT cleanup ("pruning") выполняется, если выполняется одно из условий:
a) блок заполнен более чем на 90% или более чем на FILLFACTOR (по умолчанию 100%);
b) ранее выполнявшийся UPDATE не нашел места (то есть установил в заголовке блока подсказку PD_PAGE_FULL).

Внутристраничная очистка работает в пределах одной табличной страницы, не очищает индексные страницы, не обновляет карту свободного пространства и карту видимости.
Указатели (4 байта) в заголовке блока не освобождаются, они обновляются, чтобы указывать на актуальную версию строки. Указатели освободить нельзя, так как на них могут существовать ссылки из индексов, это проверить серверный процесс не может. Только вакуум сможет освободить указатели (сделать указатели unused).

В этой части практики вы посмотрите в каком случае не выполняется условие "b" и как это влияет на выполнение HOT cleanup.

1) Создайте функцию для удобного просмотра битов-подсказок:

```
create or replace function heap_page(relname text, pageno integer) returns
 table(lp_off text, ctid tid,state text,xmin text,xmax text,hhu text,hot
text,t_ctid tid, multi text)
as $$
select lp_off, (pageno,lp)::text::tid 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,
 case when (t_infomask&4096) >0 then 't' else 'f'
 end as multi
 from heap_page_items (get_raw_page(relname, pageno))
 order by lp;
$$ language sql;
CREATE FUNCTION
```

2) Создайте таблицу, вставьте строку и обновите три раза эту строку:

```
drop table if exists t;
create table t(s text storage plain) with (autovacuum_enabled=off);
insert into t values (repeat('a',2004));
update t set s=(repeat('c',2004));
update t set s=(repeat('c',2004));
update t set s=(repeat('c',2004));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
DROP TABLE
CREATE TABLE
INSERT 0 1
UPDATE 1
UPDATE 1
UPDATE 1
 ctid  | lp_off | ctid  | state  |  xmin   |  xmax   | hhu | hot | t_ctid | multi
-------+--------+-------+--------+---------+---------+-----+-----+--------+-------
 (0,1) | 6144   | (0,1) | normal | 424011c | 424012c | t   |     | (0,2)  | f
 (0,2) | 4112   | (0,2) | normal | 424012c | 424013c | t   | t   | (0,3)  | f
 (0,3) | 2080   | (0,3) | normal | 424013c | 424014  | t   | t   | (0,4)  | f
 (0,4) | 48     | (0,4) | normal | 424014  | 0a      |     | t   | (0,4)  | f
(4 rows)

 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     0 |    40 |    48 |    8176 |    424012
(1 row)
```

В блоке организована цепочка из четырёх версий строки. Значение в столбце xmax=0a означает, что эта версия строки не удалена. Если в xmin нет "c" это означает, что статус транзакции, породившей версию строки, не указан. Если в столбце xmin есть "c", то это означает, что транзакция, создавшая версию строки была зафиксирована.

lp_off=48 или upper=48 означает, что блок заполнен более, чем на 90%.

Флаг prune_xid был установлен первым UPDATE. Флаг prune_xid является подсказкой (но не приказом), что имеется вероятность, что в блоке можно очистить место.

Первый UPDATE организовал цепочку HOT, выставив бит-подсказку:

hhu=t - подсказка процессам, что надо идти по цепочке ctid.

hot=t - подсказка процессам, что на данную версию строки нет ссылок из индексов.

3) Если дальше продолжать обновлять строку, то обновления будут идти в пределах одного блока:

```
update t set s=(repeat('c',2004));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
update t set s=(repeat('c',2004));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
update t set s=(repeat('c',2004));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
update t set s=(repeat('c',2004));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
update t set s=(repeat('c',2004));
```

```
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
update t set s=(repeat('c',2004));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
update t set s=(repeat('c',2004));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
select ctid,* from heap_page('t',1);
```

```
UPDATE 1
 ctid  | lp_off | ctid  |     state      |  xmin   |  xmax   | hhu | hot | t_ctid | multi
-------+--------+-------+----------------+---------+---------+-----+-----+--------+-------
 (0,1) | 4      | (0,1) | redirect to 4 |         |         |     |     |        | f
 (0,2) | 4112   | (0,2) | normal         | 424015  | 0a      |     | t   | (0,2)  | f
 (0,3) | 0      | (0,3) | unused         |         |         |     |     |        | f
 (0,4) | 6144   | (0,4) | normal         | 424014c | 424015 | t   | t   | (0,2)  | f
(4 rows)

 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     1 |    40 |  4112 |    8176 |    424012
(1 row)

UPDATE 1
 ctid  | lp_off | ctid  |     state      |  xmin   |  xmax   | hhu | hot | t_ctid | multi
-------+--------+-------+----------------+---------+---------+-----+-----+--------+-------
 (0,1) | 4      | (0,1) | redirect to 4 |         |         |     |     |        | f
 (0,2) | 4112   | (0,2) | normal         | 424015c | 424016  | t   | t   | (0,3)  | f
 (0,3) | 2080   | (0,3) | normal         | 424016  | 0a      |     | t   | (0,3)  | f
 (0,4) | 6144   | (0,4) | normal         | 424014c | 424015c | t   | t   | (0,2)  | f
(4 rows)

 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     1 |    40 |  2080 |    8176 |    424012
(1 row)

UPDATE 1
 ctid  | lp_off | ctid  |     state      |  xmin   |  xmax   | hhu | hot | t_ctid | multi
-------+--------+-------+----------------+---------+---------+-----+-----+--------+-------
 (0,1) | 4      | (0,1) | redirect to 4 |         |         |     |     |        | f
 (0,2) | 4112   | (0,2) | normal         | 424015c | 424016c | t   | t   | (0,3)  | f
 (0,3) | 2080   | (0,3) | normal         | 424016c | 424017  | t   | t   | (0,5)  | f
 (0,4) | 6144   | (0,4) | normal         | 424014c | 424015c | t   | t   | (0,2)  | f
 (0,5) | 48     | (0,5) | normal         | 424017  | 0a      |     | t   | (0,5)  | f
(5 rows)

 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     0 |    44 |    48 |    8176 |    424012
(1 row)

UPDATE 1
 ctid  | lp_off | ctid  |     state      |  xmin   |  xmax   | hhu | hot | t_ctid | multi
-------+--------+-------+----------------+---------+---------+-----+-----+--------+-------
 (0,1) | 5      | (0,1) | redirect to 5 |         |         |     |     |        | f
 (0,2) | 4112   | (0,2) | normal         | 424018  | 0a      |     | t   | (0,2)  | f
 (0,3) | 0      | (0,3) | unused         |         |         |     |     |        | f
 (0,4) | 0      | (0,4) | unused         |         |         |     |     |        | f
 (0,5) | 6144   | (0,5) | normal         | 424017c | 424018 | t   | t   | (0,2)  | f
(5 rows)

 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     1 |    44 |  4112 |    8176 |    424012
(1 row)

UPDATE 1
 ctid  | lp_off | ctid  |     state      |  xmin   |  xmax   | hhu | hot | t_ctid | multi
-------+--------+-------+----------------+---------+---------+-----+-----+--------+-------
 (0,1) | 5      | (0,1) | redirect to 5 |         |         |     |     |        | f
 (0,2) | 4112   | (0,2) | normal         | 424018c | 424019  | t   | t   | (0,3)  | f
 (0,3) | 2080   | (0,3) | normal         | 424019  | 0a      |     | t   | (0,3)  | f
```

```
 (0,4)  | 0       | (0,4)  | unused           |          |          |     |     |        | f
 (0,5)  | 6144    | (0,5)  | normal           | 424017c  | 424018c  | t   | t   | (0,2)  | f
(5 rows)

 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     1 |    44 |  2080 |    8176 |    424012
(1 row)

UPDATE 1
 ctid   | lp_off | ctid   |      state       |   xmin   |   xmax   | hhu | hot | t_ctid | multi
--------+--------+--------+------------------+----------+----------+-----+-----+--------+-------
 (0,1)  | 5       | (0,1)  | redirect to 5 |          |          |     |     |        | f
 (0,2)  | 4112    | (0,2)  | normal           | 424018c  | 424019c  | t   | t   | (0,3)  | f
 (0,3)  | 2080    | (0,3)  | normal           | 424019c  | 424020   | t   | t   | (0,4)  | f
 (0,4)  | 48      | (0,4)  | normal           | 424020   | 0a       |     | t   | (0,4)  | f
 (0,5)  | 6144    | (0,5)  | normal           | 424017c  | 424018c  | t   | t   | (0,2)  | f
(5 rows)

 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     1 |    44 |    48 |    8176 |    424012
(1 row)

UPDATE 1
 ctid   | lp_off | ctid   |      state       |   xmin   |   xmax  | hhu | hot | t_ctid | multi
--------+--------+--------+------------------+----------+---------+-----+-----+--------+-------
 (0,1)  | 4       | (0,1)  | redirect to 4 |          |         |     |     |        | f
 (0,2)  | 4112    | (0,2)  | normal           | 424021   | 0a      |     | t   | (0,2)  | f
 (0,3)  | 0       | (0,3)  | unused           |          |         |     |     |        | f
 (0,4)  | 6144    | (0,4)  | normal           | 424020c  | 424021  | t   | t   | (0,2)  | f
(4 rows)

 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     1 |    40 |  4112 |    8176 |    424012
(1 row)

ERROR:  block number 1 is out of range for relation "t"
CONTEXT:  SQL function "heap_page" statement 1
```

в блоке 1 нет строк, все обновления в пределах блока 0.

4) Посмотрим что произойдёт, если увеличить длинну текстового поля с 2004 до 2005 байт:

```
truncate table t;
insert into t values (repeat('a',2005));
update t set s=(repeat('c',2005));
update t set s=(repeat('c',2005));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
update t set s=(repeat('c',2005));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
select ctid,* from heap_page('t',1);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 1));
update t set s=(repeat('c',2005));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
select ctid,* from heap_page('t',1);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 1));
update t set s=(repeat('c',2005));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
select ctid,* from heap_page('t',1);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 1));
update t set s=(repeat('c',2005));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
select ctid,* from heap_page('t',1);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 1));
```

```
select ctid,* from heap_page('t',2);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 2));
update t set s=(repeat('c',2005));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
select ctid,* from heap_page('t',1);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 1));
select ctid,* from heap_page('t',2);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 2));
TRUNCATE TABLE
INSERT 0 1
UPDATE 1
UPDATE 1
 ctid  | lp_off | ctid  | state  |  xmin   |  xmax   | hhu | hot | t_ctid | multi
-------+--------+-------+--------+---------+---------+-----+-----+--------+-------
 (0,1) | 6136   | (0,1) | normal | 424023c | 424024c | t   |     | (0,2)  | f
 (0,2) | 4096   | (0,2) | normal | 424024c | 424025c | t   | t   | (0,3)  | f
 (0,3) | 2056   | (0,3) | normal | 424025  | 0a      |     | t   | (0,3)  | f
(3 rows)

 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     0 |    36 |  2056 |    8176 |    424024
(1 row)
```

flags=0, а не 2 (PD_PAGE_FULL). Условие очистки "ранее выполнявшийся UPDATE не нашел места (то есть установил в заголовке блока подсказку PD_PAGE_FULL)" не выполнено.
upper=2056 означает, что в блоке больше 10% свободного места (свободного места 2056-36=2020 байт). Условие очистки "блок заполнен более чем на 90% или более чем на FILLFACTOR (по умолчанию 100%)" не выполнено.
Так как ни одно из условий очистки не сработало, то процесс не выполняет очистку.

```
UPDATE 1
 ctid  | lp_off | ctid  | state  |  xmin   |  xmax   | hhu | hot | t_ctid | multi
-------+--------+-------+--------+---------+---------+-----+-----+--------+-------
 (0,1) | 6136   | (0,1) | normal | 424023c | 424024c | t   |     | (0,2)  | f
 (0,2) | 4096   | (0,2) | normal | 424024c | 424025c | t   | t   | (0,3)  | f
 (0,3) | 2056   | (0,3) | normal | 424025c | 424026  |     | t   | (1,1)  | f
(3 rows)
```

Свободного места в блоке не хватает для вставки новой версии строки, так как свободного места 2056-36=2020 байт. В эти 2020 байт должны поместиться: данные 2005 байт + 4 байта под новый слот в заголовке блока + 3 байта на хранение длинны поля + заголовок строки 24 байта + выравнивание строки до 8 байт. Процесс устанавливает у третьей `(0,3)` строки:
a) `xmax=424026`;
b) признак фиксации транзакции породившей версию **`xmin=`**`424025c`;
c) указатель на новую версию строки в другом блоке **`t_ctid=(1,1)`**;
d) подсказку PD_PAGE_FULL в заголовке блока, что не нашел в блоке места: **`flags=2`**.

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     2 |    36 |  2056 |    8176 |    424024
(1 row)
```

`flags=2` означает, что в блоке нет места. **`flags=2`** устанавливается, если процесс искал место в блоке, не смог разместить версию строки и установил флаг, сообщая другим процессам, что в блоке нет места.

```
 ctid  | lp_off | ctid  | state  |  xmin   | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+--------+---------+------+-----+-----+--------+-------
 (1,1) | 6136   | (1,1) | normal | 424026 | 0a   |     |     | (1,1)  | f
(1 row)
```

строка была вставлена во второй блок.

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
```

```
     0 |    28 |  6136 |    8176 |          0
(1 row)
```

заголовок второго блока не имеет флагов и подсказок.

```
UPDATE 1
 ctid  | lp_off | ctid  | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+-------+------+------+-----+-----+--------+-------
 (0,1) | 0      | (0,1) | dead  |      |      |     |     |        | f
(1 row)
```

Процесс, обновляющий строку просмотрел все блоки (методом доступа Seq Scan) в поисках строки. Увидев в первом блоке флаг **`flags=2`** процесс выполнил быструю очистку первого блока. В результате очистки место в первом блоке освободилось. Место слотов указателей в заголовке блока могут освобождаться с конца, что и было сделано. Остался один слот со статусом `state=dead`.

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     0 |    28 |  8176 |    8176 |    424024
(1 row)
```

Флаг после очистки был установлен в значение 0. `prune_xid` не менялся.

```
 ctid  | lp_off | ctid  | state  |  xmin   |  xmax   | hhu | hot | t_ctid | multi
-------+--------+-------+--------+---------+---------+-----+-----+--------+-------
 (1,1) | 6136   | (1,1) | normal | 424026c | 424027  | t   |     | (1,2)  | f
 (1,2) | 4096   | (1,2) | normal | 424027  | 0a      |     | t   | (1,2)  | f
(2 rows)
```

Новая версия строки добавилась во второй блок, создав цепочку HOT в этом блоке.

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     0 |    32 |  4096 |    8176 |    424027
(1 row)

UPDATE 1
 ctid  | lp_off | ctid  | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+-------+------+------+-----+-----+--------+-------
 (0,1) | 0      | (0,1) | dead  |      |      |     |     |        | f
(1 row)

 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     0 |    28 |  8176 |    8176 |    424024
(1 row)

 ctid  | lp_off | ctid  | state  |  xmin   |  xmax   | hhu | hot | t_ctid | multi
-------+--------+-------+--------+---------+---------+-----+-----+--------+-------
 (1,1) | 6136   | (1,1) | normal | 424026c | 424027c | t   |     | (1,2)  | f
 (1,2) | 4096   | (1,2) | normal | 424027c | 424028  | t   | t   | (1,3)  | f
 (1,3) | 2056   | (1,3) | normal | 424028  | 0a      |     | t   | (1,3)  | f
(3 rows)
```

цепочка HOT растёт, пока есть место в блоке.

```
 flags | lower | upper | special | prune_xid к
-------+-------+-------+---------+-----------
     0 |    36 |  2056 |    8176 |    424027
(1 row)

UPDATE 1
 ctid  | lp_off | ctid  | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+-------+------+------+-----+-----+--------+-------
 (0,1) | 0      | (0,1) | dead  |      |      |     |     |        | f
(1 row)
```

Первый блок не помеменялся. fsm файл, если он есть, HOT не обновляется. fsm обновляется вакуумом.

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     0 |    28 |  8176 |    8176 |    424024
(1 row)

 ctid  | lp_off | ctid  | state  |  xmin   |  xmax   | hhu | hot | t_ctid | multi
-------+--------+-------+--------+---------+---------+-----+-----+--------+-------
 (1,1) | 6136   | (1,1) | normal | 424026c | 424027c | t   |     | (1,2)  | f
 (1,2) | 4096   | (1,2) | normal | 424027c | 424028c | t   | t   | (1,3)  | f
 (1,3) | 2056   | (1,3) | normal | 424028c | 424029  |     | t   | (2,1)  | f
(3 rows)

 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     2 |    36 |  2056 |    8176 |    424027
(1 row)
```

процесс искал место в блоке, не смог разместить версию строки и установил флаг PD_PAGE_FULL в заголовке блока (**flags=2**)

```
 ctid  | lp_off | ctid  | state  |  xmin   | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+--------+---------+------+-----+-----+--------+-------
 (2,1) | 6136   | (2,1) | normal | 424029 | 0a   |     |     | (2,1)  | f
(1 row)
```

Поскольку fsm (карта свободного места) либо нет, либо fsm отсутствует, то серверный процесс не использует первый блок, а добавляет блок к файлу и вставляет строку в новый- третий блок.

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     0 |    28 |  6136 |    8176 |         0
(1 row)

UPDATE 1
 ctid  | lp_off | ctid  | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+-------+------+------+-----+-----+--------+-------
 (0,1) | 0      | (0,1) | dead  |      |      |     |     |        | f
(1 row)
```

Первый блок не помеменялся

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     0 |    28 |  8176 |    8176 |    424024
(1 row)

 ctid  | lp_off | ctid  | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+-------+------+------+-----+-----+--------+-------
 (1,1) | 0      | (1,1) | dead  |      |      |     |     |        | f
(1 row)
```

процесс выполнил быструю очистку второго блока, так как в нем увидел флаг **flags=2**

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     0 |    28 |  8176 |    8176 |    424027
(1 row)

 ctid  | lp_off | ctid  | state  |  xmin   |  xmax   | hhu | hot | t_ctid | multi
-------+--------+-------+--------+---------+---------+-----+-----+--------+-------
 (2,1) | 6136   | (2,1) | normal | 424029c | 424030  | t   |     | (2,2)  | f
 (2,2) | 4096   | (2,2) | normal | 424030  | 0a      |     | t   | (2,2)  | f
(2 rows)
```

Строка была вставлена в третий блок, начав цепочку HOT.

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     0 |    32 |  4096 |    8176 |    424030
(1 row)
```

Что интересно: для таблицы с одним столбцом типа `text`, поле начиная с размера 2005 байт будет вытеснено в TOAST. Поле размером 2004 байт и меньше вытеснено не будет и для полей такого размера режим `storage plain`, указанный при создании таблицы, не имеет значения.

Практический смысл этой части практики в том, что если процент заполнения блока меньше 90% или FILLFACTOR, а новая версия строки или вставляемая строка не помещается в свободное место, то строки будут вставляться в новые блоки. Такое не будет происходить, если длинна строки 816 байт (10% от 8192 байт с учетом выравнивания) и меньше. Точную статистику случаев, когда при UPDATE не было найдено место в блоке и новая версия строки была вставлена в другой блок показывает **pg_stat_all_tables.n_tup_newpage_upd**.

В примере использовалось обновление одной строки, но можно использовать несколько строк, команды удаления и вставки строк. HOT cleanup очищает не только цепочки HOT, а любые удалённые версии строк, вышедшие за горизонт базы данных.

Часть 5. Мониторинг HOT cleanup

Для мониторинга HOT используются столбцы представлений **pg_stat_all_tables** и **pg_stat_user_tables**. Пример:

```
postgres=# select relname, n_tup_upd, n_tup_hot_upd, n_tup_newpage_upd,
round(n_tup_hot_upd*100/n_tup_upd,2) as hot_ratio from pg_stat_all_tables where
n_tup_upd<>0 order by 5;
    relname    | n_tup_upd | n_tup_hot_upd | n_tup_newpage_upd | hot_ratio
---------------+-----------+---------------+-------------------+-----------
 pg_rewrite    |        14 |             9 |                 5 |     64.00
 pg_proc       |        33 |            23 |                10 |     69.00
 pg_class      |     71645 |         63148 |              8351 |     88.00
 pg_attribute  |       270 |           267 |                 3 |     98.00
 pg_statistic  |      3974 |          3904 |                70 |     98.00
```

Сравним примеры из предыдущей части практики когда когда новая версия строки или вставляемая строка не помещается в свободное место блока и когда помещается.

1) Выполните команды создания таблиц и скрипта для теста:

```
postgres@tantor:~$ psql -c "drop table if exists t1;"
psql -c "create table t1(s text storage plain) with (autovacuum_enabled=on);"
psql -c "insert into t1 values (repeat('a',2004));"
psql -c "vacuum analyze t1;"
echo "update t1 set s=(repeat('c',2004));" > hot1.sql
psql -c "drop table if exists t2;"
psql -c "create table t2(s text storage plain) with (autovacuum_enabled=on);"
psql -c "insert into t2 values (repeat('a',2005));"
psql -c "vacuum analyze t2;"
echo "update t2 set s=(repeat('c',2005));" > hot2.sql
DROP TABLE
CREATE TABLE
INSERT 0 1
VACUUM
DROP TABLE
CREATE TABLE
INSERT 0 1
VACUUM
```

На таблицах включён автовакуум, в отличие от предыдущей части практики. В предыдущей части мы пошагово изучали как выглядит содержимое блоков после изменений, чтобы понимать что происходит с блоками на низком уровне. Тесты должны быть приближены к условиям реальной эксплуатации.

2) По умолчанию интервал вызова автовакуума 1 минута. Чтобы не тратить время на долгие тесты поставьте интервал вызова автовакуума в 1 секунду. на нагруженных СУБД это рекомендуется делать. На настройки автоанализа не будем обращать внимание, хотя частое обновление статистики обычно бесполезно (собранная статистика такая же как до пересбора) и

снижает производительность. Почему часто считают, что "надо обновлять статистику", "статистика должна быть актуальной"? Потому, что в Oracle Database если не обновить статистику, то это эквивалентно ее отсутствию и эту логику переносят на другие СУБД. В PostgreSQL нет "устаревшей" (stale) статистики, используется та статистика, которая есть.

Выполните команды уменьшающие частоту вызова автовакуума до 1 секунды:

```
postgres@tantor:~$ psql -c "alter system set autovacuum_naptime = '1s';"
pg_ctl reload
ALTER SYSTEM
server signaled
```

3) Запустите тест и посмотрите результат:

```
postgres@tantor:~$ pgbench -T 30-f hot1.sql 2> /dev/null | grep tps
pgbench -T 30-f hot2.sql 2> /dev/null | grep tps
psql -c "select pg_relation_size('t1') t1, pg_relation_size('t2') t2";
psql -c "select relname name, n_tup_upd upd, n_tup_hot_upd hot_upd,
n_tup_newpage_upd newpage_upd, round(n_tup_hot_upd*100/n_tup_upd,2) as hot_ratio,
seq_scan sel, n_tup_ins ins, n_tup_del del, n_live_tup live, n_dead_tup dead,
autovacuum_count vacuum from pg_stat_all_tables where n_tup_upd<>0 and relname in
('t1','t2') order by 1;"
tps = 246.374879 (without initial connection time)
tps = 245.593952 (without initial connection time)
    t1|   t2
------+--------
8192  | 688128
(1 row)

name | upd  | hot_upd | newpage_upd | hot_ratio | sel  | ins | del | live | dead | vacuum
-----+------+---------+-------------+-----------+------+-----+-----+------+------+--------
t1   | 7391 |    7391 |           0 |    100.00 | 7391 |   1 |   0 |    1 |    2 |      0
t2   | 7367 |    4915 |        2452 |     66.00 | 7367 |   1 |   0 |    1 |   77 |     29
(2 rows)
```

Столбец **n_tup_newpage_upd** показывает, что в тесте с таблицей t2 цепочка HOT прерывалась и новые версии строк вставлялись в новые блоки. Процент строк, которые были обновлены HOT меньше - 66%. Столбец **n_tup_del** показывает сколько строк было удалено командой DELETE. Столбец **n_tup_ins** показывает сколько строк было вставлено командой INSERT.

Число чтений строк **seq_scan** методом Seq Scan примерно одинаково.

TPS немного уменьшился и только из-за автовакуума и автоанализа.

Благодаря автовакууму, который срабатывал каждую секунду на таблице t2 таблица раздулась не сильно, всего до 688128 байт. Если бы автовакуум не срабатывал в течение теста, то таблица бы разрослась до **10**444800 байт.

4) Тест проводился в одной сессии. Как поменяются результаты, если таблицы будут обновляться в нескольких сессиях? Выполним такой тест.

Пересоздайте таблицы:

```
postgres@tantor:~$ psql -c "drop table if exists t1;"
psql -c "create table t1(s text storage plain) with (autovacuum_enabled=on);"
psql -c "insert into t1 values (repeat('a',2004));"
psql -c "vacuum analyze t1;"
echo "update t1 set s=(repeat('c',2004));" > hot1.sql
psql -c "drop table if exists t2;"
psql -c "create table t2(s text storage plain) with (autovacuum_enabled=on);"
psql -c "insert into t2 values (repeat('a',2005));"
psql -c "vacuum analyze t2;"
echo "update t2 set s=(repeat('c',2005));" > hot2.sql
DROP TABLE
CREATE TABLE
INSERT 0 1
VACUUM
```

```
DROP TABLE
CREATE TABLE
INSERT 0 1
VACUUM
```

5) Выполните тесты с числом сессий 4 (или по числу ядер процессора):

```
pgbench -T 30 -c 4 -f hot1.sql 2> /dev/null | grep tps
pgbench -T 30 -c 4 -f hot2.sql 2> /dev/null | grep tps
psql -c "select pg_relation_size('t1') t1, pg_relation_size('t2') t2";
psql -c "select relname name, n_tup_upd upd, n_tup_hot_upd hot_upd,
n_tup_newpage_upd newpage_upd, round(n_tup_hot_upd*100/n_tup_upd,2) as hot_ratio,
seq_scan sel, n_tup_ins ins, n_tup_del del, n_live_tup live, n_dead_tup dead,
autovacuum_count vacuum, autoanalyze_count analyze from pg_stat_all_tables where
n_tup_upd<>0 and relname in ('t1','t2') order by 1;"
```

```
tps = 251.225621 (without initial connection time)
tps = 250.060665 (without initial connection time)
  t1   |   t2
-------+---------
 32768 | 2244608
(1 row)

 name | upd  | hot_upd | newpage_upd | hot_ratio | sel  | ins | del | live | dead | vacuum | analyze
------+------+---------+-------------+-----------+------+-----+-----+------+------+--------+---------
 t1   | 7537 |    5480 |        2057 |     72.00 | 7537 |   1 |   0 |    1 |    0 |     10 |      10
 t2   | 7501 |    4873 |        2628 |     64.00 | 7501 |   1 |   0 |    1 |   12 |      9 |       9
(2 rows)
```

TPS одинаковый, так как число срабатываний автовакуума и автоанализа не отличаются.

TPS возрос, что ожидаемо, так как зона максимальных TPS достигается начиная с числа сессий, соответствующих числу ядер процессора.

Различия в проценте HOT уменьшились. Несколько процессов вносили изменения в один и тот же блок и в большей части случаев первый процесс вставлял строку (находился в процессе вставки новой версии строки и изменения блока), а второй процесс уже не мог найти свободное место в блоке.

6) Уменьшим размер строки до 900 байт:

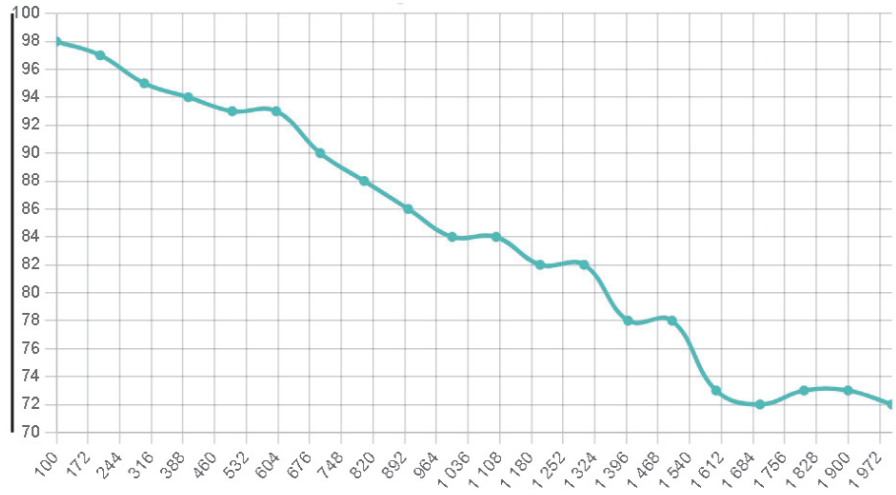
```
psql -c "drop table if exists t1;"
psql -c "drop table if exists t2;"
psql -c "create table t2(s text storage plain) with (autovacuum_enabled=on);"
psql -c "insert into t2 values (repeat('a',900));"
psql -c "vacuum analyze t2;"
echo "update t2 set s=(repeat('c',900));" > hot2.sql
pgbench -T 10 -c 4 -f hot2.sql 2> /dev/null | grep tps
psql -c "select pg_relation_size('t2')";
psql -c "select relname name, n_tup_upd upd, n_tup_hot_upd hot_upd,
n_tup_newpage_upd newpage_upd, round(n_tup_hot_upd*100/n_tup_upd,2) as hot_ratio,
seq_scan sel, n_tup_ins ins, n_tup_del del, n_live_tup live, n_dead_tup dead,
autovacuum_count vacuum, autoanalyze_count analyze from pg_stat_all_tables where
n_tup_upd<>0 and relname='t2' order by 1;"
```

```
tps = 251.986305 (without initial connection time)
 pg_relation_size
------------------
           524288
(1 row)

 name | upd  | hot_upd | newpage_upd | hot_ratio | sel  | ins | del | live | dead | vacuum | analyze
------+------+---------+-------------+-----------+------+-----+-----+------+------+--------+---------
 t2   | 2519 |    2184 |         335 |     86.00 | 2519 |   1 |   0 |    1 |   41 |      4 |       4
(1 row)
```

Результат улучшился. Можно продолжить менять длинну строки, но не будем на это тратить время. Достаточно посмотреть результат в зависимости от числа символов в поле:

длина	upd	hot_upd	newpage_upd	hot_ratio
100 :	2519	2478	41	98.00
200 :	2513	2439	74	97.00
300 :	2536	2429	107	95.00
400 :	2523	2379	144	94.00
500 :	2599	2421	178	93.00
600 :	2523	2348	175	93.00
700 :	2515	2278	237	90.00
800 :	2509	2218	291	88.00
900 :	2529	2195	334	86.00
1000 :	2521	2131	390	84.00
1100 :	2513	2132	381	84.00
1200 :	2532	2089	443	82.00
1300 :	2523	2076	447	82.00
1400 :	2503	1953	550	78.00
1500 :	2860	2252	608	78.00
1600 :	2891	2115	776	73.00
1700 :	2878	2091	787	72.00
1800 :	2930	2168	762	73.00
1900 :	2945	2175	770	73.00

График:



Наиболее эффективно при проектировании схем хранения данных делать размер строк небольшими. Судя по графику, разумный размер строки 600 байт.

Можно провести тесты меняя значение FILLFACTOR и убедиться, что на вероятность HOT cleanup он не окажет влияния. Установка FILLFACTOR в значение, отличное от значения по умолчанию (FILLFACTOR=100) в большинстве случаев бесполезно и только увеличивает размер файлов таблицы.

Тест можно было бы приблизить к реальности, вставив большое число строк в таблицу, чтобы использовался индексный доступ, но это затруднит выявление зависимостей.

Часть 6. Типы данных небольшого размера

1) Посмотрите список типов данных:

```
postgres=# select typname, typalign, typstorage, typcategory, typlen from
pg_type where typtype='b' and typcategory <>'A' order by typlen,typalign,typname;
          typname           | typalign | typstorage | typcategory | typlen
----------------------------+----------+------------+-------------+--------
 path                       | d        | x          | G           |     -1
 pg_snapshot                | d        | x          | U           |     -1
 polygon                    | d        | x          | G           |     -1
 txid_snapshot              | d        | x          | U           |     -1
```

```
 bit                          | i          | x           | V             |          -1
 bpchar                       | i          | x           | S             |          -1
 bytea                        | i          | x           | U             |          -1
 cidr                         | i          | m           | I             |          -1
 gtsvector                    | i          | p           | U             |          -1
 inet                         | i          | m           | I             |          -1
 json                         | i          | x           | U             |          -1
 jsonb                        | i          | x           | U             |          -1
 jsonpath                     | i          | x           | U             |          -1
 numeric                      | i          | m           | N             |          -1
 pg_brin_bloom_summary        | i          | x           | Z             |          -1
 pg_brin_minmax_multi_summary | i          | x           | Z             |          -1
 pg_dependencies              | i          | x           | Z             |          -1
 pg_mcv_list                  | i          | x           | Z             |          -1
 pg_ndistinct                 | i          | x           | Z             |          -1
 pg_node_tree                 | i          | x           | Z             |          -1
 refcursor                    | i          | x           | U             |          -1
 text                         | i          | x           | S             |          -1
 tsquery                      | i          | p           | U             |          -1
 tsvector                     | i          | x           | U             |          -1
 varbit                       | i          | x           | V             |          -1
 varchar                      | i          | x           | S             |          -1
 xml                          | i          | x           | U             |          -1
 bool                         | c          | p           | B             |           1
 char                         | c          | p           | Z             |           1
 int2                         | s          | p           | N             |           2
 cid                          | i          | p           | U             |           4
 date                         | i          | p           | D             |           4
 float4                       | i          | p           | N             |           4
 int4                         | i          | p           | N             |           4
 oid                          | i          | p           | N             |           4
 regclass                     | i          | p           | N             |           4
 regcollation                 | i          | p           | N             |           4
 regconfig                    | i          | p           | N             |           4
 regdictionary                | i          | p           | N             |           4
 regnamespace                 | i          | p           | N             |           4
 regoper                      | i          | p           | N             |           4
 regoperator                  | i          | p           | N             |           4
 regproc                      | i          | p           | N             |           4
 regprocedure                 | i          | p           | N             |           4
 regrole                      | i          | p           | N             |           4
 regtype                      | i          | p           | N             |           4
 macaddr                      | i          | p           | U             |           6
 tid                          | s          | p           | U             |           6
 float8                       | d          | p           | N             |           8
 int8                         | d          | p           | N             |           8
 money                        | d          | p           | N             |           8
 pg_lsn                       | d          | p           | U             |           8
 time                         | d          | p           | D             |           8
 timestamp                    | d          | p           | D             |           8
 timestamptz                  | d          | p           | D             |           8
 xid8                         | d          | p           | U             |           8
 macaddr8                     | i          | p           | U             |           8
 xid                          | x          | p           | U             |           8
 timetz                       | d          | p           | D             |          12
 uuid                         | c          | p           | U             |          16
 aclitem                      | d          | p           | U             |          16
 interval                     | d          | p           | T             |          16
 point                        | d          | p           | G             |          16
 circle                       | d          | p           | G             |          24
 line                         | d          | p           | G             |          24
 box                          | d          | p           | G             |          32
 lseg                         | d          | p           | G             |          32
 name                         | c          | p           | S             |          64
(68 rows)
```

typlen=-1 типы данных переменной ширины
typalign=i выравнивание по 4 байта, d - по 8 байт
typstorage режим хранения по умолчанию
Типы данных bool и "char" имеют фиксированный размер 1 байт.
bpchar(N) это синоним char(N).

2) Можно спутать "char" с char (синоним character(1) или char(1)). char занимает 2 байта, а не 1, но хранит символы в кодировке базы данных, то есть символов больше, чем в кодировке ASCII:

```
 drop table if exists t5;
 create table t5( c1 "char" default '1');
 insert into t5 values(default);
 select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;

 lp_off | lp_len | t_hoff | t_data
--------+--------+--------+--------
   8144 |     25 |     24 | \x31

 drop table if exists t5;
 create table t5( c1 char default '1');
 insert into t5 values(default);
 select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;

 lp_off | lp_len | t_hoff | t_data
--------+--------+--------+--------
   8144 |     26 |     24 | \x0531
```

"char" занимает 1 байт, а char 2 байта. Почему lp_off (смещение внутри блока, указывающее на начало строки) одинаков? Потому, что длинна всей строки выравнивается по 8 байт. "char" предназначен для использования в таблицах системного каталога, но может использоваться в обычных таблицах.

3) Тип boolean также занимает 1 байт, но хранит только два значения true, false и пустое (NULL).

```
postgres=# drop table if exists t5;
 create table t5( c1 boolean default true);
 insert into t5 values(default);
 insert into t5 values(false);
 insert into t5 values(NULL);
 select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
 DROP TABLE
 CREATE TABLE
 INSERT 0 1
 INSERT 0 1
 INSERT 0 1
  lp_off | lp_len | t_hoff | t_data
 --------+--------+--------+--------
    8088 |     24 |     24 | \x
    8112 |     25 |     24 | \x00
    8144 |     25 |     24 | \x01
 (3 rows)
```

4) Посмотрите как хранятся строки таблицы с 2-байтовыми столбцами фиксированной длинны типа `int2` (синоним `smallint`):

```
postgres=# drop table if exists t5;
 create table t5( c1 int2 default 1);
 insert into t5 values(default);
 insert into t5 values(2);
 select lp, lp_off, lp_len, t_ctid, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
 DROP TABLE
 CREATE TABLE
 INSERT 0 1
 INSERT 0 1
```

```
 lp | lp_off | lp_len | t_ctid | t_hoff | t_data
----+--------+--------+--------+--------+--------
  2 |   8112 |     26 | (0,2)  |     24 | \x0200
  1 |   8144 |     26 | (0,1)  |     24 | \x0100
(2 rows)
```

Размер строки: 8144-8112= 32 байта
lp - номер слота в заголовке блока

```
postgres=# drop table if exists t5;
create table t5(c1 int2, c2 int2, c3 int2, c4 int2);
insert into t5 values(1,2,3,4);
insert into t5 values(5,6,7,8);
select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
DROP TABLE
CREATE TABLE
INSERT 0 1
INSERT 0 1
 lp_off | lp_len | t_hoff |       t_data
--------+--------+--------+--------------------
   8112 |     32 |     24 | \x0500060007000800
   8144 |     32 |     24 | \x0100020003000400
(2 rows)
```

Размер строки останется тем же: 8144-8112= 32 байта.

```
postgres=# drop table if exists t5;
create table t5(c1 int2, c2 int2, c3 int2, c4 int2, c5 int2);
insert into t5 values(1,2,3,4,5);
select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
DROP TABLE
CREATE TABLE
INSERT 0 1
 lp_off | lp_len | t_hoff |         t_data
--------+--------+--------+------------------------
   8136 |     34 |     24 | \x01000200030004000500
(1 row)
```

Начиная с пятого столбца размер строки увеличился сразу на 8 байт: 8144-8136=8 байт.
Пропуски (padding) между строками использовать под хранение данных невозможно.
Вывод из примера: лучше создать таблицу с 4 столбцами **(c1 int2, c2 int2, c3 int2, c4 int2)**, чем с одним столбцом **(c1 int2)**, так как размер таблиц будет одинаков, при этом в первой таблице можно сохранить больше данных.
Пример полезен для того, чтобы понимать как рассчитать по значениям в `lp_off` полный размер строки; как в поле `t_data` отображаются значения использованных типов данных.

Часть 7. Хранение типов данных переменной длинны

У типов переменной длинны в столбце typlen таблицы pg_type отрицательное число: -1 обозначает тип varlena.

1) Посмотрите характеристики типа text:

```
postgres=# select * from pg_type where typname='text'\gx
-[ RECORD 1 ]--+---------
oid            | 25
typname        | text
typnamespace   | 11
typowner       | 10
typlen         | -1  тип переменной длинны varlena
```

```
typbyval        | f    для типов переменной длины всегда f
typtype         | b    базовый тип, а не составной, не интервальный
typcategory     | S    строковый тип
typispreferred  | t    является предпочтительным в категории S строковых типов
typisdefined    | t
typdelim        | ,    разделитель значений этого типа при парсинге массива значений этого типа
typrelid        | 0
typsubscript    | -
typelem         | 0
typarray        | 1009
typinput        | textin
typoutput       | textout
typreceive      | textrecv название функции преобразования двоичного значения в этот тип
typsend         | textsend название функции преобразования в двоичный формат
typmodin        | -
typmodout       | -
typanalyze      | -
typalign        | i    выравнивается до 4 байт
typstorage      | x    по умолчанию режим хранения EXTENDED (сжатие, потом перемещение в TOAST)
typnotnull      | f
typbasetype     | 0
typtypmod       | -1
typndims        | 0
typcollation    | 100
typdefaultbin   |
typdefault      |
typacl          |
```

Описание столбцов представления **pg_type** есть в документации
https://docs.tantorlabs.ru/tdb/ru/16_6/se/catalog-pg-type.html

2) Проверьте, выравниваются ли поля типа text, если их длинна до 126 байт:

```
postgres=# drop table if exists t;
create table t(c1 text, c2 text);
insert into t values (repeat('a',1), repeat('b',1));
insert into t values (repeat('a',2), repeat('b',2));
insert into t values (repeat('a',3), repeat('b',3));
insert into t values (repeat('a',4), repeat('b',4));
insert into t values (repeat('a',5), repeat('b',5));
insert into t values (repeat('a',6), repeat('b',6));
insert into t values (repeat('a',7), repeat('b',7));
insert into t values (repeat('a',8), repeat('b',8));
select pg_column_size(t.*), pg_column_size(c1) from t;
select lp_off, lp_len, t_data from heap_page_items(get_raw_page('t', 0));
DROP TABLE
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
 pg_column_size | pg_column_size
----------------+----------------
             28 |              2
             30 |              3
             32 |              4
             34 |              5
             36 |              6
             38 |              7
             40 |              8
             42 |              9
(8 rows)

 lp_off | lp_len |                  t_data
--------+--------+-----------------------------------------
   8144 |     28 | \x0561056 2
   8112 |     30 | \x0761610762 62
```

```
   8080 |      32 | \x09616161 09626262
   8040 |      34 | \x0b61616161 0b62626262
   8000 |      36 | \x0d6161616161 0d6262626262
   7960 |      38 | \x0f616161616161 0f626262626262
   7920 |      40 | \x1161616161616161 1162626262626262
   7872 |      42 | \x136161616161616161 136262626262626262
(8 rows)
```

'61' это символ 'a' в формате HEX.
'62' это символ 'b' в формате HEX.
'00' (padding нулями) между полями нет, это означает, что **выравнивание полей типа text размером до 127 байт отсутствует**.

В первом байте хранится длинна (30 бит) и 2 бита: если биты 00, то длинна меньше 127 байт. Значение \x03 означает пустую строку; \x05 строку длинной 1 байт; \x07 строку длинной 2 байта.

Хранение длинны нужно, чтобы поддерживать TOAST. В поле любого типа varlena первый байт или первые 4 байта всегда (даже если размер поля небольшой и не вытеснен в TOAST) содержат общую длину поля в байтах (включая эти 4 байта). Причем, эти байты могут (но не всегда) быть сжаты вместе с данными, то есть храниться в сжатом виде. Один байт используется, если длинна поля не превышает 126 байт. Поэтому, при хранении данных поля размером до 127 байт "экономится" три байта на каждой версии строки, а также отсутствует выравнивание, на чем можно сэкономить до 7 байт. Поля varlena с одним байтом длинны не выравниваются, а поля с 4 байтами выравниваются, что будет проверяться в следующих пунктах.

3) Проверьте, по сколько байт выравниваются поля типа text, если их длинна от 127 байт:

```
postgres=# drop table if exists t;
create table t(c1 text, c2 text);
insert into t values (repeat('a',127), repeat('b',127));
insert into t values (repeat('a',128), repeat('b',128));
insert into t values (repeat('a',129), repeat('b',129));
insert into t values (repeat('a',130), repeat('b',130));
insert into t values (repeat('a',131), repeat('b',131));
insert into t values (repeat('a',132), repeat('b',132));
insert into t values (repeat('a',133), repeat('b',133));
insert into t values (repeat('a',134), repeat('b',134));
select pg_column_size(t.*), pg_column_size(c1) from t;
select lp_off, lp_len, substring(t_data from 130 for 18) from
heap_page_items(get_raw_page('t', 0));
DROP TABLE
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
 pg_column_size
----------------
 pg_column_size | pg_column_size
----------------+----------------
            287 |            131
            288 |            132
            293 |            133
            294 |            134
            295 |            135
            296 |            136
            301 |            137
            302 |            138
(8 rows)

 lp_off | lp_len |                  substring
--------+--------+--------------------------------------------
   7888 |    287 | \x616100 0c020000 6262626262626262626262
   7600 |    288 | \x616161 10020000 6262626262626262626262
```

```
   7304 |     293 | \x6161616100000014020000626262626262
   7008 |     294 | \x6161616161000018020000626262626262
   6712 |     295 | \x616161616161001c020000626262626262
   6416 |     296 | \x6161616161616120020000626262626262
   6112 |     301 | \x616161616161616100000024020000626262
   5808 |     302 | \x616161616161616161000028020000626262
(8 rows)
```

После значений '61' и до '62' идёт padding (нули) и 4 байта заголовок второго (столбец c2) поля. Красным цветом выделены нули-заполнители, чтобы первое поле (столбец c1) было выровнено. Максимальное число заполнителей три, значит выравнивание по 4 байта, а не по 8 байт.
Поля типа text размером от 127 байт выравниваются по 4 байта, что соответствует значению `pg_type.typalign=i`.

4) Проверьте, изменится ли выравнивание по 4 байта, если третьим столбцом добавить int8, который выравнивается по 8 байт?

```
postgres=# drop table if exists t;
create table t(c1 text, c2 text, c3 int8);
insert into t values (repeat('a',127), repeat('b',127), 111);
insert into t values (repeat('a',128), repeat('b',128), 111);
insert into t values (repeat('a',129), repeat('b',129), 111);
insert into t values (repeat('a',130), repeat('b',130), 111);
insert into t values (repeat('a',131), repeat('b',131), 111);
insert into t values (repeat('a',132), repeat('b',132), 111);
insert into t values (repeat('a',133), repeat('b',133), 111);
insert into t values (repeat('a',134), repeat('b',134), 111);
select pg_column_size(t.*), pg_column_size(c1) from t;
select lp_off, lp_len, substring(t_data from 130 for 18) from
heap_page_items(get_raw_page('t', 0));
```

```
 pg_column_size | pg_column_size
----------------+----------------
            296 |            131
            296 |            132
            304 |            133
            304 |            134
            304 |            135
            304 |            136
            312 |            137
            312 |            138
(8 rows)
```

```
 lp_off | lp_len |                substring
--------+--------+-----------------------------------------
   7880 |    296 | \x6161000c020000626262626262626262626262
   7584 |    296 | \x6161611002000062626262626262626262626262
   7280 |    304 | \x6161616100000014020000626262626262
   6976 |    304 | \x6161616161000018020000626262626262
   6672 |    304 | \x616161616161001c020000626262626262
   6368 |    304 | \x6161616161616120020000626262626262
   6056 |    312 | \x616161616161616100000024020000626262
   5744 |    312 | \x616161616161616161000028020000626262
(8 rows)
```

Добавление столбца int8 справа на выравнивание text не повлияло. Также не повлияет добавление столбца с выравниванием по 16 байт вместо столбца int8 ни слева, ни справа, ни одновременно слева и справа:

```
create table t(c1 text, c2 text, c3 uuid default gen_random_uuid());
insert into t values (repeat('a',127), repeat('b',127), default);
create table t(c3 uuid default gen_random_uuid(), c1 text, c2 text);
insert into t values (default, repeat('a',127), repeat('b',127));
```

```
create table t(c3 uuid default gen_random_uuid(), c1 text, c2 text, c4 uuid
default gen_random_uuid());
insert into t values (default, repeat('a',127), repeat('b',127), default);
```

выравнивание столбца c1 останется по 4 байта.

5) Посмотрите как хранятся в блоке таблицы текстовые поля, вынесенные в TOAST и выравниваются ли они:

```
postgres=# drop table if exists t;
create table t (c1 text storage external, c3 boolean, c2 text storage external);
insert into t values (repeat('a',2005), false, '');
insert into t values (repeat('a',2005), true, repeat('b',2001));
insert into t values (repeat('a',2005), false, repeat('b',2002));
select pg_column_size(t.*), pg_column_size(c1), pg_column_size(c3),
pg_column_size(c2) from t;
select lp_off, lp_len, t_data from heap_page_items(get_raw_page('t', 0));

DROP TABLE
CREATE TABLE
ALTER TABLE
ALTER TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
 pg_column_size | pg_column_size | pg_column_size | pg_column_size
----------------+----------------+----------------+----------------
           2035 |           2005 |              1 |              1
           4041 |           2005 |              1 |           2001
           4042 |           2005 |              1 |           2002
(3 rows)

lp_off|lp_len|                                        t_data
------+------+-----------------------------------------------------------------
  8128|    44| \x0112d9070000d5070000035b0800015b08000003
  8064|    61| \x0112d9070000d5070000045b0800015b0800010112d5070000d1070000055b0800015b0800
  8000|    61| \x0112d9070000d5070000065b0800015b0800000112d6070000d2070000075b0800015b0800
(3 rows)
```

Поля не выравнивались.
Длинна полей, вынесенных в TOAST, 18 байт.
Функция **pg_column_size(c1)** для полей, вынесенных в TOAST выдаёт размер области данных этого поля.
Рассмотрим что хранится в 18 байта:
первый байт имеет значение 0x01, это признак того, что поле вынесено в TOAST;
второй байт - длинна этой записи (значение 0x12 = 18 байт);
4 байта длинна поля с заголовком поля до сжатия;
4 байта длинна того, что вынесено в TOAST;
4 байта - указатель на первый чанк в TOAST (столбец chunk_id таблицы TOAST);
4 байта - oid toast-таблицы (pg_class.reltoastrelid). TOAST таблица одна и та же для всех полей несекционированной таблицы.

6) Пример хранения текстовых полей в сжатом виде без выноса в TOAST:

```
postgres=# drop table if exists t;
create table t (c1 text storage main, c3 boolean, c2 text storage main);
insert into t values (repeat('a',2003), false, '');
insert into t values (repeat('a',1984), true, repeat('b',1984));
select pg_column_size(t.*), pg_column_size(c1), pg_column_size(c3),
pg_column_size(c2) from t;
select lp_off, lp_len, t_data from heap_page_items(get_raw_page('t', 0))\gx
DROP TABLE
CREATE TABLE
ALTER TABLE
ALTER TABLE
```

```
INSERT 0 1
INSERT 0 1
 pg_column_size | pg_column_size | pg_column_size | pg_column_size
----------------+----------------+----------------+----------------
             65 |             39 |              1 |              1
            103 |             39 |              1 |             39
(2 rows)

-[ RECORD 1]-
lp_off | 8104
lp_len | 65
t_data | \x9e000000d3070000fe610f01ff0f01ff0f01ff0f01ff0f01ff0f01ff0f01ff010f064561616161610003
-[ RECORD 2]-
lp_off | 8000
lp_len | 103
t_data | \x9e000000c0070000fe610f01ff0f01ff0f01ff0f01ff0f01ff0f01ff0f01ff010f08326161616161 01
         9e000000c0070000fe620f01ff0f01ff0f01ff0f01ff0f01ff0f01ff0f01ff010f083262626262
```

7) Хранение кириллицы. Создайте таблицу и вставьте строку с кириллическим знаком:

```
drop table if exists t5;
create table t5(c1 text default '1',c2 text default 'э', c3 text default '');
insert into t5 values(default, default, default);
select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
DROP TABLE
CREATE TABLE
INSERT 0 1
 lp_off | lp_len | t_hoff |     t_data
--------+--------+--------+----------------
   8144 |     30 |     24 | \x053107d18d03
(1 row)
```

Кириллический знак занимает 2 байта: **d18d**
Знак ASCII занимает 1 байт.
Первый байт в каждом поле (**05 07 03**) хранит размер поля.

Часть 8. Тип данных для столбца первичного ключа

1) Сравнение использования bigint и uuid в качестве первичного ключа:

```
postgres@tantor:~$
psql -c "drop table if exists tt1;"
psql -c "create table tt1 (id bigint generated by default as identity (cache 60) primary
key, data bigint);"
echo "insert into tt1(data) values(1);" >txn.sql
psql -c "vacuum analyze tt1;"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from tt1;"
psql -c "reindex table tt1;"
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from tt1;"
psql -c "drop table if exists tt1;"
psql -c "create table tt1 (id uuid default gen_random_uuid() primary key, data bigint);"
psql -c "vacuum analyze tt1;"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from tt1;"
psql -c "reindex table tt1;"
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from tt1;"

DROP TABLE
CREATE TABLE
VACUUM
tps = 556.855615 (without initial connection time)
 count | pg_indexes_size | pg_table_size
```

```
-------+-----------------+---------------
 16698 |          589824 |        786432
(1 row)

REINDEX
 count | pg_indexes_size | pg_table_size
-------+-----------------+---------------
 16698 |          393216 |        786432
(1 row)

DROP TABLE
CREATE TABLE
VACUUM
tps = 553.336293 (without initial connection time)
 count | pg_indexes_size | pg_table_size
-------+-----------------+---------------
 16593 |          622592 |        901120
(1 row)

REINDEX
 count | pg_indexes_size | pg_table_size
-------+-----------------+---------------
 16593 |          540672 |        901120
(1 row)
```

Скорость вставки соизмерима. Вставки в индекс при использовании uuid менее эффективна, размер индекса больше.

После перестроения индексов их размер уменьшился. **Из-за параметра `cache=60` вставка в индекс была неэффективна**. Часть вставок шла не в правй блок из-за того, что параллельные процессы (параметр `-c=4`) успевали разделить правый блок, а другие процессы кэшировали значения, которые вынуждены были вставляться в более левые блоки.

2) Уменьшите cache до значения по умолчанию cache=1 и повторите тест:

```
postgres@tantor:~$
psql -c "drop table if exists tt1;"
psql -c "create table tt1 (id bigint generated by default as identity (cache 1)
primary key, data bigint);"
echo "insert into tt1(data) values(1);" >txn.sql
psql -c "vacuum analyze tt1;"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from
tt1;"
psql -c "reindex table tt1;"
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from
tt1;"
psql -c "drop table if exists tt1;"
psql -c "create table tt1 (id uuid default gen_random_uuid() primary key, data
bigint);"
psql -c "vacuum analyze tt1;"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from
tt1;"
psql -c "reindex table tt1;"
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from
tt1;"
DROP TABLE
CREATE TABLE
VACUUM
tps = 622.324117 (without initial connection time)
 count | pg_indexes_size | pg_table_size
-------+-----------------+---------------
```

```
 18664 |          434176 |        868352
(1 row)

REINDEX
 count | pg_indexes_size | pg_table_size
-------+-----------------+---------------
 18664 |          434176 |        868352
(1 row)

DROP TABLE
CREATE TABLE
VACUUM
tps = 639.807107 (without initial connection time)
 count | pg_indexes_size | pg_table_size
-------+-----------------+---------------
 19185 |          778240 |       1048576
(1 row)

REINDEX
 count | pg_indexes_size | pg_table_size
-------+-----------------+---------------
 19185 |          622592 |       1048576
(1 row)
```

Индекс по столбцу **bigint** остался эффективным, так как размер индекса после перестройки не уменьшился.

При меньшем числе строк размер индекса по столбцу типа uuid существенно больше, чем размер индекса по столбцу типа bigint.

Причина в том, что размер поля типа uuid 16 байт (128 бит, размер удобен для использования в качестве ключа шифрования) в два раза больше, чем размер поля bigint (8 байт).

Для генерации уникальных значений для первичного ключа стоит использовать последовательности. Для бизнес-задач могут использоваться другие форматы. Например, "booking reference" (бронирование), который использует ~30 знаков (знаки алфавита в верхнем регистре ABCDEFGHJKMNPQRSTUXY и цифры).

Часть 9. Типы данных для хранения дат и времени

Типы данных timestamp, timestamptz хранят время и дату с точностью до микросекунды, занимают 8 байт. Оба типа не хранят часовой пояс, значения физически хранятся в одинаковом виде.

1) Тип данных timestamptz отображает и выполняет вычисления во временной зоне, задаваемой параметром timezone. Посмотрите параметр конфигурации timezone:

```
postgres=# show timezone;
   TimeZone
---------------
 Europe/Moscow
(1 row)
```

2) Создайте таблицу со столбцами типов timestamp и timestamptz:

```
postgres=# drop table if exists t;
create table t(t TIMESTAMP, ttz TIMESTAMPTZ);
insert into t values (CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);
select t, ttz from t;
DROP TABLE
CREATE TABLE
```

```
INSERT 0 1
             t              |              ttz
----------------------------+-------------------------------
 2025-01-02 20:00:24.87259 | 2025-01-02 20:00:24.87259+03
(1 row)
```

3) Поменяйте параметр timezone на уровне сессии:

```
postgres=# set timezone='UTC';
select t, ttz from t;
SET
             t              |              ttz
----------------------------+-------------------------------
 2025-01-02 20:00:24.87259 | 2025-01-02 17:00:24.87259+00
(1 row)
```

Отображаемое значение в столбце типа timestamptz поменялось.

4) Посмотрите значения полей в блоке таблицы:

```
postgres=# select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t','main',0)) order by lp_off;
 lp_off | lp_len | t_hoff |               t_data
--------+--------+--------+-----------------------------------
   8136 |     40 |     24 | \x8e36b061bdcd02008e4af5ddbacd0200
(1 row)
```

Значения физически различны.

5) Сделайте значения полей равными присвоив значению столбца ttz значение из столбца t и посмотрите как поменялось значение в ttz:

```
postgres=# update t set ttz=t;
select lp, lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t','main',0)) order by lp_off;
UPDATE 1
lp | lp_off | lp_len | t_hoff |               t_data
---+--------+--------+--------+-------------------------------
 2 |   8096 |     40 |     24 | \x8e36b061bdcd02008e36b061bdcd0200
 1 |   8136 |     40 |     24 | \x8e36b061bdcd02008e4af5ddbacd0200
(2 rows)
```

Значения в полях обоих столбцов в актуальной версии строки (`lp=2`) стали физически равны.

6) Сейчас значения в столбцах физически равны. Поменяйте timezone с UTC на исходную и посмотрите результат:

```
postgres=# select t, ttz from t;
             t              |              ttz
----------------------------+-------------------------------
 2025-01-02 20:00:24.87259 | 2025-01-02 20:00:24.87259+00
(1 row)

postgres=# RESET timezone;
RESET
postgres=# select t, ttz from t;
             t              |              ttz
----------------------------+-------------------------------
 2025-01-02 20:00:24.87259 | 2025-01-02 23:00:24.87259+03
(1 row)
```

timestamptz физически хранит значения в UTC.
Тип данных timestamp не отображает часовой пояс, не использует временную зону, сохраняет значение как есть (без преобразований).

Часть 10. Типы данных float и real

1) Числа могут выводиться с степенями 10. Степень указывается после знаков "e+". В примере значения одинаковы:

```
postgres=# select 12345.6::float4, '12.3456e+03'::float4,
'123.456e+02'::float4,'1234.56e+01'::float4;
 float4   | float4   | float4   | float4
---------+---------+---------+---------
 12345.6 | 12345.6 | 12345.6 | 12345.6
(1 row)
```

2) Точность float8 15 десятичных разрядов (значащих чисел в десятичной системе счисления).
Точности float4 может быть недостаточно: 6 десятичных разрядов
Последний разряд округляется:

```
postgres=# select 123456789012345678901234567890.1234567890123456789::float4::numeric;
            numeric
-------------------------------
 123457000000000000000000000000
(1 row)

postgres=# select 123456789012345678901234567890.1234567890123456789::float8::numeric;
            numeric
-------------------------------
 123456789012346000000000000000
(1 row)
```

3) На разряды больше 15 и 6 полагаться не стоит, хотя они хранятся (в том числе в строках таблиц) и могут учитываться в вычислениях. Пример отображения дополнительных разрядов:

```
postgres=# select 123456789012345678901234567890.1234567890123456789::float4;
    float4
---------------
 1.2345679e+28
(1 row)

postgres=# select 123456789012345678901234567890.1234567890123456789::float8;
         float8
------------------------
 1.2345678901234568e+28
(1 row)
```

4) Пример округления при вычислениях с типами данных float8 (синоним float) и float4 (синоним real):

```
postgres=# select 234567890.199999989::float8, 1.19999999123::float4;
   float8     | float4
--------------+--------
 234567890.2 |    1.2
(1 row)

postgres=# select 1234567890.123456789::float8, 1.123456789::float4;
       float8        |  float4
---------------------+-----------
```

```
 1234567890.1234567 | 1.1234568
(1 row)

postgres=# select 234567890.199999989::float8::numeric,
1.19999999123::float4::numeric;
   numeric    | numeric
--------------+---------
 234567890.2 |     1.2
(1 row)

postgres=# select 1234567890.123456789::float8::numeric,
1.123456789::float4::numeric;
     numeric       | numeric
-------------------+---------
 1234567890.12346 | 1.12346
(1 row)
```

5) Недостаток float4 и float8 в том, что добавление к большому числу маленького числа эквивалентно добавлению нуля, из-за чего могут проявляться плохо диагностируемые ошибки в приложениях:

```
postgres=# select (12345678901234567890123456789.1234567890123456789::float8 +
123456789::float8);
        ?column?
------------------------
 1.2345678901234568e+28
(1 row)

postgres=# select (12345678901234567890123456789.1234567890123456789::float8 +
123456789::float8)::numeric;
            numeric
-------------------------------
 12345678901234600000000000000
(1 row)
```

15 разряд был округлён при приведении к типу numeric.
Из-за потери точности float4 и float8 не подходят там, где нужно сохранять точность вычислений.

6) Посмотрите как хранятся типы float8, float4, numeric:

```
postgres=# drop table if exists t5;
create table t5( c1 double precision, c2 real, c3 numeric);
insert into t5 values
 (1,1,1),
 (1.0/3, 1.0/3, 1.0/3),
 (1111,1111,1111),
 (1111.11,1111.11, 1111.11);
select lp, lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp;
select pg_column_size(c1), pg_column_size(c2), pg_column_size(c3) from t5;
select * from t5;
DROP TABLE
CREATE TABLE
INSERT 0 4
 lp | lp_off | lp_len | t_hoff |                       t_data
----+--------+--------+--------+----------------------------------------------------
  1 |   8128 |     41 |     24 | \x000000000000f03f0000803f0b00800100
  2 |   8072 |     49 |     24 | \x555555555555d53fabaaaa3e1b7f8a050d050d050d050d050d
  3 |   8024 |     41 |     24 | \x00000000005c914000e08a440b00805704
  4 |   7976 |     43 |     24 | \x3d0ad7a3705c914085e38a440f008157044c04
```

```
 pg_column_size | pg_column_size | pg_column_size
----------------+----------------+----------------
              8 |              4 |              5
              8 |              4 |             13
              8 |              4 |              5
              8 |              4 |              7
(4 rows)

         c1          |     c2      |            c3
---------------------+-------------+--------------------------
                   1 |           1 |                        1
 0.3333333333333333  | 0.33333334  | 0.33333333333333333333
                1111 |        1111 |                     1111
             1111.11 |     1111.11 |                  1111.11
(4 rows)
```

При выборе типа данных для хранения вещественных чисел стоит учесть, что тип numeric имеет переменную длину и для небольших чисел хранит данные компактнее, чем float8: в примере это все строки, кроме второй.

float4 хранит данные компактно и с фиксированной шириной, но у этого типа всего 6 разрядов.

Разрядность типа numeric огромная: 131072 цифр до десятичной точки и 16383 после. Поля типа numeric могут вытесняться в TOAST.

Почему в результате деления `1.0/3` разрядность получившегося числа хоть и больше, чем float8, но всё равно не велика?

7) Работает ли оператор деления с операндами типа numeric? Посмотрите список операторов деления:

```
postgres=# \doS+ /
                                                                 List of operators
   Schema   | Name |  Left arg type   |  Right arg type  |   Result type    |   Function    | Description
------------+------+------------------+------------------+------------------+---------------+-------------
 pg_catalog | /    | bigint           | bigint           | bigint           | int8div       | divide
 pg_catalog | /    | bigint           | integer          | bigint           | int84div      | divide
 pg_catalog | /    | bigint           | smallint         | bigint           | int82div      | divide
 pg_catalog | /    | box              | point            | box       | box_div | divide box by point (scale)
 pg_catalog | /    | circle           | point            | circle           | circle_div_pt | divide
 pg_catalog | /    | double precision | double precision | double precision | float8div     | divide
 pg_catalog | /    | double precision | real             | double precision | float84div    | divide
 pg_catalog | /    | integer          | bigint           | bigint           | int48div      | divide
 pg_catalog | /    | integer          | integer          | integer          | int4div       | divide
 pg_catalog | /    | integer          | smallint         | integer          | int42div      | divide
 pg_catalog | /    | interval         | double precision | interval         | interval_div  | divide
 pg_catalog | /    | money            | bigint           | money            | cash_div_int8 | divide
 pg_catalog | /    | money            | double precision | money            | cash_div_flt8 | divide
 pg_catalog | /    | money            | integer          | money            | cash_div_int4 | divide
 pg_catalog | /    | money            | money            | double precision | cash_div_cash | divide
 pg_catalog | /    | money            | real             | money            | cash_div_flt4 | divide
 pg_catalog | /    | money            | smallint         | money            | cash_div_int2 | divide
 pg_catalog | /    | numeric          | numeric          | numeric          | numeric_div   | divide
 pg_catalog | /    | path             | point            | path     | path_div_pt | divide (rotate/scale path)
 pg_catalog | /    | point            | point            | point    | point_div | divide points (scale/rotate)
 pg_catalog | /    | real             | double precision | double precision | float48div    | divide
 pg_catalog | /    | real             | real             | real             | float4div     | divide
 pg_catalog | /    | smallint         | bigint           | bigint           | int28div      | divide
 pg_catalog | /    | smallint         | integer          | integer          | int24div      | divide
 pg_catalog | /    | smallint         | smallint         | smallint         | int2div       | divide
(25 rows)
```

Оператор "/" работает с операндами типа `numeric` и реализуется функцией `numeric_div(..)`.

8) Посмотрите тело функции `numeric_div(..)`:

```
postgres=# \sf numeric_div
CREATE OR REPLACE FUNCTION pg_catalog.numeric_div(numeric, numeric)
 RETURNS numeric
 LANGUAGE internal
 IMMUTABLE PARALLEL SAFE STRICT
AS $function$numeric_div$function$
```

Функция реализована в коде PostgreSQL, значит реализацию функции нужно смотреть в исходном коде PostgreSQL (функции `numeric_div -> numeric_div_opt_error -> select_div_scale` в файле `numeric.c`).

Число значащих цифр, выдаваемых функцией `numeric_div(..)`:
не менее `NUMERIC_MIN_SIG_DIGITS=16` значащих цифр, то есть не хуже, чем float8;
не меньше, чем разрядность любого из входных параметров.

9) Проверьте утверждение "**не менее масштаба отображения любого из входных параметров**":

```
postgres=# select 1/1.0000000000000000000000000000000001;
                                ?column?
----------------------------------------
0.9999999999999999999999999999999999999
 (1 row)
```

В этом примере разрядность результата больше 16 и такая же, как у правого операнда.

10) При вставке в таблицу действует то же самое правило:

```
postgres=# insert into t5 values (1,1,
1.0000000000000000000000000000000001/3);
select lp, lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp desc limit 1;
select * from t5 order by ctid desc limit 1;
INSERT 0 1
 lp | lp_off | lp_len | t_hoff |                                t_data
----+--------+--------+--------+-----------------------------------------------------------------------
  5 |   7912 |     57 |     24 | \x000000000000f03f0000803f2b7f92050d050d050d050d050d050d050d060d
(1 row)

 c1 | c2 |                  c3
----+----+--------------------------------------
  1 |  1 | 0.33333333333333333333333333333333334
(1 row)
```

Утверждение верно: разрядность результата деления двух типов numeric определяется **наибольшей разрядностью операндов**, но не менее чем 16.

Практика к главе 9

Часть 1. Использование PostgreSQL в контейнере docker

1) Откройте терминал и переключитесь в root.

```
astra@tantor:~$ su -
Password: root
root@tantor:~#
```

Для того, чтобы можно было выполнять команды в терминале пользователей postgres и astra не используя sudo и не переключаясь в root добавьте пользователей astra и postgres в группу с названием docker:

```
root@tantor:~# sudo usermod -aG docker postgres
sudo usermod -aG docker astra
```

Команды в этой части практики можно будет выполнять в терминале пользователей root или postgres или astra.

2) Посмотрите список доступных образов в репозитории docker.io:

```
root@tantor:~# docker search --filter stars=10 --no-trunc postgres
NAME                DESCRIPTION                                     STARS     OFFICIAL
postgres            The PostgreSQL object-relational database sy…   14016     [OK]
circleci/postgres   The PostgreSQL object-relational database sy…   32
ubuntu/postgres     PostgreSQL is an open source object-relation…   40
supabase/postgres   Unmodified Postgres with some useful plugins…   44
debezium/postgres   PostgreSQL for use with Debezium change data…   28
```

У официального образа имя postgres.
Страница с описанием образа: https://hub.docker.com/_/postgres

3) Посмотрите имеется ли образ postgres

```
root@tantor:~# docker images
REPOSITORY   TAG       IMAGE ID       CREATED      SIZE
postgres     latest    de869b203456   2 days ago   494MB
```

4) Если в списке есть образ postgres и доступ в интернет в витуальной машине ограничен, то новый образ можно не скаичвать.
Если список пуст, то скачайте образ:

```
root@tantor:~# docker pull postgres
Using default tag: latest
latest: Pulling from library/postgres
...
Status: Downloaded newer image for postgres:latest
docker.io/library/postgres:latest
```

Был скачан наиболее свежий образ postgres.

У названия образа есть формат:
образ[:TAG][@DIGEST]
где tag - версия образа, по умолчанию latest.

5) Формат команды для создания из образа и запуска контейнера:

```
docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

Если в образе указан ENTRYPOINT, то COMMAND добавится к параметрам ENTRYPOINT

Запустите следующую команду для создания контейнера:

```
root@tantor:~# docker run -d -e POSTGRES_USER=postgres -e
POSTGRES_PASSWORD=postgres -e POSTGRES_INITDB_ARGS="-k" -e
POSTGRES_HOST_AUTH_METHOD=trust -p 5434:5434 -e PGDATA=/var/lib/postgresql/data -
d -v /root/data:/var/lib/postgresql/data --name postgres postgres
```

Описание параметров команды:
-d (--deatach) запуск контейнера в фоновом режиме
--name postgres установить имя **контейнеру**
-e переменные окружения
`-e POSTGRES_USER=postgres` имя пользователя кластера баз данных
`-e POSTGRES_PASSWORD=postgres` задание пароля для пользователя кластера баз данных
`-e POSTGRES_INITDB_ARGS="-k"` параметры утилите initdb, если будет создаваться кластер
`-e POSTGRES_HOST_AUTH_METHOD=trust` способ аутентификации по умолчанию
-v смонтировать директорию /var/lib/postgresql/data внутри контейнера на директорию /root/data
-p (--publish) мапирование портов из контейнера на хост
последний параметр: название **образа**

6) Чтобы проверить, что контейнер успешно запущен, выполните команду:

```
root@tantor:~# docker ps
CONTAINER ID IMAGE    COMMAND                  CREATED        STATUS       PORTS          NAMES
75db63ec0558 postgres "docker-entrypoint.s…" 3 minutes ago Up 3 minutes 5432/tcp,...  postgres
```

В выводе вы увидите список всех запущенных контейнеров.
В списке должен присутствовать контейнер с именем postgres со статусом Up.
Команда полезна тем, что выдаёт `CONTAINER ID=75db63ec0558`, значение которого можно использовать в командах утилиты docker вместо имени контейнера `postgres`.

Команда для просмотра портов контейнера:

```
root@tantor:~# docker port postgres
5434/tcp -> 0.0.0.0:5434
5434/tcp -> [::]:5434
```

тот же самый результат можно получить используя `CONTAINER ID`:

```
root@tantor:~# docker port 75db63ec0558
5434/tcp -> 0.0.0.0:5434
5434/tcp -> [::]:5434
```

Команда для запуска терминала в контейнере:

```
root@tantor:~# docker exec -it postgres bash
```

Команда для остановки контейнера:

```
root@tantor:~# docker stop postgres
```

Команда для запуска контейнера:

```
root@tantor:~# docker start postgres
```

7) Экземпляр в контейнере не ведёт диагностический журнал. Это полезно, чтобы не возникла нехватка места из-за разрастания файлов журнала. По умолчанию параметр конфигурации

`log_destination=stderr` и лог направляется в стандартный поток ошибок, который сохраняется и очищается докером. Команда для просмотра `stderr`:

```
root@tantor:~# docker container logs postgres

PostgreSQL Database directory appears to contain a database; Skipping initialization

08:04:07.353 UTC [1] LOG:  starting PostgreSQL 17.3 (Debian 17.3-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
08:04:07.353 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port 5432
08:04:07.353 UTC [1] LOG:  listening on IPv6 address "::", port 5432
08:04:07.368 UTC [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
08:04:07.387 UTC [29] LOG:  database system was interrupted; last known up at 08:00:29 UTC
08:04:09.351 UTC [29] LOG:  database system was not properly shut down; automatic recovery in progress
08:04:09.361 UTC [29] LOG:  redo starts at 0/177D060
08:04:09.361 UTC [29] LOG:  invalid record length at 0/177D098: expected at least 24, got 0
08:04:09.361 UTC [29] LOG:  redo done at 0/177D060 system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
08:04:09.378 UTC [27] LOG:  checkpoint starting: end-of-recovery immediate wait
08:04:09.428 UTC [27] LOG:  checkpoint complete: wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.016 s, sync=0.007 s, total=0.058 s; sync files=2, longest=0.004 s, average=0.004 s; distance=0 kB, estimate=0 kB; lsn=0/177D098, redo lsn=0/177D098
08:04:09.439 UTC [1] LOG:  database system is ready to accept connections
```

Можно подключиться к выводу и наблюдать за появляющимися сообщениями используя параметр `--follow` или `-f`, для ограничения числа выдаваемых строк можно использовать параметр `--tail`:

```
root@tantor:~# docker container logs -f postgres --tail 1
08:04:09.439 UTC [1] LOG:  database system is ready to accept connections
^c
```

Для возврата промпта достаточно набрать на клавиатуре `<ctrl+c>`.

8) Для того, чтобы изучить как менять параметры конфигурации включите создание диагностического журнала кластера и перезапустите контейнер:

```
root@tantor:~# docker exec -it postgres psql -U postgres -c "alter system set logging_collector=on"
ALTER SYSTEM
root@tantor:~# docker restart postgres
postgres
```

Пока контейнер не в эксплуатации, можно пользоваться простотой докера: быстро перезапускать простой командой контейнер.

9) В контейнере отсутствуют команды диагностики. Например, нет команды `ps`. Установите в контейнере утилиту `ps`, которой можно просматривать списки процессов и их PID:

```
root@tantor:~# docker exec -it postgres apt update
root@tantor:~# docker exec -it postgres  apt install -y procps
```

10) Посмотрите PID процесса postgres:

```
root@tantor:~# docker exec -it postgres /usr/bin/ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
postgres       1       0  0 07:11 pts/0    00:00:00 postgres
postgres      27       1  0 21:23 ?        00:00:00 postgres: logger
postgres      28       1  0 21:23 ?        00:00:00 postgres: checkpointer
postgres      29       1  0 21:23 ?        00:00:00 postgres: background writer
postgres      31       1  0 21:23 ?        00:00:00 postgres: walwriter
postgres      32       1  0 21:23 ?        00:00:00 postgres: autovacuum launcher
postgres      33       1  0 21:23 ?        00:00:00 postgres: logical replication launcher
root          34       0 66 21:23 pts/0    00:00:00 ps -ef
```

У основного процесса экземпляра **PID=1**. **Это нежелательно для экземпляров в промышленной эксплуатации.** Понять или поверить почему это неприемлемо сложно. Посмотрим, что экземпляр может легко сбойнуть и перезапуститься.

11) Запустите в контейнере терминал bash:

```
root@tantor:~# docker exec -it postgres bash
root@1d7f295f6f60:/#
```

12) В запущенном терминале запустите psql и подсоединитесь к базе:

```
root@1d7f295f6f60:/# psql -U postgres
psql (17.2 (Debian 17.2-1.pgdg120+1))
Type "help" for help.

postgres=#
```

13) Дальше нужно отправить psql в фон и закрыть терминал. Наберите на клавиатуре комбинации клавиш **<ctrl+z> <ctrl+d> <ctrl+d>**:

```
postgres=# <ctrl+z>
[1]+  Stopped                 psql -U postgres
root@1d7f295f6f60:/# <ctrl+d>
exit
There are stopped jobs.
root@1d7f295f6f60:/# <ctrl+d>
exit

root@tantor:~#
```

Терминал был закрыт, процесс psql осиротел.

14) Посмотрите PID процесса checkpointer и других процессов экземпляра:

```
root@tantor:~# docker exec -it postgres /usr/bin/ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
postgres       1       0  0 07:11 pts/0    00:00:00 postgres
postgres      27       1  0 21:23 ?        00:00:00 postgres: logger
postgres      50       1  0 21:24 ?        00:00:00 postgres: checkpointer
postgres      51       1  0 21:24 ?        00:00:00 postgres: background writer
postgres      52       1  0 21:24 ?        00:00:00 postgres: walwriter
postgres      53       1  0 21:24 ?        00:00:00 postgres: autovacuum launcher
postgres      54       1  0 21:24 ?        00:00:00 postgres: logical replication launcher
root          55       0 60 21:24 pts/0    00:00:00 ps -ef
```

Процессы экземпляра поменяли номера. Это означает, что они были перезапущены. Процесс postgres перезапустил экземпляр.

Действия по запуску утилиты `psql` в контейнере и ее остановки не должны приводить к перезапуску экземпляра. Последовательность остановки `psql` ,которая была использована, **<ctrl+z> <ctrl+d> <ctrl+d>** это один из примеров. Любая программа или процесс внутри контейнера может самостоятельно сбойнуть. Если у процесса исчезает родительский процесс, то его родителем становится процесс с номером 1. Появление у процесса postgres неизвестному ему дочернего процесса приводит к принудительному перезапуску экземпляра.

При отсутствии утилит ps (устанавливали отдельно), незнании на что смотреть в диагностическом логе (изменение номеров процессов экземпляра) сложно обнаружить перезапуски экземпляров в контейнерах.

15) То что экземпляр перезапустился можно посмотреть по диагностическому журналу:

```
root@tantor:~# tail -20 ~/data/log/postgresql-*.log
```

```
 21:23:03.887 UTC [1] LOG:  starting PostgreSQL 17.2 (Debian 17.2-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled
by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
 21:23:03.887 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port 5432
 21:23:03.887 UTC [1] LOG:  listening on IPv6 address "::", port 5432
 21:23:03.906 UTC [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
 21:23:03.927 UTC [30] LOG:  database system was shut down at 21:22:59 UTC
 21:23:03.944 UTC [1] LOG:  database system is ready to accept connections
 21:24:06.851 UTC [1] LOG:  server process (PID 46) was terminated by signal 15: Terminated
 21:24:06.851 UTC [1] LOG:  terminating any other active server processes
 21:24:06.862 UTC [1] LOG:  all server processes terminated; reinitializing
 21:24:06.888 UTC [49] LOG:  database system was interrupted; last known up at 21:23:03 UTC
 21:24:08.858 UTC [49] LOG:  database system was not properly shut down; automatic recovery in progress
 21:24:08.869 UTC [49] LOG:  redo starts at 0/177D760
 21:24:08.869 UTC [49] LOG:  invalid record length at 0/177D798: expected at least 24, got 0
 21:24:08.869 UTC [49] LOG:  redo done at 0/177D760 system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed:
0.00 s
 21:24:08.892 UTC [50] LOG:  checkpoint starting: end-of-recovery immediate wait
 21:24:08.962 UTC [50] LOG:  checkpoint complete: wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0
recycled; write=0.016 s, sync=0.006 s, total=0.081 s; sync files=2, longest=0.004 s, average=0.003 s; distance=0
kB, estimate=0 kB; lsn=0/177D798, redo lsn=0/177D798
 21:24:08.973 UTC [1] LOG:  database system is ready to accept connections
```

или командой:

```
root@tantor:~# docker container logs postgres --tail 20
```

Дальше посмотрим как создавать и запускать контейнер docker, чтобы PID не был равен 1.

15) Остановите и удалите созданный контейнер:

```
root@tantor:~# docker rm -f postgres
docker network prune -f
docker volume prune -f
docker network create postgres

postgres
Total reclaimed space: 0B
b4785b7d1814d39fb2ca92de80f5e248fffe22ce0181095ceedfc1baf20c7d3d
```

Длинное HEX-значение - идентификатор объекта в docker.
Команда **docker network prune -f** удаляет неиспользуемые контейнерами сети. Таких сетей не было создано.
Команда **docker volume prune -f** удаляет неиспользуемые контейнерами тома.
Команда **docker network create postgres** создаёт сеть. IP адрес генерируется автоматически.

```
root@tantor:~# ifconfig | head -2
br-b4785b7d1814: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.18.0.1  netmask 255.255.0.0  broadcast 172.27.255.255
```

16) Создайте контейнер с опцией **--init**:

```
root@tantor:~# docker run -d --init  --network postgres -e DB_HOST=127.0.0.1 -e
POSTGRES_USER=postgres -e POSTGRES_PASSWORD=postgres -e POSTGRES_INITDB_ARGS="--
data-checksums" -e POSTGRES_HOST_AUTH_METHOD=trust -p 5434:5434 -e
PGDATA=/var/lib/postgresql/data -d -v /root/data:/var/lib/postgresql/data --name
postgres postgres
```

You can use the **--init** flag to indicate that an init process should be used as the PID 1 in the container. Specifying an init process ensures the usual responsibilities of an init system, such as reaping zombie processes, are performed inside the created container. The default init process used is the first docker-init executable found in the system path of the Docker daemon process. This docker-init binary, included in the default installation, is backed by `tini`.

17) Посмотрите ip адрес, назначенный созданной сети, которая была назначена контейнеру:

```
root@tantor:~# docker inspect postgres | grep 172
                    "Gateway": "172.18.0.1",
                    "IPAddress": "172.18.0.2",
```

18) Подсоединиться к экземпляру в контейнере можно с хоста по ip-адресу сети контейнера. Для этого нужно указать ip-адрес контейнера и порт внутри контейнера, а это порт 5432, а не 5434:

```
root@tantor:~# psql -h 172.18.0.2 -p 5432 -U postgres -c "show port; select
version();"
 port
------
 5432
(1 row)
                                  version
-------------------------------------------------------------------------------
 PostgreSQL 17.2 (Debian 17.2-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled by gcc
(Debian 12.2.0-14) 12.2.0, 64-bit
(1 row)
```

Этот пункт иллюстриритрует то, что необязательно использовать команды **sudo docker exec**, чтобы подсоединиться к экзмпляру в контейнере. Подсоединяться к экземпляру в контейнере можно по сетевому (ip) адресу, если контейнеру была назначена сеть.

19) Установите утилиту ps и посмотрите PID процесса postgres:

```
root@tantor:~# docker exec -it postgres apt update
docker exec -it postgres  apt install -y procps
docker exec -it postgres /usr/bin/ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 21:27 ?        00:00:00 /sbin/docker-init -- docker-entrypoint.sh postgres
postgres       7       1  0 21:27 ?        00:00:00 postgres
postgres      28       7  0 21:27 ?        00:00:00 postgres: logger
postgres      29       7  0 21:27 ?        00:00:00 postgres: checkpointer
postgres      30       7  0 21:27 ?        00:00:00 postgres: background writer
postgres      32       7  0 21:27 ?        00:00:00 postgres: walwriter
postgres      33       7  0 21:27 ?        00:00:00 postgres: autovacuum launcher
postgres      34       7  0 21:27 ?        00:00:00 postgres: logical replication launcher
root         253       0 50 21:28 pts/0    00:00:00 /usr/bin/ps -ef
```

У основного процесса экземпляра **PID=7**.
Номер процесса 1 у процесса `/sbin/docker-init -- docker-entrypoint.sh postgres`

У контейнера есть ENTRYPOINT=`docker-entrypoint.sh` и параметр, который мы передавали при создании контейнера стал параметром, переданным скрипту `docker-entrypoint.sh`.

Скрипт находится внутри контейнера в директории **/usr/local/bin**:

```
root@tantor:~# docker exec -it postgres ls /usr/local/bin
docker-enforce-initdb.sh  docker-ensure-initdb.sh  docker-entrypoint.sh  gosu
```

20) Повторите действия, приведшие к перезапуску экземпляра:

```
root@tantor:~# docker exec -it postgres bash
root@1d7f295f6f60:/# psql -U postgres
psql (17.2 (Debian 17.2-1.pgdg120+1))
Type "help" for help.

postgres=# <ctrl+z>
[1]+  Stopped                 psql -U postgres
root@1d7f295f6f60:/# <ctrl+d>
exit
There are stopped jobs.
```

```
root@1d7f295f6f60:/# <ctrl+d>
exit

root@tantor:~#
```

21) Посмотрите PID процесса checkpointer и других процессов экземпляра:

```
root@tantor:~# docker exec -it postgres /usr/bin/ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 21:27 ?        00:00:00 /sbin/docker-init -- docker-entrypoint.sh postgres
postgres       7       1  0 21:27 ?        00:00:00 postgres
postgres      28       7  0 21:27 ?        00:00:00 postgres: logger
postgres      29       7  0 21:27 ?        00:00:00 postgres: checkpointer
postgres      30       7  0 21:27 ?        00:00:00 postgres: background writer
postgres      32       7  0 21:27 ?        00:00:00 postgres: walwriter
postgres      33       7  0 21:27 ?        00:00:00 postgres: autovacuum launcher
postgres      34       7  0 21:27 ?        00:00:00 postgres: logical replication launcher
root         266       0 37 21:29 pts/0    00:00:00 /usr/bin/ps -ef
```

Номера процессов не поменялись, экземпляр не перезапускался.

22) В диагностическом журнале записей о перезапуске процессов нет:

```
root@tantor:~# cat ~/data/log/postgresql-<TAB>_212736.log
21:27:36.139 UTC [7] LOG:  starting PostgreSQL 17.2 (Debian 17.2-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled
by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
21:27:36.139 UTC [7] LOG:  listening on IPv4 address "0.0.0.0", port 5432
21:27:36.139 UTC [7] LOG:  listening on IPv6 address "::", port 5432
21:27:36.161 UTC [7] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
21:27:36.183 UTC [31] LOG:  database system was interrupted; last known up at 21:24:08 UTC
21:27:38.136 UTC [31] LOG:  database system was not properly shut down; automatic recovery in progress
21:27:38.144 UTC [31] LOG:  redo starts at 0/177D810
21:27:38.144 UTC [31] LOG:  invalid record length at 0/177D848: expected at least 24, got 0
21:27:38.144 UTC [31] LOG:  redo done at 0/177D810 system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed:
0.00 s
21:27:38.165 UTC [29] LOG:  checkpoint starting: end-of-recovery immediate wait
21:27:38.246 UTC [29] LOG:  checkpoint complete: wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0
recycled; write=0.034 s, sync=0.008 s, total=0.094 s; sync files=2, longest=0.004 s, average=0.004 s; distance=0
kB, estimate=0 kB; lsn=0/177D848, redo lsn=0/177D848
21:27:38.257 UTC [7] LOG:  database system is ready to accept connections
```

Экземпляр не рестартовал.

23) В целях проверки производительности PostgreSQL, работающего в контейнере docker выполните тест из 7 части предыдущей практики:

```
root@tantor:~# sudo docker exec -it postgres bash
root@328a75c4c4c4:/# export PGUSER=postgres
root@328a75c4c4c4:/# psql -c "drop table if exists tt1;"
psql -c "create table tt1 (id bigint generated by default as identity (cache 1)
primary key, data bigint);"
echo "insert into tt1(data) values(1);" >txn.sql
psql -c "vacuum analyze tt1;"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
psql -c "drop table if exists tt1;"
psql -c "create table tt1 (id uuid default gen_random_uuid() primary key, data
bigint);"
psql -c "vacuum analyze tt1;"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
NOTICE:  table "tt1" does not exist, skipping
DROP TABLE
CREATE TABLE
VACUUM
tps = 551.864768 (without initial connection time)
DROP TABLE
CREATE TABLE
VACUUM
tps = 542.598808 (without initial connection time)
root@328a75c4c4c4:/# exit
```

```
exit
```

Для первичного ключа типа biginteger `tps = 551`, для uuid `tps = 542`.
В тесте 7 части предыдущей практики результаты были: biginteger `tps = 556`, для uuid `tps = 553`.
Результаты теста схожи.

24) Остановите контейнер, чтобы освободить память:

```
root@tantor:~# docker stop postgres
```

25) Директория PGDATA кластера внутри контейнера была смаппирована на директорию хоста `/root/data` и после остановки контейнера доступна:

```
root@tantor:~# ls -al /root/data
total 136
drwx------ 19 dnsmasq root             4096 13:17 .
drwxr-x--- 18 root    root             4096 12:54 ..
drwx------  5 dnsmasq systemd-journal  4096 12:31 base
drwx------  2 dnsmasq systemd-journal  4096 12:44 global
drwx------  2 dnsmasq systemd-journal  4096 12:31 pg_commit_ts
drwx------  2 dnsmasq systemd-journal  4096 12:31 pg_dynshmem
-rw-------  1 dnsmasq systemd-journal  5839 12:31 pg_hba.conf
-rw-------  1 dnsmasq systemd-journal  2640 12:31 pg_ident.conf
drwx------  4 dnsmasq systemd-journal  4096 13:22 pg_logical
drwx------  4 dnsmasq systemd-journal  4096 12:31 pg_multixact
drwx------  2 dnsmasq systemd-journal  4096 12:31 pg_notify
drwx------  2 dnsmasq systemd-journal  4096 12:31 pg_replslot
drwx------  2 dnsmasq systemd-journal  4096 12:31 pg_serial
drwx------  2 dnsmasq systemd-journal  4096 12:31 pg_snapshots
drwx------  2 dnsmasq systemd-journal  4096 13:17 pg_stat
drwx------  2 dnsmasq systemd-journal  4096 12:31 pg_stat_tmp
drwx------  2 dnsmasq systemd-journal  4096 12:31 pg_subtrans
drwx------  2 dnsmasq systemd-journal  4096 12:31 pg_tblspc
drwx------  2 dnsmasq systemd-journal  4096 12:31 pg_twophase
-rw-------  1 dnsmasq systemd-journal     3 12:31 PG_VERSION
drwx------  4 dnsmasq systemd-journal  4096 12:31 pg_wal
drwx------  2 dnsmasq systemd-journal  4096 12:31 pg_xact
-rw-------  1 dnsmasq systemd-journal    88 12:31 postgresql.auto.conf
-rw-------  1 dnsmasq systemd-journal 30777 12:31 postgresql.conf
-rw-------  1 dnsmasq systemd-journal    36 13:17 postmaster.opts
-rw-------  1 dnsmasq systemd-journal    94 13:17 postmaster.pid
```

На хосте можно редактировать файлы конфигурации и копировать директорию в целях резервирования.
Если удалить контейнер, то содержимое директории не будет удалено. При создании контейнера с параметрами **`PGDATA=/var/lib/postgresql/data -d -v /root/data:/var/lib/postgresql/data`** утилита initdb не пересоздаст кластер и экземпляр запустится используя содержимое директории. Это удобно для копирование PGDATA с хоста в контейнер.

26) Если контейнер не работоспособен или не нужен, то можно удалить контейнер и директорию:

```
root@tantor:~# docker rm -f postgres
postgres
root@tantor:~# rm -rf /root/data
root@tantor:~# exit
logout
astra@tantor:~$ exit
```

Если директорию не удалять, то она будет использоваться вместе с содержимым при создании нового контейнера.
При желании контейнер можно будут созать снова. Простота создания контейнеров - основное преимущество docker.

24) Посмотрите не было ли предупреждени и ошибок, относящихся к docker:

```
root@tantor:~# dmesg | tail -20
[  457.152664] vethd2310dc (unregistering): left allmulticast mode
[  457.152669] vethd2310dc (unregistering): left promiscuous mode
[  457.152693] br-37b8e28c6b2f: port 1(vethd2310dc) entered disabled state
[  473.723967] overlayfs: fs on
'/var/lib/docker/overlay2/3b717e6fb2d2e00fd7463b8c3a586f92dfffb26dd6e993b402cbddfe4214f68e/merged' does not
support file handles, falling back to xino=off.
[  475.027287] br-1fc82bacd93a: port 1(veth63f8123) entered blocking state
[  475.027293] br-1fc82bacd93a: port 1(veth63f8123) entered disabled state
[  475.027308] veth63f8123: entered allmulticast mode
[  475.027360] veth63f8123: entered promiscuous mode
[  475.334287] eth0: renamed from veth68a4798
[  475.354710] br-1fc82bacd93a: port 1(veth63f8123) entered blocking state
[  475.354726] br-1fc82bacd93a: port 1(veth63f8123) entered forwarding state
[  475.368511] WARNING: chroot access!
[  708.878087] veth68a4798: renamed from eth0
[  708.902616] br-1fc82bacd93a: port 1(veth63f8123) entered disabled state
[  708.948680] br-1fc82bacd93a: port 1(veth63f8123) entered disabled state
[  708.949310] veth63f8123 (unregistering): left allmulticast mode
[  708.949315] veth63f8123 (unregistering): left promiscuous mode
[  708.949340] br-1fc82bacd93a: port 1(veth63f8123) entered disabled state
```

Параметры запуска, относящиеся к overlay можно посмотреть командой:

```
root@tantor:~# cat /boot/config-6.6.28-1-generic | grep OVERLAY
CONFIG_EFI_CUSTOM_SSDT_OVERLAYS=y
CONFIG_OVERLAY_FS=m
# CONFIG_OVERLAY_FS_REDIRECT_DIR is not set
CONFIG_OVERLAY_FS_REDIRECT_ALWAYS_FOLLOW=y
# CONFIG_OVERLAY_FS_INDEX is not set
CONFIG_OVERLAY_FS_XINO_AUTO=y
# CONFIG_OVERLAY_FS_METACOPY is not set
# CONFIG_OVERLAY_FS_DEBUG is not set
```

Причина предупреждений может быть в том, что параметры контейнера отличаются от параметров операционной системы, в которой запущен контейнер.

27) Этот пункт не нужно выполнять, только посмотреть пример.

Следующая команда показывает список имеющихся образов:

```
root@tantor:~# docker images
REPOSITORY                  TAG       IMAGE ID       CREATED         SIZE
postgres                    latest    de869b203456   2 days ago      494MB
postgres                    17        4bc6cc20ca7a   2 months ago    435MB
```

В примере два образа. Образ postgres с двумя разными версиями PostgreSQL 17.1 и последней.

Можно удалить ненужные образы, чтобы не занимали место. В команде можно указать IMAGE ID или REPOSITORY. Пример:

```
root@tantor:~# docker image rm postgres
Untagged: postgres:latest
Untagged: postgres@sha256:6e3358e46e34dae6c184f48fd06fe1b3dbf958ad5b83480031907e52b9ec2a7d
Deleted: sha256:de869b20345625dc5430fbe0f15e490dad3921823915b5524ce1436ec8378793
...
Deleted: sha256:7914c8f600f532b7adbd0b003888e3aa921687d62dbe2f1f829d0ab6234a158a
```

Список образов после удаления:

```
root@tantor:~# docker images
REPOSITORY                  TAG       IMAGE ID       CREATED         SIZE
postgres                    17        4bc6cc20ca7a   2 months ago    435MB
```

Один из образов не был удалён. Попробуем использовать `IMAGE ID`:

```
root@tantor:~# docker image rm 4bc6cc20ca7a
Error response from daemon: conflict: unable to delete 4bc6cc20ca7a (must be
forced) - image is being used by stopped container 3116d7318306
```

Образ может быть удалён с параметром `-f` (forced). Образ используется остановленным контейнером, который не выдаётся в списке:

```
root@tantor:~# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

Удалим несуществующий контейнер, используя его `CONTAINER ID`:

```
root@tantor:~# docker rm -f 3116d7318306
3116d7318306
```

Теперь удаление образа пройдёт успешно:

```
root@tantor:~# docker image rm 4bc6cc20ca7a
Untagged: postgres:17
Untagged:
postgres@sha256:888402a8cd6075c5dc83a31f58287f13306c318eaad016661ed12e076f3e6341
...
Deleted: sha256:8b296f48696071aafb5a6286ca60d441a7e559b192fc7f94bb63ee93dae98f17
```

Освобождение места:

```
root@tantor:~# docker image prune -f
```

Список образов пуст:

```
root@tantor:~# docker images -all
REPOSITORY   TAG       IMAGE ID   CREATED   SIZE
```

Часть 2. Разделяемая память экземпляра

1) Посмотрите список структур разделяемой памяти экземпляра:

```
postgres=# select * from (select *, lead(off) over(order by off) - off as
true_size from pg_shmem_allocations) as a order by 1;
```

name	off	size	allocated_size	true_size
<anonymous>		**4984960**	**4984960**	
Archiver Data	147726208	8	128	128
Async Queue Control	147729280	3984	4096	4096
AutoVacuum Data	147713792	5328	5376	5376
Backend Activity Buffer	146921600	131072	131072	131072
Backend Application Name Buffer	146905216	8192	8192	8192
Backend Client Host Name Buffer	146913408	8192	8192	8192
Backend GSS Status Buffer	147094144	8576	8576	8576
Backend SSL Status Buffer	147052672	41472	41472	41472
Backend Status Array	146845824	59392	59392	59392
Background Worker Data	147102848	4496	4608	4608
BTree Vacuum State	147726976	1476	1536	1536
Buffer Blocks	6894592	134221824	134221824	134221824
Buffer Descriptors	5846016	1048576	1048576	1048576
Buffer IO Condition Variables	141116416	262144	262144	262144
Buffer Strategy Status	142667648	28	128	128
Checkpoint BufferIds	141378560	327680	327680	327680
Checkpointer Data	147189376	524352	524416	524416
commit_timestamp	4907520	267424	267520	267520
CommitTs shared	5175040	32	128	128
Control File	4377088	328	384	384
Fast Path Strong Relation Lock Data	144058112	4100	4224	4224
FinishedSerializableTransactions	146353792	16	128	128

```
 KnownAssignedXids                   | 146774400 |      63440 |          63488 |      63488
 KnownAssignedXidsValid              | 146837888 |       7930 |           7936 |       7936
 LOCK hash                           | 142667776 |       2896 |           2944 |     695168
 Logical Replication Launcher Data   | 147726336 |        496 |            512 |        512
 multixact_member                    |   5576448 |     267424 |         267520 |     267520
 multixact_offset                    |   5442688 |     133760 |         133760 |     133760
 notify                              | 147733376 |     133760 |         133760 |     133760
 OldSnapshotControlData              | 147726848 |         88 |            128 |        128
 PMSignalState                       | 147176960 |       1016 |           1024 |       1024
 PREDICATELOCK hash                  | 144507648 |       2896 |           2944 |    1260416
 PREDICATELOCKTARGET hash            | 144062336 |       2896 |           2944 |     445312
 PredXactList                        | 145768064 |         88 |            128 |     234368
 Prepared Transaction Table          | 147102720 |         16 |            128 |        128
 Proc Array                          | 146773760 |        544 |            640 |        640
 Proc Header                         | 146621568 |        136 |            256 |     152192
 PROCLOCK hash                       | 143362944 |       2896 |           2944 |     695168
 ProcSignal                          | 147177984 |      11272 |          11392 |      11392
 ReplicationOriginState              | 147722112 |        568 |            640 |        640
 ReplicationSlot Ctl                 | 147719168 |       2880 |           2944 |       2944
 RWConflictPool                      | 146060800 |         24 |            128 |     292992
 SerialControlData                   | 146621440 |         24 |            128 |        128
 serializable                        | 146353920 |     267424 |         267520 |     267520
 SERIALIZABLEXID hash                | 146002432 |       2896 |           2944 |      58368
 Shared Buffer Lookup Table          | 141706240 |       2896 |           2944 |     961408
 Shared Memory Stats                 | 147867136 |     277880 |         277888 |     316800
 Shared MultiXact State              |   5843968 |       2008 |           2048 |       2048
 shmInvalBuffer                      | 147107456 |      69464 |          69504 |      69504
 subtransaction                      |   5175168 |     267424 |         267520 |     267520
 Sync Scan Locations List            | 147728512 |        656 |            768 |        768
 Tantor Control File                 |   4377472 |         40 |            128 |        128
 transaction                         |   4377856 |     529568 |         529664 |     529664
 Wal Receiver Ctl                    | 147723904 |       2264 |           2304 |       2304
 Wal Sender Ctl                      | 147722752 |       1144 |           1152 |       1152
 XLOG Ctl                            |    168832 |    4208232 |        4208256 |    4208256
 XLogPrefetchStats                   |   4377600 |         72 |            128 |        128
 XLOG Recovery Ctl                   |   4377728 |        104 |            128 |        128
                                     | 148183936 |    1885312 |        1885312 |
(60 rows)
```

Значения в байтах.

Названия структур ни очем не говорят и в документации отсутствуют. В книгах и учебниках разделяемая память рассматривается на примере нескольких структур: кэша буферов, журнального буфера, буфера статусов транзакций CLOG (Xact). В столбце `name` представления **`pg_shmem_allocations`** эти структуры нелегко найти.

`Buffer Blocks` – кэш буферов, размер задается параметром shared_buffers

`Buffer Descriptors` - заголовки буферов, в которых хранятся признаки закрепления pin count, число обращений к буферу usage count и другие биты. Размер зависит от числа буферов.

`Shared Buffer Lookup Table` - (Buffer Mapping Table) хэш-таблица для поиска блоков в буферном кэше.

`Buffer Strategy Status` - (структура BufferStrategyControl) данные о свободных блоках в буферном кэше (первый свободный буфер firstFreeBuffer, последний свободный буфер lastFreeBuffer, указатель на блок для алгоритма clock sweep и другие значения)

`XLOG Ctl` - WAL буфер, буфер журнала, размер задается параметром wal_buffers.

`transaction` буфер, размер которого задается параметром transaction_buffers.

`subtransaction` буфер, размер которого задается параметром subtransaction_buffers.

`serializable` буфер, размер которого задается параметром serializable_buffers.

`notify` буфер, размер которого задается параметром notify_buffers.

`multixact_member` буфер, размер которого задается параметром multixact_member_buffers.

`multixact_offset` буфер, размер которого задается параметром multixact_offset_buffers.

`commit_timestamp` буфер, размер которого задается параметром commit_timestamp_buffers.

`shmInvalBuffer` - кольцевой буфер на 4096 сообщений об изменениях в таблицах системного каталога.

Размер `KnownAssignedXids` и `KnownAssignedXidsValid` (в 8 раз меньше `KnownAssignedXids`) пропорционально сумме значений параметров max_connections + максимальное число фоновых процессов экземпляра +max_prepared_transactions. Структуры выделяются только, если параметр `hot_standby=on`, в том числе на мастере. Используются репликой в режиме восстановления.

2) Список структур разделяемой памяти для которых данные о выделенной памяти расходятся с данными size и allocated_size:

```
postgres=# select * from (select *, lead(off) over(order by off) - off as
true_size from pg_shmem_allocations) as a where a.true_size<>a.allocated_size
order by 1;
            name             |    off    |  size  | allocated_size | true_size
-----------------------------+-----------+--------+----------------+-----------
 LOCK hash                   | 142667776 |   2896 |           2944 |    695168
 PREDICATELOCK hash          | 144507648 |   2896 |           2944 |   1260416
 PREDICATELOCKTARGET hash    | 144062336 |   2896 |           2944 |    445312
 PredXactList                | 145768064 |     88 |            128 |    234368
 Proc Header                 | 146621568 |    136 |            256 |    152192
 PROCLOCK hash               | 143362944 |   2896 |           2944 |    695168
 RWConflictPool              | 146060800 |     24 |            128 |    292992
 SERIALIZABLEXID hash        | 146002432 |   2896 |           2944 |     58368
 Shared Buffer Lookup Table  | 141706240 |   2896 |           2944 |    961408
 Shared Memory Stats         | 147867136 | 277880 |         277888 |    316800
(10 rows)
```

3) Статистика использования SLRU-кэшей:

```
postgres=# select name, blks_hit, blks_read, blks_written, blks_exists, flushes,
truncates from pg_stat_slru;
       name        | blks_hit | blks_read | blks_written | blks_exists | flushes | truncates
-------------------+----------+-----------+--------------+-------------+---------+-----------
 commit_timestamp  |        0 |         0 |            0 |           0 |     253 |         0
 multixact_member  |        0 |         0 |            0 |           0 |     253 |         0
 multixact_offset  |        0 |         6 |            5 |           0 |     253 |         0
 notify            |        0 |         0 |            0 |           0 |       0 |         0
 serializable      |        0 |         0 |            0 |           0 |       0 |         0
 subtransaction    |        0 |         0 |          210 |           0 |     253 |       252
 transaction       |   409116 |         7 |          221 |           0 |     253 |         0
 other             |        0 |         0 |            0 |           0 |       0 |         0
(8 rows)
```

Статистика сохраняется. В представлении есть столбец `stats_reset` с времени последнего сброса статистики. Сброс статистики выполняется функцией: select pg_stat_reset_slru(null);
В качестве аргумента можно передать название кэша или NULL, если нужно сбросить статистику для всех кэшей.
Статистику из представления можно использовать для установки параметров конфигурации, задающих размеры SLRU-кэшей:

```
postgres=# \dconfig *_buffers
         Parameter          | Value
----------------------------+-------
 commit_timestamp_buffers   | 256kB
 multixact_member_buffers   | 256kB
 multixact_offset_buffers   | 128kB
 notify_buffers             | 128kB
 serializable_buffers       | 256kB
 shared_buffers             | 128MB
 subtransaction_buffers     | 256kB
 temp_buffers               | 8MB
 transaction_buffers        | 256kB
 wal_buffers                | 4MB
```

Дальше посмотрим как определить на какие разделяемые структуры памяти экземпляра влияют изменения параметров конфигурации. Часть параметров конфигурации влияет на большое число структур памяти. Размер изменения тоже важен, так как к части структур процессы обращаются очень часто и такие структуры (или их части, к которым идут частые обращения) будут находиться (или вытесняться и каждый раз загружаться) находиться в кэшах процессоров, а ссылки на их страницы в TLB. Размеры кэшей процессоров не очень большие: от

килобайт до мегабайт и даже небольшое увеличение некоторых структур (например, `Proc Header` и `Proc Array` со 100Кб до 200Кб) может привести к резкому увеличению числа вытеснений и загрузки в кэши.

Второй момент: для доступа (прочесть, поменять содержимое) к частям структур в разделяемой памяти (или всей структуре если она небольшого размера) процесс всегда получает блокировку: либо SpinLock, либо LWLock. Если число LWLock, которыми защищена структура не меняется, то при увеличении размера структуры (или её части защищенной одной блокировкой) процесс будет читать/менять содержимое структуры дольше, что приведёт к увеличению длительности удержания LWLock. Это может привести к появлению или увеличению числа столкновений процессов при получении LWLock. LWLock не рассчитаны на долгое время удержания и на долгое ожидание получения блокировки.

При увеличении числа LWLock можно определить доступ к какой структуру памяти защищает этот тип блокировки и проверить не увеличивалась ли эта структура каким-то параметром конфигурации. Возможно, уменьшение структуры памяти уберёт конкуренцию за LWLock.

В некоторых форках параметрами может увеличиваться число LWLock, которыми защищаются части структуры памяти (число "траншей"), но сам LWLock также находится в структуре памяти (обычно защищается SpinLock) и размер такой структуры также увеличивается. Узким местом может стать доступ к такой структуре.

5) Создайте представления для удобства написания запросов и таблицу с исходными (с которыми будут сравниваться) размерами структур разделяемой памяти:

```
postgres=# drop table if exists shmem_reference cascade;
create table shmem_reference as select coalesce(name, name, '<NULL>') name,
size, allocated_size, coalesce(true_size, true_size, allocated_size) true_size
from (select *, lead(off) over(order by off) - off as true_size from
pg_shmem_allocations) as a order by 1;
create or replace view shmem_v as select coalesce(name, name, '<NULL>') name,
size, allocated_size, coalesce(true_size, true_size, allocated_size) true_size
from (select *, lead(off) over(order by off) - off as true_size from
pg_shmem_allocations) as a order by 1;
create or replace view shmem_compare as select v.name, v.allocated_size
allocated, r.allocated_size ref_alloc, v.true_size true_size, r.true_size
ref_true, v.size, r.size ref_size from shmem_reference r full outer join shmem_v
v on (r.name=v.name) where r.true_size<>v.true_size order by v.name;
\q
NOTICE:  table "shmem_reference" does not exist, skipping
DROP TABLE
SELECT 60
CREATE VIEW
CREATE VIEW
postgres@tantor:~$
```

6) Увеличьте размер буферного кэша до 130Мб, рестартуйте экземпляр и посмотрите размеры каких структур памяти изменились:

```
postgres@tantor:~$ psql -c "alter system set shared_buffers='130MB'"
sudo restart
psql -c "select * from shmem_compare"
ALTER SYSTEM
             name              | allocated | ref_alloc | true_size | ref_true  |   size    | ref_size
-------------------------------+-----------+-----------+-----------+-----------+-----------+----------
 <anonymous>                   |   4958336 |   4946048 |   4958336 |   4946048 |   4958336 |   4946048
 Buffer Blocks                 | 136318976 | 134221824 | 136318976 | 134221824 | 136318976 | 134221824
 Buffer Descriptors            |   1064960 |   1048576 |   1064960 |   1048576 |   1064960 |   1048576
 Buffer IO Condition Variables |    266240 |    262144 |    266240 |    262144 |    266240 |    262144
 Checkpoint BufferIds          |    332800 |    327680 |    332800 |    327680 |    332800 |    327680
 Checkpointer Data             |    532608 |    524416 |    532608 |    524416 |    532544 |    524352
 <NULL>                        |   1918976 |   1924224 |   1918976 |   1924224 |   1918976 |   1924224
 Shared Buffer Lookup Table    |      2944 |      2944 |    973696 |    961408 |      2896 |      2896
 XLOG Ctl                      |   4273920 |   4208256 |   4273920 |   4208256 |   4273832 |   4208232
(9 rows)
```

Изменение повлияло на 9 структур разделяемой памяти. В том числе структур, способ выделения памяти под которые не позволяет назначить им имя и они выводятся как `<anonymous>` и `<NULL>` (пусто). Под этими именами учитывается не одна, а несколько структур. `<NULL>` - выделенная, но неиспользуемая `<anonymous>` память.

Размер журнального буфера `XLOG Ctl` увеличился до 1/32 **`shared_buffers`**: **`136318976`**/**`4273920`**=`31,8955`. Почему число не круглое 32? Потому, что в структуре `Buffer Blocks` хранятся не только буфера, число которых устанавливается параметром **`shared_buffers`**.

7) Увеличьте размер буферного кэша до 1024Мб, рестартуйте экземпляр и посмотрите размеры каких структур памяти изменились:

```
postgres@tantor:~$ psql -c "alter system set shared_buffers='1024MB'"
sudo restart
psql -c "select * from shmem_compare"
psql -c "alter system reset shared_buffers"
ALTER SYSTEM
             name               | allocated  | ref_alloc | true_size  | ref_true  |    size    | ref_size
--------------------------------+------------+-----------+------------+-----------+------------+----------
 <anonymous>                    |   11368576 |   4946048 |   11368576 |   4946048 |   11368576 |   4946048
 Buffer Blocks                  | 1073745920 | 134221824 | 1073745920 | 134221824 | 1073745920 | 134221824
 Buffer Descriptors             |    8388608 |   1048576 |    8388608 |   1048576 |    8388608 |   1048576
 Buffer IO Condition Variables  |    2097152 |    262144 |    2097152 |    262144 |    2097152 |    262144
 Checkpoint BufferIds           |    2621440 |    327680 |    2621440 |    327680 |    2621440 |    327680
 Checkpointer Data              |    4194432 |    524416 |    4194432 |    524416 |    4194368 |    524352
 commit_timestamp               |    2138624 |    267520 |    2138624 |    267520 |    2138560 |    267424
 <NULL>                         |    1920384 |   1924224 |    1920384 |   1924224 |    1920384 |   1924224
 Shared Buffer Lookup Table     |       9088 |      2944 |    7390080 |    961408 |       9040 |      2896
 subtransaction                 |    2138624 |    267520 |    2138624 |    267520 |    2138560 |    267424
 transaction                    |    4235776 |    529664 |    4235776 |    529664 |    4235712 |    529568
 XLOG Ctl                       |   16803456 |   4208256 |   16803456 |   4208256 |   16803432 |   4208232
(12 rows)
ALTER SYSTEM
```

Большее изменение повлияло ещё на **три** структуры. Это может быть из-за выравнивания, если структура небольшого размера или размер структуры рассчитывается по формуле, в которой присутствует число буферов (NBuffers).

Размер журнального буфера `XLOG Ctl` увеличился до 16Мб, так как по умолчанию он устанавливается либо в размер WAL-сегмента 16Мб, либо 1/32 **`shared_buffers`**. Сейчас `XLOG Ctl` был установлен в размер WAL-сегмента.

Под каким `name` "спрятали" буфер CLOG, который часто приводят как пример структуры разделяемой памяти? Размер этого SLRU-буфера рассчитывается по формуле: `Min(128, Max(4, NBuffers/512))`. Для 128Мб 262144 байт, 1Гб - 1028576 байт. CLOG может скрываться под любой из строк.

Попробуем выявить CLOG другим способом. Под CLOG память выделяется функцией `CLOGShmemInit()`, которая вызывается в коде `ipci.c`. Выделение разделяемой памяти идёт в следующей последовательности:

```
/*
 * Set up xlog, clog, and buffers
 */
XLOGShmemInit();
XLogPrefetchShmemInit();
XLogRecoveryShmemInit();
CLOGShmemInit();
CommitTsShmemInit();
SUBTRANSShmemInit();
MultiXactShmemInit();
InitBufferPool();
/*
 * Set up lock manager
 */
InitLocks();
```

8) Выведите структуры в порядке, в котором операционная система выделила память:

```
postgres=# select * from (select *, lead(off) over(order by off) - off as
true_size from pg_shmem_allocations) as a order by off limit 20;
            name               |    off     |    size    | allocated_size | true_size
-------------------------------+------------+------------+----------------+-----------
 XLOG Ctl                      |     168832 |   16803432 |       16803456 |   16803456
 Control File                  |   16972288 |        328 |            384 |        384
 Tantor Control File           |   16972672 |         40 |            128 |        128
 XLogPrefetchStats             |   16972800 |         72 |            128 |        128
 XLOG Recovery Ctl             |   16972928 |        104 |            128 |        128
 transaction                   |   16973056 |    4235712 |        4235776 |    4235776
 commit_timestamp              |   21208832 |    2138560 |        2138624 |    2138624
 CommitTs shared               |   23347456 |         32 |            128 |        128
 subtransaction                |   23347584 |    2138560 |        2138624 |    2138624
 multixact_offset              |   25486208 |     133760 |         133760 |     133760
 multixact_member              |   25619968 |     267424 |         267520 |     267520
 Shared MultiXact State        |   25887488 |       2008 |           2048 |       2048
 Buffer Descriptors            |   25889536 |    8388608 |        8388608 |    8388608
 Buffer Blocks                 |   34278144 | 1073745920 |     1073745920 | 1073745920
 Buffer IO Condition Variables | 1108024064 |    2097152 |        2097152 |    2097152
 Checkpoint BufferIds          | 1110121216 |    2621440 |        2621440 |    2621440
 Shared Buffer Lookup Table    | 1112742656 |       9040 |           9088 |    7390080
 Buffer Strategy Status        | 1120132736 |         28 |            128 |        128
 LOCK hash                     | 1120132864 |       2896 |           2944 |     695168
 PROCLOCK hash                 | 1120828032 |       2896 |           2944 |     695168
(20 rows)
```

CLOG буфер в представлении имеет имя `transaction`.

Размер буфера **`transaction`** задается параметром конфигурации `transaction_buffers`.

Описание параметра в документации https://docs.tantorlabs.ru/tdb/ru/16_6/se/runtime-config-resource.html:

количество общей памяти, используемой для кэширования содержимого директории `PGDATA/`**`pg_xact`**. Если это значение указано без единиц измерения, оно принимается за блоки (BLCKSZ=8kB). Значение по умолчанию - 0 и размер будет установлен от **`shared_buffers/512`** до 1024 блоков, но не менее 16 блоков.

Мы посмотрели, что столбец **`off`** представления **`pg_shmem_allocations`** может использоваться не только для выявления "дыр" ("holes") в виртуальном адресном пространстве, но и **для идентификации структур памяти**.

9) Посмотрите как влияет на структуры памяти изменение параметра **`max_connections`**:

```
postgres@tantor:~$ psql -c "alter system reset shared_buffers"
psql -c "alter system set max_connections=200"
sudo restart
psql -c "select * from shmem_compare"
psql -c "select sum(true_size) - sum(ref_size) delta_true from shmem_compare"
psql -c "alter system reset max_connections"
ALTER SYSTEM
ALTER SYSTEM
              name               | allocated | ref_alloc | true_size | ref_true |  size   | ref_size
---------------------------------+-----------+-----------+-----------+----------+---------+----------
 <anonymous>                     |   8131200 |   4946048 |   8131200 |  4946048 | 8131200 |  4946048
 Async Queue Control             |      7296 |      4096 |      7296 |     4096 |    7184 |     3984
 Backend Activity Buffer         |    233472 |    131072 |    233472 |   131072 |  233472 |   131072
 Backend Application Name Buffer |     14592 |      8192 |     14592 |     8192 |   14592 |     8192
 Backend Client Host Name Buffer |     14592 |      8192 |     14592 |     8192 |   14592 |     8192
 Backend GSS Status Buffer       |     15360 |      8576 |     15360 |     8576 |   15276 |     8576
 Backend SSL Status Buffer       |     73984 |     41472 |     73984 |    41472 |   73872 |    41472
 Backend Status Array            |    105856 |     59392 |    105856 |    59392 |  105792 |    59392
 BTree Vacuum State              |      2688 |      1536 |      2688 |     1536 |    2676 |     1476
 KnownAssignedXids               |    115456 |     63488 |    115456 |    63488 |  115440 |    63440
 KnownAssignedXidsValid          |     14464 |      7936 |     14464 |     7936 |   14430 |     7930
 LOCK hash                       |      2944 |      2944 |   1264512 |   695168 |    2896 |     2896
 <NULL>                          |   3441152 |   1924224 |   3441152 |  1924224 | 3441152 |  1924224
 PMSignalState                   |      1920 |      1024 |      1920 |     1024 |    1816 |     1016
 PREDICATELOCK hash              |      2944 |      2944 |   2313088 |  1260416 |    2896 |     2896
 PREDICATELOCKTARGET hash        |      2944 |      2944 |    818048 |   445312 |    2896 |     2896
 PredXactList                    |       128 |       128 |    426368 |   234368 |      88 |       88
 Proc Array                      |      1024 |       640 |      1024 |      640 |     944 |      544
 Proc Header                     |       256 |       256 |    271872 |   152192 |     136 |      136
 PROCLOCK hash                   |      2944 |      2944 |   1272704 |   695168 |    2896 |     2896
 ProcSignal                      |     20096 |     11392 |     20096 |    11392 |   20072 |    11272
```

```
 RWConflictPool                |       128 |       128 |     532992 |   292992 |       24 |       24
 SERIALIZABLEXID hash          |      2944 |      2944 |     106752 |    58368 |     2896 |     2896
 Shared MultiXact State        |      3712 |      2048 |       3712 |     2048 |     3608 |     2008
 shmInvalBuffer                |     72704 |     69504 |      72704 |    69504 |    72664 |    69464
(25 rows)
 delta_true
------------
   11972874
(1 row)
ALTER SYSTEM
```

Изменение значения параметра **max_connections** со 100 до 200 повлияло на большое число структур памяти. Было выделено дополнительно **11972874** байт.

10) Посмотрите как влияет на структуры памяти изменение параметра **max_connections**, который наравне с **max_locks_per_transaction** ограничивает общее число блокировок объектов на всём экземпляре:

```
postgres@tantor:~$ psql -c "alter system set max_locks_per_transaction=128"
sudo restart
psql -c "select * from shmem_compare"
psql -c "select sum(true_size) - sum(ref_size) delta_true from shmem_compare"
psql -c "alter system reset max_locks_per_transaction"
ALTER SYSTEM
     name      | allocated | ref_alloc | true_size | ref_true |  size   | ref_size
---------------+-----------+-----------+-----------+----------+---------+----------
 <anonymous>   |   6322304 |   4946048 |   6322304 |  4946048 | 6322304 |  4946048
 LOCK hash     |      2944 |      2944 |   1383296 |   695168 |    2896 |     2896
 <NULL>        |   3579008 |   1924224 |   3579008 |  1924224 | 3579008 |  1924224
 PROCLOCK hash |      2944 |      2944 |   1383296 |   695168 |    2896 |     2896
(4 rows)
 delta_true
------------
    5791840
(1 row)
ALTER SYSTEM
```

Число блокировок объектов и рекомендательных блокировок (advisory locks) ограничено на экземпляре произведением **max_locks_per_transaction** * (**max_connections** + max_prepared_transactions). Если нужно увеличить число блокировок на экземпляре, то увеличивают один из этих параметров или сразу оба параметра.

Изменение значения параметра **max_locks_per_transaction** с 64 до 128 повлияло 3 структуры памяти и неиспользуемую (<NULL>) область памяти. Было выделено дополнительно **5791840** байт, что в 2 раза меньше, чем при изменении **max_connections**.

Однако, **нельзя сказать** увеличение какого параметра больше снизит производительность, так как неизвестно размер какой структуры больше влияет на производительность и насколько. Если к структуре памяти обращения идут сильно чаще, то увеличение размера этой структуры сильнее снижает производительность. Структура тяжелых блокировок находится в <anonymous>. Для доступа к структуре тяжелых блокировок используется LOCK hash. Обычно доступ к структуре PROCLOCK hash идёт чаще (в ~"число процессов" раз), чем к структуре тяжелых блокировок.

Запросами, приведенными в этой части практики можно получить как повлияет изменение параметра конфигурации на размер структур памяти и оценить какие блокировки, связанные со своими структурами памяти будут появляться чаще.

Часть 3. Локальная память серверного процесса

1) Выполните запросы в новой сессии psql:

```
postgres=# select sum(total_bytes) total, sum(total_nblocks) blocks, sum(used_bytes)
used, sum(free_bytes) free, sum(free_chunks) f_chunks from pg_backend_memory_contexts;
 with recursive dep as
 (select name, total_bytes as total, ident, parent, 1 as level, name as path from
pg_backend_memory_contexts where parent is null
 union all
```

```
 select c.name, c.total_bytes, c.ident, c.parent, p.level + 1, p.path || '->' || c.name
 from dep p, pg_backend_memory_contexts c
 where c.parent = p.name)
 select name, total, parent, level l, path  from dep where parent<>'CacheMemoryContext';
 select sum(total_bytes) total, sum(total_nblocks) blocks, sum(used_bytes) used,
sum(free_bytes) free, sum(free_chunks) f_chunks from pg_backend_memory_contexts;
```

```
  total   | blocks |  used  |  free  | f_chunks
---------+--------+--------+--------+----------
 1321128 |    196 | 965664 | 355464 |      130
(1 row)
```

```
          name           | total  |      parent       | l |                  path
-------------------------+--------+-------------------+---+-----------------------------------------
Type information cache   |  24368 | TopMemoryContext  | 2 | TopMemoryContext->Type information cache
Operator lookup cache    |  24576 | TopMemoryContext  | 2 | TopMemoryContext->Operator lookup cache
TopTransactionContext    |   8192 | TopMemoryContext  | 2 | TopMemoryContext->TopTransactionContext
Record information cache |   8192 | TopMemoryContext  | 2 | TopMemoryContext->Record information cache
RowDescriptionContext    |   8192 | TopMemoryContext  | 2 | TopMemoryContext->RowDescriptionContext
MessageContext           | 262144 | TopMemoryContext  | 2 | TopMemoryContext->MessageContext
Operator class cache     |   8192 | TopMemoryContext  | 2 | TopMemoryContext->Operator class cache
smgr relation table      |  32768 | TopMemoryContext  | 2 | TopMemoryContext->smgr relation table
PgStat Shared Ref Hash   |   7216 | TopMemoryContext  | 2 | TopMemoryContext->PgStat Shared Ref Hash
PgStat Shared Ref        |   4096 | TopMemoryContext  | 2 | TopMemoryContext->PgStat Shared Ref
PgStat Pending           |   8192 | TopMemoryContext  | 2 | TopMemoryContext->PgStat Pending
TransactionAbortContext  |  32768 | TopMemoryContext  | 2 | TopMemoryContext->TransactionAbortContext
Portal hash              |   8192 | TopMemoryContext  | 2 | TopMemoryContext->Portal hash
TopPortalContext         |   8192 | TopMemoryContext  | 2 | TopMemoryContext->TopPortalContext
Relcache by OID          |  16384 | TopMemoryContext  | 2 | TopMemoryContext->Relcache by OID
CacheMemoryContext       | 524288 | TopMemoryContext  | 2 | TopMemoryContext->CacheMemoryContext
WAL record construction  |  50200 | TopMemoryContext  | 2 | TopMemoryContext->WAL record construction
PrivateRefCount          |   8192 | TopMemoryContext  | 2 | TopMemoryContext->PrivateRefCount
MdSmgr                   |   8192 | TopMemoryContext  | 2 | TopMemoryContext->MdSmgr
LOCALLOCK hash           |   8192 | TopMemoryContext  | 2 | TopMemoryContext->LOCALLOCK hash
GUCMemoryContext         |  24576 | TopMemoryContext  | 2 | TopMemoryContext->GUCMemoryContext
Timezones                | 104112 | TopMemoryContext  | 2 | TopMemoryContext->Timezones
ErrorContext             |   8192 | TopMemoryContext  | 2 | TopMemoryContext->ErrorContext
PortalContext            |   1024 | TopPortalContext  | 3 | TopMemoryContext->TopPortalContext->PortalContext
GUC hash table           |  32768 | GUCMemoryContext  | 3 | TopMemoryContext->GUCMemoryContext->GUC hash table
ExecutorState            | 147696 | PortalContext     | 4 | TopMemoryContext->TopPortalContext->PortalContext-
                                                                >ExecutorState
printtup                 |   8192 | ExecutorState     | 5 | TopMemoryContext->TopPortalContext->PortalContext-
                                                                >ExecutorState->printtup
ExprContext              |   8192 | ExecutorState     | 5 | TopMemoryContext->TopPortalContext->PortalContext-
                                                                >ExecutorState->ExprContext
ExprContext              |   8192 | ExecutorState     | 5 | TopMemoryContext->TopPortalContext->PortalContext-
                                                                >ExecutorState->ExprContext
ExprContext              |   8192 | ExecutorState     | 5 | TopMemoryContext->TopPortalContext->PortalContext-
                                                                >ExecutorState->ExprContext
Table function arguments |   8192 | ExecutorState     | 5 | TopMemoryContext->TopPortalContext->PortalContext-
                                                                >ExecutorState->Table function arguments
ExprContext              |   8192 | ExecutorState     | 5 | TopMemoryContext->TopPortalContext->PortalContext-
                                                                >ExecutorState->ExprContext
ExprContext              |   8192 | ExecutorState     | 5 | TopMemoryContext->TopPortalContext->PortalContext-
                                                                >ExecutorState->ExprContext
Table function arguments |   8192 | ExecutorState     | 5 | TopMemoryContext->TopPortalContext->PortalContext-
                                                                >ExecutorState->Table function arguments
ExprContext              |   8192 | ExecutorState     | 5 | TopMemoryContext->TopPortalContext->PortalContext-
                                                                >ExecutorState->ExprContext
(35 rows)
```

```
  total   | blocks |  used   |  free  | f_chunks
---------+--------+---------+--------+----------
 1387480 |    202 | 1033400 | 354080 |      134
(1 row)
```

Первый и третий запрос показывает общую память серверного процесса (total), используемую (used), выделенную, но не используемую (free). Число блоков и чанков не информативны, так как размеры блоков и чанков разные и неизвестны.

Для получения размера памяти контекста нужно суммировать с его размером размер памяти его дочерних контекстов.

Второй запрос выдаёт иерархию контекстов памяти. Для компактности из вывода убраны части системного кэша (`CacheMemoryContext`), в которых в столбце ident (убран из вывода для компактности) присутствуют названия объектов системного каталога.

Результат третьего запроса отличается от первого: total и used немного увеличились. Это произошло из-за того, что для выполнения запросов выделялась память. После выполнения

запросов контексты освобождались, но уже выделенная процессу память не возвращается операционной системе.

2) Выполните команды:

```
postgres=# set temp_buffers = '100MB';
create temp table temp1 (id integer);
insert into temp1 select * from generate_series(1, 1000000);
explain (analyze, buffers) select * from temp1;
drop table temp1;
select sum(total_bytes) total, sum(total_nblocks) blocks, sum(used_bytes) used,
sum(free_bytes) free, sum(free_chunks) f_chunks from pg_backend_memory_contexts;
with recursive dep as
(select name, total_bytes as total, ident, parent, 1 as level, name as path from
pg_backend_memory_contexts where parent is null
union all
select c.name, c.total_bytes, c.ident, c.parent, p.level + 1, p.path || '->' ||
c.name
from dep p, pg_backend_memory_contexts c
where c.parent = p.name)
select name, total, parent, level l, path  from dep where name like '%Local%';
SET
CREATE TABLE
INSERT 0 1000000
                                                    QUERY PLAN
------------------------------------------------------------------------------------
 Seq Scan on temp1  (cost=0.00..15708.75 rows=1128375 width=4) (actual time=0.026..4071.777 rows=1000000
loops=1)
   Buffers: local hit=4425
 Planning:
   Buffers: shared hit=20
 Planning Time: 0.125 ms
 Execution Time: 8043.230 ms
(6 rows)

DROP TABLE
  total   | blocks |   used   |  free  | f_chunks
----------+--------+----------+--------+----------
 68948568 |    239 | 68399976 | 548592 |      167
(1 row)

          name           |  total   |      parent      | l |              path
-------------------------+----------+------------------+---+--------------------------------
 LocalBufferContext      | 67023280 | TopMemoryContext | 2 | TopMemoryContext->LocalBufferContext
 Local Buffer Lookup Table |   524288 | TopMemoryContext | 2 | TopMemoryContext->Local Buffer Lookup Table
(2 rows)
```

Команды установили размер буфера под блоки временных таблиц в 10. Создали временную таблицу, заполнили ее блоками локальный буфер и удалили временную таблицу. Локальный буфер не освободил память постольку поскольку в этой сессии могут ещё создаваться временные таблицы, а выделять в операционной системе и возвращать операционной системе память неоптимально.

3) **Освободить память локального буфера в течение сессии нельзя, только закрыть сессию:**

```
postgres=# set temp_buffers = '8MB';
ERROR:  invalid value for parameter "temp_buffers": 1024
DETAIL:  "temp_buffers" cannot be changed after any temporary tables have been
accessed in the session.
```

Команда RESET не выдает ошибку, но буфер не освобождается:

```
postgres=# reset temp_buffers;
show temp_buffers;
RESET
 temp_buffers
```

```
--------------
 8MB
(1 row)

postgres=# select sum(total_bytes) total, sum(total_nblocks) blocks,
sum(used_bytes) used, sum(free_bytes) free, sum(free_chunks) f_chunks from
pg_backend_memory_contexts;
   total    | blocks |   used   |  free  | f_chunks
------------+--------+----------+--------+----------
 68998536 |    244 | 68455416 | 543120 |      207
(1 row)

postgres=# with recursive dep as
 (select name, total_bytes as total, ident, parent, 1 as level, name as path from
pg_backend_memory_contexts where parent is null
 union all
 select c.name, c.total_bytes, c.ident, c.parent, p.level + 1, p.path || '->' ||
c.name
 from dep p, pg_backend_memory_contexts c
 where c.parent = p.name)
 select name, total, parent, level l, path  from dep where name like '%Local%';
          name           |  total   |      parent      | l |                    path
-------------------------+----------+------------------+---+-----------------------------------------
 LocalBufferContext      | 67023280 | TopMemoryContext | 2 | TopMemoryContext->LocalBufferContext
 Local Buffer Lookup Table |   524288 | TopMemoryContext | 2 | TopMemoryContext->Local Buffer Lookup Table
(2 rows)
```

4) Более того, если повторить команды создания и удаления временной таблицы, то локальный буфер вырастет в размере вплоть до установленных **set temp_buffers = '100MB'**:

```
postgres=# create temp table temp1 (id integer);
insert into temp1 select * from generate_series(1, 1000000);
explain (analyze, buffers) select * from temp1;
drop table temp1;
select sum(total_bytes) total, sum(total_nblocks) blocks, sum(used_bytes) used,
sum(free_bytes) free, sum(free_chunks) f_chunks from pg_backend_memory_contexts;
with recursive dep as
(select name, total_bytes as total, ident, parent, 1 as level, name as path from
pg_backend_memory_contexts where parent is null
union all
select c.name, c.total_bytes, c.ident, c.parent, p.level + 1, p.path || '->' || c.name
from dep p, pg_backend_memory_contexts c
where c.parent = p.name)
select name, total, parent, level l, path  from dep where name like '%Local%';
CREATE TABLE
INSERT 0 1000000
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------
 Seq Scan on temp1  (cost=0.00..15708.75 rows=1128375 width=4) (actual time=0.028..4069.873 rows=1000000
loops=1)
   Buffers: local hit=4425
 Planning:
   Buffers: shared hit=4
 Planning Time: 0.050 ms
 Execution Time: 8078.346 ms
(6 rows)

DROP TABLE
   total    | blocks |   used    |  free   | f_chunks
------------+--------+-----------+---------+----------
 107406776 |    246 | 106343504 | 1063272 |      211
(1 row)

          name           |   total   |      parent      | l |                    path
-------------------------+-----------+------------------+---+-----------------------------------------
 LocalBufferContext      | 104907232 | TopMemoryContext | 2 | TopMemoryContext->LocalBufferContext
 Local Buffer Lookup Table |    524288 | TopMemoryContext | 2 | TopMemoryContext->Local Buffer Lookup Table
(2 rows)
```

5) Не только локальный буфер не освобождает память. Также не возвращает память операционной системе кэш системного каталога. В новой сессии psql выполните команды:

```
postgres=# drop table if exists parttab;
create table parttab(n numeric,k numeric, v varchar(100)) partition by range (n);
do
$$
declare
cnt integer;
v varchar(200);
begin
for i in 0..2000 loop
v:= concat('create table parttab_',i,' partition of parttab for values from
(',i*10,') to (',(i+1)*10,')');
execute v;
end loop;
end;
$$
;

 select sum(total_bytes) total, sum(total_nblocks) blocks, sum(used_bytes) used,
sum(free_bytes) free, sum(free_chunks) f_chunks from pg_backend_memory_contexts;

with recursive dep as
 (select name, total_bytes as total, used_bytes as used, free_bytes as free,
ident, parent, 1 as level, name as path from pg_backend_memory_contexts where
parent is null
union all
select c.name, c.total_bytes, c.used_bytes, c.free_bytes, c.ident, c.parent,
p.level + 1, p.path || '->' || c.name
from dep p, pg_backend_memory_contexts c
where c.parent = p.name)
select name, total, used, free, parent, level l, path  from dep where name like
'%CacheMemory%';

 drop table if exists parttab;

select sum(total_bytes) total, sum(total_nblocks) blocks, sum(used_bytes) used,
sum(free_bytes) free, sum(free_chunks) f_chunks from pg_backend_memory_contexts;

with recursive dep as
 (select name, total_bytes as total, used_bytes as used, free_bytes as free,
ident, parent, 1 as level, name as path from pg_backend_memory_contexts where
parent is null
union all
select c.name, c.total_bytes, c.used_bytes, c.free_bytes, c.ident, c.parent,
p.level + 1, p.path || '->' || c.name
from dep p, pg_backend_memory_contexts c
where c.parent = p.name)
select name, total, used, free, parent, level l, path  from dep where name like
'%CacheMemory%';
```

```
 NOTICE:  table "parttab" does not exist, skipping
DROP TABLE
CREATE TABLE
DO
   total    | blocks |  used   |   free   | f_chunks
 ----------+--------+---------+----------+----------
  21854616 |    223 | 2016048 | 19838568 |    52954
 (1 row)

       name        |  total   |  used  |   free   |      parent      | l |                 path
 ------------------+----------+--------+----------+------------------+---+--------------------------------------
 CacheMemoryContext | 16924912 | 485512 | 16439400 | TopMemoryContext | 2 | TopMemoryContext->CacheMemoryContext
 (1 row)
```

```
DROP TABLE
NOTICE:  table "parttab" does not exist, skipping
DROP TABLE
  total   | blocks |  used   |   free   | f_chunks
----------+--------+---------+----------+----------
 24485160 |    403 | 3309016 | 21176144 |    69436
(1 row)

       name        |  total   |  used  |   free   |      parent      | l |                 path
-------------------+----------+--------+----------+------------------+---+--------------------------------------
CacheMemoryContext | 17023312 | 585528 | 16437784 | TopMemoryContext | 2 | TopMemoryContext->CacheMemoryContext
(1 row)
```

В отличие от локального буфера кэш системного каталога пометил память как свободную.

В память кэша системного каталога загружались данные о 2000 секциях таблицы при выполнении команды **`execute v;`**. Секции использовались как простой пример. При реальной работе кэш системного каталога заполнится данными о большом числе объектов, если в команде **на этапе планирования** были обращения за информацией о большом числе объектов базы данных. На этапе выполнения команды число объектов может уменьшиться или остаться таким же, если все объекты остались в созданном плане.

В результате показан размер только корня `CacheMemoryContext`. Размер его его наследников не показан.

6) Кэш системного каталога удерживает данные до окончания запроса или транзакции?

Можно предположить, что с одной стороны изменения в системном каталоге должны стать видны по окончании транзакции, с другой стороны сама транзакция должна видеть свои изменения. Для проверки в новой сессии psql выполните команды:

```
postgres=# \q
postgres@tantor:~$ psql

postgres=# drop table if exists parttab;
create table parttab(n numeric,k numeric, v varchar(100)) partition by range
(n);
begin transaction;
do
$$
declare
cnt integer;
v varchar(200);
begin
for i in 0..3000 loop
v:= concat('create table parttab_',i,' partition of parttab for values from
(',i*10,') to (',(i+1)*10,')');
execute v;
end loop;
end;
$$
;
select sum(total_bytes) total, sum(total_nblocks) blocks, sum(used_bytes) used,
sum(free_bytes) free, sum(free_chunks) f_chunks from pg_backend_memory_contexts;

with recursive dep as
(select name, total_bytes as total, used_bytes as used, free_bytes as free,
ident, parent, 1 as level, name as path from pg_backend_memory_contexts where
parent is null
union all
select c.name, c.total_bytes, c.used_bytes, c.free_bytes, c.ident, c.parent,
p.level + 1, p.path || '->' || c.name
from dep p, pg_backend_memory_contexts c
where c.parent = p.name)
```

```
 select name, total, used, free, parent, level l, path  from dep where name like
'%CacheMemory%';

 commit;

 NOTICE:  table "parttab" does not exist, skipping
 DROP TABLE
 CREATE TABLE
 BEGIN
 DO
 DO
 DO
   total    | blocks |   used    |  free   | f_chunks
 -----------+--------+-----------+---------+----------
  43441704 |   6396 | 37203376 | 6238328 |     3553
 (1 row)

        name        |  total   |   used   |  free   |     parent     | l |                 path
 -------------------+----------+----------+---------+----------------+---+------------------------------
 CacheMemoryContext| 25460976 | 22374816 | 3086160 |TopMemoryContext| 2 |TopMemoryContext->CacheMemoryContext
 (1 row)

 COMMIT
```

Выполнялись три команды, добавляющие по 1000 секций в таблицу. Размер корня контекста `CacheMemoryContext` стал `25460976` байт, а в предыдущем примере при добавлении 2000 секций был `16924912`.

7) Проверьте, что после фиксации транзакции содержимое контекста `CacheMemoryContext` освободилось, но память, выделенная `CacheMemoryContext` не будет возвращена операционной системе:

```
 postgres=# select sum(total_bytes) total, sum(total_nblocks) blocks,
sum(used_bytes) used, sum(free_bytes) free, sum(free_chunks) f_chunks from
pg_backend_memory_contexts;

 with recursive dep as
 (select name, total_bytes as total, used_bytes as used, free_bytes as free,
ident, parent, 1 as level, name as path from pg_backend_memory_contexts where
parent is null
 union all
 select c.name, c.total_bytes, c.used_bytes, c.free_bytes, c.ident, c.parent,
p.level + 1, p.path || '->' || c.name
 from dep p, pg_backend_memory_contexts c
 where c.parent = p.name)
 select name, total, used, free, parent, level l, path  from dep where name like
'%CacheMemory%';
   total    | blocks |  used   |   free   | f_chunks
 -----------+--------+---------+----------+----------
  31021256 |    269 | 2671688 | 28349568 |    79149
 (1 row)

        name         |  total   |  used  |   free   |     parent     | l |                 path
 --------------------+----------+--------+----------+----------------+---+-------------------------------------
 CacheMemoryContext | 25460976 | 633008 | 24827968 |TopMemoryContext| 2 |TopMemoryContext->CacheMemoryContext
 (1 row)
```

При этом часть памяти серверного процесса освободилась: стала `31021256`, а была `43441704`. Большая часть освобожденной памяти (`5818512`) относилась к `TopTransactionContext`, который полностью освобождается после завершения транзакции. Другая часть относилась к контекстам "index info" (3000 штук по 2048 байт), наследникам `CacheMemoryContext`.

8) Удалите таблицу `parttab`:

```
 postgres=# drop table parttab;
 DROP TABLE
```

Часть 4. Логирование памяти процесса в диагностический журнал

1) Чтобы просматривать диагностический журнал в удобном месте - директории `PGDATA/log` включите `logging_collector`:

```
postgres=# \q
postgres@tantor:~$ psql -c "alter system set logging_collector=on"
ALTER SYSTEM
postgres@tantor:~$ sudo restart
postgres@tantor:~$ cat $PGDATA/current_logfiles
stderr log/postgresql-YYYY-MM-DD_HHmiss.log
postgres@tantor:~$ ls $PGDATA/log
postgresql-YYYY-MM-DD_HHmiss.log
postgres@tantor:~$ psql
```

Название текущего файла диагностического журнала указывается в файле **`$PGDATA/current_logfiles`**. В названии файла присутствуют дата и время его создания.

2) Выведите в журнал данные по памяти своего серверного процесса:

```
postgres=# select pg_log_backend_memory_contexts(pg_backend_pid());
 pg_log_backend_memory_contexts
--------------------------------
 t
(1 row)
```

3) Посмотрите что появилось конце файла лога:

```
postgres=# \! tail -112 $PGDATA/log/postgresql-20*
LOG:  level: 0; TopMemoryContext: 97664 total in 5 blocks; 13920 free (8 chunks); 83744 used
LOG:  level: 1; TopTransactionContext: 8192 total in 1 blocks; 7752 free (1 chunks); 440 used
LOG:  level: 1; RowDescriptionContext: 8192 total in 1 blocks; 6896 free (0 chunks); 1296 used
LOG:  level: 1; MessageContext: 16384 total in 2 blocks; 5960 free (0 chunks); 10424 used
LOG:  level: 1; Operator class cache: 8192 total in 1 blocks; 592 free (0 chunks); 7600 used
LOG:  level: 1; smgr relation table: 16384 total in 2 blocks; 4640 free (2 chunks); 11744 used
LOG:  level: 1; PgStat Shared Ref Hash: 7216 total in 2 blocks; 688 free (0 chunks); 6528 used
LOG:  level: 1; PgStat Shared Ref: 4096 total in 3 blocks; 1808 free (2 chunks); 2288 used
LOG:  level: 1; PgStat Pending: 8192 total in 4 blocks; 5960 free (21 chunks); 2232 used
LOG:  level: 1; TransactionAbortContext: 32768 total in 1 blocks; 32504 free (0 chunks); 264 used
LOG:  level: 1; Portal hash: 8192 total in 1 blocks; 592 free (0 chunks); 7600 used
LOG:  level: 1; TopPortalContext: 8192 total in 1 blocks; 7664 free (0 chunks); 528 used
LOG:  level: 2; PortalContext: 1024 total in 1 blocks; 608 free (0 chunks); 416 used: <unnamed>
LOG:  level: 3; ExecutorState: 8192 total in 1 blocks; 3920 free (0 chunks); 4272 used
LOG:  level: 4; printtup: 8192 total in 1 blocks; 7928 free (0 chunks); 264 used
LOG:  level: 4; ExprContext: 8192 total in 1 blocks; 7928 free (0 chunks); 264 used
LOG:  level: 1; Relcache by OID: 16384 total in 2 blocks; 3584 free (2 chunks); 12800 used
LOG:  level: 1; CacheMemoryContext: 524288 total in 7 blocks; 117128 free (0 chunks); 407160 used
...
LOG:  level: 2; index info: 2048 total in 2 blocks; 912 free (0 chunks); 1136 used: pg_authid_rolname_index
LOG:  level: 1; WAL record construction: 50200 total in 2 blocks; 6376 free (0 chunks); 43824 used
LOG:  level: 1; PrivateRefCount: 8192 total in 1 blocks; 2648 free (0 chunks); 5544 used
LOG:  level: 1; MdSmgr: 8192 total in 1 blocks; 7912 free (0 chunks); 280 used
LOG:  level: 1; LOCALLOCK hash: 8192 total in 1 blocks; 592 free (0 chunks); 7600 used
LOG:  level: 1; GUCMemoryContext: 40960 total in 3 blocks; 28968 free (11 chunks); 11992 used
LOG:  level: 2; GUC hash table: 32768 total in 3 blocks; 12704 free (5 chunks); 20064 used
LOG:  level: 1; Timezones: 104112 total in 2 blocks; 2648 free (0 chunks); 101464 used
LOG:  level: 1; ErrorContext: 8192 total in 1 blocks; 7928 free (4 chunks); 264 used
LOG:  Grand total: 1201272 bytes in 183 blocks; 345272 free (139 chunks); 856000 used
```

4) Сравните последнюю строку **`Grand total:`** с результатом запроса к представлению **`pg_backend_memory_contexts`**:

```
postgres=# select sum(total_bytes) total, sum(total_nblocks) blocks,
sum(used_bytes) used, sum(free_bytes) free, sum(free_chunks) f_chunks from
pg_backend_memory_contexts;
  total   | blocks |  used   |  free   | f_chunks
---------+--------+---------+--------+----------
```

```
 1337512 |     197 | 968264 | 369248 |       138
(1 row)
```

Результат похож. Отличия в том, что результат запроса учитывает контексты локальной памяти процесса, которые использовались для выполнения самого запроса.

Логирование памяти в журнал менее удобно, чем запрос к представлению. Логирование функцией **pg_log_backend_memory_contexts(PID)** используется для получения данных о контекстах памяти чужих процессов в процессе их работы.

Часть 5. Взаимоблокировки при проведении тестов

1) Выполните команды в терминале linux:

```
postgres@tantor:~$ psql -c "drop table if exists t cascade;"
psql -c "create table t(pk bigserial, c1 text default
'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa');"
psql -c "insert into t select *, 'a' from generate_series(1, 10);"
psql -c "alter table t add constraint pk primary key (pk);"
echo "select * from t for update;" > lock1.sql
echo "update t set c1='aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa';" >> lock1.sql
pgbench -T 300 -c 9 -P 5 -f lock1.sql

DROP TABLE
CREATE TABLE
INSERT 0 10
ALTER TABLE
starting vacuum...end.
progress: 5.0 s, 16.4 tps, lat 255.268 ms stddev 571.587, 12 failed
progress: 10.0 s, 15.8 tps, lat 297.406 ms stddev 670.923, 18 failed
progress: 15.0 s, 12.2 tps, lat 464.690 ms stddev 1119.552, 7 failed
progress: 20.0 s, 0.2 tps, lat 5073.576 ms stddev 0.000, 5 failed
progress: 25.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 5 failed
progress: 30.0 s, 6.2 tps, lat 1263.564 ms stddev 2396.412, 9 failed
```

Команды запускают тест на 5 минут. Тест раз в 5 секунд выдаёт промежуточные результаты на терминал, что удобно для наблюдения за показателями: `tps, latency` и другими показателями.

2) Во втором терминале выполните команды:

```
postgres@tantor:~$ psql -c "alter system set deadlock_timeout = '10s';"
psql -c "select pg_reload_conf();"
ALTER SYSTEM
 pg_reload_conf
----------------
 t
(1 row)
```

Команды увеличивают значение параметра конфигурации **deadlock_timeout** до 10 секунд.

2) Посмотрите как изменится вывод утилиты pgbench:

```
progress: 70.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 75.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 5 failed
progress: 80.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 85.0 s, 3.4 tps, lat 6541.419 ms stddev 7628.907, 1 failed
progress: 90.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
```

Увеличение значения параметра плохо повлияло на tps. tps уменьшился до нуля.

Попробуйте выяснить причину почему такое произошло. Ведь увеличение значения **deadlock_timeout** должно благотворно влиять на производительность.

3) Если изменение параметра конфигурации приводит к деградации производительности, то нужно возвращать значение обратно или менять в другую сторону. Уменьшите значение **deadlock_timeout** до 10 миллисекунд:

```
postgres@tantor:~$ psql -c "alter system set deadlock_timeout = '10ms';"
psql -c "select pg_reload_conf();"
ALTER SYSTEM
 pg_reload_conf
----------------
 t
(1 row)
```

Изменения вступят в силу не сразу и даже не через 10 секунд.
Дождитесь, когда tps увеличится с нуля до ~100 и прервите работу pgbench набрав **<ctrl+c>**:

```
progress: 205.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 210.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 6 failed
progress: 215.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 220.0 s, 91.8 tps, lat 491.150 ms stddev 5336.318, 63 failed
progress: 225.0 s, 106.4 tps, lat 74.779 ms stddev 35.140, 90 failed
progress: 230.0 s, 88.4 tps, lat 78.530 ms stddev 35.427, 156 failed
progress: 235.0 s, 113.6 tps, lat 74.780 ms stddev 35.355, 40 failed
progress: 240.0 s, 104.0 tps, lat 74.289 ms stddev 34.357, 96 failed
progress: 245.0 s, 106.6 tps, lat 75.671 ms stddev 33.622, 73 failed
^C
postgres@tantor:~$
```

4) Верните значение **deadlock_timeout** к значению по умолчанию.

```
postgres@tantor:~$ psql -c "alter system reset deadlock_timeout;"
psql -c "select pg_reload_conf();"
ALTER SYSTEM
 pg_reload_conf
----------------
 t
(1 row)
```

5) Для поиска причин ошибок стоит просматривать диагностический журнал. Посмотрите его содержимое:

```
postgres@tantor:~$ tail -40 $PGDATA/log/postgresql-20*
02:50:31.635 MSK [33653] ERROR:  deadlock detected
02:50:31.635 MSK [33653] DETAIL:  Process 33653 waits for ShareLock on transaction 1704599; blocked by
process 33652.
        Process 33652 waits for ShareLock on transaction 1704594; blocked by process 33659.
        Process 33659 waits for ExclusiveLock on tuple (18,5) of relation 3013274 of database 5; blocked by
process 33653.
        Process 33653: select * from t for update;
        Process 33652: select * from t for update;
        Process 33659: update t set c1='aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa';
02:50:31.635 MSK [33653] HINT:  See server log for query details.
02:50:31.635 MSK [33653] CONTEXT:  while locking tuple (18,5) in relation "t"
02:50:31.635 MSK [33653] STATEMENT:  select * from t for update;
02:50:31.646 MSK [33659] ERROR:  deadlock detected
02:50:31.646 MSK [33659] DETAIL:  Process 33659 waits for ShareLock on transaction 1704599; blocked by
process 33652.
        Process 33652 waits for ShareLock on transaction 1704594; blocked by process 33659.
        Process 33659: update t set c1='aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa';
        Process 33652: select * from t for update;
02:50:31.646 MSK [33659] HINT:  See server log for query details.
02:50:31.646 MSK [33659] CONTEXT:  while locking tuple (18,5) in relation "t"
02:50:31.646 MSK [33659] STATEMENT:  update t set c1='aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa';
02:50:32.622 MSK [33658] LOG:  could not send data to client: Broken pipe
02:50:32.622 MSK [33658] FATAL:  connection to client lost
02:50:32.662 MSK [33656] LOG:  could not send data to client: Broken pipe
02:50:32.662 MSK [33656] FATAL:  connection to client lost
02:52:44.350 MSK [32258] LOG:  received SIGHUP, reloading configuration files
02:52:44.350 MSK [32258] LOG:  parameter "deadlock_timeout" removed from configuration file, reset to default
02:53:26.379 MSK [32260] LOG:  checkpoint starting: time
```

```
   02:53:30.074 MSK [32260] LOG:  checkpoint complete: wrote 42 buffers (0.3%); 0 WAL file(s) added, 1 removed,
0 recycled; write=3.615 s, sync=0.030 s, total=3.696 s; sync files=7, longest=0.018 s, average=0.005 s;
distance=7573 kB, estimate=85834 kB; lsn=2C/42273240, redo lsn=2C/422731F8
```

Ошибка "`could not send data to client: Broken pipe`" это следствие прерывания работы утилиты pgbench нажатием клавиш `^C`.

Ошибки "`ERROR:  deadlock detected`" это и есть причина уменьшения tps при увеличении значения параметра и наоборот: увеличения tps при уменьшении значения параметра до 10 миллисекунд.

Утилита pgbench сигнализировала об ошибках сообщениями в конце строк "`failed`", указывающими на то, что СУБД выдавала ошибки. Падение tps до нуля происходит, если набор команд в скрипте теста выполняется дольше интервала, задаваемого параметром "`-P 5`". В том интервале, где скрипт выполняется tps увеличивается и становится больше нуля.

Часть 6. Мультитранзакции

1) Выполните команды:

```
postgres=# drop table if exists t;
create table t (c int); insert into t values(1);
begin;
select c from t where c=1 for update;
savepoint s1;
update t set c=1 where c=1;
commit;
DROP TABLE
CREATE TABLE
INSERT 0 1
BEGIN
 c
---
 1
(1 row)

SAVEPOINT
UPDATE 1
COMMIT
```

Команда обновления строки создаёт мультитранзакцию.

2) Посмотрите содержимое блока:

```
postgres=# select lp, lp_off, lp_len, t_ctid, t_xmin, t_xmax, t_ctid,
t_infomask, (t_infomask&4096)!=0 as m from heap_page_items(get_raw_page('t', 0));
select * from heap_page('t',0);
 lp | lp_off | lp_len | t_ctid | t_xmin  | t_xmax  | t_ctid | t_infomask | m
----+--------+--------+--------+---------+---------+--------+------------+---
  1 |   8144 |     28 | (0,2)  | 1695169 |   21016 | (0,2)  |       4416 | t
  2 |   8112 |     28 | (0,2)  | 1695171 | 1695170 | (0,2)  |       8336 | f
(2 rows)

 lp_off | ctid  | state  |   xmin    |  xmax   | hhu | hot | t_ctid | multi
--------+-------+--------+-----------+---------+-----+-----+--------+-------
 8144   | (0,1) | normal | 1695169c  | 21016   | t   |     | (0,2)  | t
 8112   | (0,2) | normal | 1695171   | 1695170 |     | t   | (0,2)  | f
(2 rows)
```

В первой версии строки `lp=1` установлен бит-подсказка, что в версии строки используется мультитранзакция. В поле заголовка строки `t_xmax` указан номер `21016`, относящийся к

нумерации мультитранзакций. Также организована цепочка HOT, так как на таблице нет индексов.
hhu - подсказка процессам, что надо идти по цепочке ctid.
hot - на данную версию строки нет ссылок из индексов.
HOT cleanup не был выполнен потому, что не было условий для его срабатывания - в блоке больше 10% свободного места.

3) Прочтите строку и посмотрите содержимое блока:

```
postgres=# select * from t;
select * from heap_page('t',0);
 c
---
 1
(1 row)

 lp_off | ctid  | state  |   xmin   |  xmax   | hhu | hot | t_ctid | multi
--------+-------+--------+----------+---------+-----+-----+--------+-------
 8144   | (0,1) | normal | 1695169c | 21016   | t   |     | (0,2)  | t
 8112   | (0,2) | normal | 1695171c | 1695170 |     | t   | (0,2)  | f
(2 rows)
```

При чтении стоки процесс проверил, что транзакци, породившая версию строки `(0,2)` была зафиксирована и установил бит-подсказку.

4) Выполните вакуумирование таблицы:

```
postgres=# vacuum verbose t;
INFO:  vacuuming "postgres.public.t"
INFO:  finished vacuuming "postgres.public.t": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total)
tuples: 1 removed, 1 remain, 0 are dead but not yet removable, oldest xmin: 1695172
removable cutoff: 1695172, which was 0 XIDs old when operation ended
new relfrozenxid: 1695172, which is 4 XIDs ahead of previous value
new relminmxid: 21017, which is 1 MXIDs ahead of previous value
frozen: 1 pages from table (100.00% of total) had 1 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 113.499 MB/s
buffer usage: 6 hits, 0 misses, 6 dirtied
WAL usage: 8 records, 6 full page images, 41033 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

Горизонт базы не удерживался и вакуум очистил блок от старой версии строки.

5) Посмотрите содержимое блока после вакуумирования:

```
postgres=# select * from heap_page('t',0);
 lp_off | ctid  |     state     | xmin | xmax | hhu | hot | t_ctid | multi
--------+-------+---------------+------+------+-----+-----+--------+-------
 2      | (0,1) | redirect to 2 |      |      |     |     |        | f
 8144   | (0,2) | normal        | 2c   | 0a   |     | t   | (0,2)  | f
(2 rows)
```

Признаки `c` `a` и xmin=2 означают, что версия строки была заморожена вакуумом. Блок был очищен от номеров мультитранзакций.

7) В другом терминале с psql выполните команды, которые будут удерживать горизонт базы:

```
postgres=# begin transaction;
select pg_current_xact_id();
BEGIN
```

```
 pg_current_xact_id
--------------------
             1695208
(1 row)

postgres=*#
```

6) Выполните команды, которые повторяют создание версии строки:

```
postgres=# drop table if exists t;
create table t (c int);
insert into t values(1);
begin;
select c from t where c=1 for update;
savepoint s1;
update t set c=1 where c=1;
commit;
select * from t;
select * from heap_page('t',0);
vacuum verbose t;
select * from heap_page('t',0);
DROP TABLE
CREATE TABLE
INSERT 0 1
BEGIN
 c
---
 1
(1 row)

SAVEPOINT
UPDATE 1
COMMIT
 c
---
 1
(1 row)

 lp_off | ctid  | state  |   xmin   |  xmax   | hhu | hot | t_ctid | multi
--------+-------+--------+----------+---------+-----+-----+--------+-------
 8144   | (0,1) | normal | 1695211c | 21025   | t   |     | (0,2)  | t
 8112   | (0,2) | normal | 1695213c | 1695212 |     | t   | (0,2)  | f
(2 rows)

INFO:  vacuuming "postgres.public.t"
INFO:  finished vacuuming "postgres.public.t": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total)
tuples: 0 removed, 2 remain, 1 are dead but not yet removable, oldest xmin: 1695208
removable cutoff: 1695208, which was 6 XIDs old when operation ended
new relminmxid: 21026, which is 1 MXIDs ahead of previous value
frozen: 1 pages from table (100.00% of total) had 2 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 75.120 MB/s
buffer usage: 6 hits, 0 misses, 3 dirtied
WAL usage: 5 records, 3 full page images, 25027 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM

 lp_off | ctid  | state  |   xmin   |  xmax   | hhu | hot | t_ctid | multi
--------+-------+--------+----------+---------+-----+-----+--------+-------
 8144   | (0,1) | normal | 1695211c | 1695213 | t   |     | (0,2)  | f
 8112   | (0,2) | normal | 1695213c | 0a      |     | t   | (0,2)  | f
(2 rows)
```

Вакуум не смог очистить ни одной версии строк. При этом вакуум убрал данные о мультитранзакции и установил признак заморозки версии строки - биты-подсказки `ca` .

Если в блоке присутствуют признаки мультитранзакций, то процессы чтобы определить статус транзакций, входящих в мультитранзакцию должны обращаться к буферам размеры, которых задаются параметрами конфигурации `multixact_member_buffers` и

multixact_offset_buffers. Очистка версий строк от мультитранзакций позволяет не обращаться к буферам.

7) Во второй сессии завершите открытую транзакцию:

```
postgres=*# commit;
COMMIT
postgres=#
```

8) Выполните вакуумирование таблицы:

```
postgres=# vacuum verbose t;
select * from heap_page('t',0);
INFO:  vacuuming "postgres.public.t"
INFO:  finished vacuuming "postgres.public.t": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total)
tuples: 1 removed, 1 remain, 0 are dead but not yet removable, oldest xmin: 1695214
removable cutoff: 1695214, which was 0 XIDs old when operation ended
new relfrozenxid: 1695214, which is 6 XIDs ahead of previous value
frozen: 1 pages from table (100.00% of total) had 1 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 228.659 MB/s
buffer usage: 10 hits, 0 misses, 6 dirtied
WAL usage: 8 records, 6 full page images, 41521 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
 lp_off | ctid  |     state      | xmin | xmax | hhu | hot | t_ctid | multi
--------+-------+----------------+------+------+-----+-----+--------+-------
 2      | (0,1) | redirect to 2 |      |      |     |     |        | f
 8144   | (0,2) | normal         | 2c   | 0a   |     | t   | (0,2)  | f
(2 rows)
```

Вакуум очистил блок от старых версий, вышедших за горизонт. xmin у замороженной версии строки был заменен на номер транзакции 2, который также используется как признак замороженной версии.

9) Посмотрим как меняет заголовки строк заморозка. Выполните команды:

```
postgres=# \set ON_ERROR_ROLLBACK interactive \\
truncate table t;
insert into t values(1);
insert into t values(2);
begin;
select * from t for share;
update t set c=1 where c=1;
update t set c=1 where c=1;
commit;
select * from heap_page('t',0);
vacuum freeze t;
select * from heap_page('t',0);
TRUNCATE TABLE
INSERT 0 1
INSERT 0 1
BEGIN
 c
---
 1
 2
(2 rows)

SAVEPOINT
UPDATE 1
UPDATE 1
COMMIT
 lp_off | ctid  | state  |   xmin   |  xmax   | hhu | hot | t_ctid | multi
--------+-------+--------+----------+---------+-----+-----+--------+-------
 8144   | (0,1) | normal | 2036130c | 108921  | t   |     | (0,3)  | t
```

```
 8112   | (0,2) | normal | 2036131c | 2036133 |      |      | (0,2)  | f
 8080   | (0,3) | normal | 2036136  | 108922  | t    | t    | (0,4)  | t
 8048   | (0,4) | normal | 2036137  | 2036133 |      | t    | (0,4)  | f
(4 rows)

VACUUM
 lp_off | ctid  |     state     | xmin | xmax | hhu | hot | t_ctid | multi
--------+-------+---------------+------+------+-----+-----+--------+-------
 4      | (0,1) | redirect to 4 |      |      |     |     |        | f
 8144   | (0,2) | normal        | 2c   | 0a   |     |     | (0,2)  | f
 0      | (0,3) | unused        |      |      |     |     |        | f
 8112   | (0,4) | normal        | 2c   | 0a   |     | t   | (0,4)  | f
(4 rows)
```

Вставляется две строки, чтобы показать, что в блоке может быть несколько цепочек версий. **for share update** используется, чтобы показать, что и другие типы блокировок с точками сохранения устанавливаемыми явно и неявно (**\set ON_ERROR_ROLLBACK interactive неявно** устанавливает точки сохранения после каждой команды в транзакции) порождают мультитранзакции.

Заморозка установила биты-подсказки xmin commited и xmax aborted, а также xmin=2. xmax был заменен на 0, чтобы убрать номера мультитранзакций. У мультитранзакций свой счётчик номеров.

Часть 7. Пример теста

1) Выполните команды в терминале linux:

```
postgres@tantor:~$
echo "select abalance from pgbench_accounts where bid=1 limit 1;" >bquery.sql
pgbench -n -i --partitions=6 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql  2> /dev/null | egrep 'latency|tps'
pgbench -n -i --partitions=7 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql  2> /dev/null | egrep 'latency|tps'
pgbench -n -i --partitions=8 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql  2> /dev/null | egrep 'latency|tps'
pgbench -n -i --partitions=9 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql  2> /dev/null | egrep 'latency|tps'
pgbench -n -i --partitions=10 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql  2> /dev/null | egrep 'latency|tps'

latency average = 0.275 ms
tps = 3633.933144 (without initial connection time)
latency average = 0.280 ms
tps = 3570.395697 (without initial connection time)
latency average = 0.291 ms
tps = 3440.218181 (without initial connection time)
latency average = 0.319 ms
tps = 3132.354586 (without initial connection time)
latency average = 0.320 ms
tps = 3124.262538 (without initial connection time)
```

Тест из одного простого запроса. Хочется протестировать влияет ли на время планирования и выполнения запроса число relations, которые будут использоваться при планировании запроса. Для проведения теста используется таблица стандартно поставляемого теста, в ней меняется число секций от 5 до 10. По быстрому пути будет заблокировано 7 и менее секций с с индексами. На каждый процесс максимальное число блокировок по быстрому 16.

В проведённом тесте время (latency) увеличивается с увеличением числа секций, а tps пропорционально уменьшается с ростом числа секций.

Хочется автоматизировать тест, чтобы устранить "дублирование кода". Вдруг число итераций захочется сделать не 5, а больше.

2) Выполните те же команды с помощью скрипта, который автоматизирует выполнение тестов:

```
postgres@tantor:~$
for (( i=6; i <= 10; i+=1 ))
do
pgbench -n -i --partitions=$1 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql  2> /dev/null | egrep 'latency|tps'
done
latency average = 0.190 ms
tps = 5256.213787 (without initial connection time)
latency average = 0.181 ms
tps = 5539.238994 (without initial connection time)
latency average = 0.178 ms
tps = 5620.314925 (without initial connection time)
latency average = 0.181 ms
tps = 5509.927934 (without initial connection time)
latency average = 0.183 ms
tps = 5471.600964 (without initial connection time)
```

tps резко и значительно "улучшились". Более того, время выполнения запроса перестало зависеть от числа секций.

Хочется выяснить в чём причина увеличения tps на 70% и какой результат "правильный".

3) Проверьте, что команда в тесте выполняется без ошибок:

```
postgres@tantor:~$ psql -f bquery.sql
 abalance
----------
        0
(1 row)
```

Команда обычная и выполняется без ошибок.

4) Замените запрос на другой. В терминале пользователя postgres выполните команды:

```
postgres@tantor:~$ echo "select bbalance from pgbench_branches where bid=1 limit
1;" >bquery.sql
for (( i=5; i <= 7; i+=1 ))
do
pgbench -n -i --partitions=$1 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql  2> /dev/null | egrep 'latency|tps'
done
pgbench -n -i --partitions=5 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql  2> /dev/null | egrep 'latency|tps'
pgbench -n -i --partitions=6 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql  2> /dev/null | egrep 'latency|tps'
pgbench -n -i --partitions=7 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql  2> /dev/null | egrep 'latency|tps'
latency average = 0.181 ms
tps = 5539.150039 (without initial connection time)
latency average = 0.181 ms
tps = 5533.754683 (without initial connection time)
latency average = 0.182 ms
tps = 5492.463973 (without initial connection time)
latency average = 0.190 ms
tps = 5253.629658 (without initial connection time)
latency average = 0.181 ms
tps = 5511.707677 (without initial connection time)
latency average = 0.180 ms
tps = 5570.755883 (without initial connection time)
```

При использовании другого запроса результат при запуске pgbench из цикла и вне цикла одинаков.

5) Причина ошибки в названии переменной в скрипте. Замените переменную на **правильную** и выполните скрипт:

```
  postgres@tantor:~$ echo "select abalance from pgbench_accounts where bid=1
limit 1;" >bquery.sql
 for (( i=6; i <= 10; i+=1 ))
 do
 pgbench -n -i --partitions=$i > /dev/null 2> /dev/null
 pgbench -T 10 -f bquery.sql  2> /dev/null | egrep 'latency|tps'
 done
 latency average = 0.266 ms
 tps = 3755.374467 (without initial connection time)
 latency average = 0.270 ms
 tps = 3704.714460 (without initial connection time)
 latency average = 0.291 ms
 tps = 3440.145333 (without initial connection time)
 latency average = 0.318 ms
 tps = 3146.117186 (without initial connection time)
 latency average = 0.333 ms
 tps = 3006.232748 (without initial connection time)
```

Тест стал работать правильно и результат соответствует командам без цикла.
Усложнение скриптов может привести к опечаткам и ошибкам.

4) Для большей точности можно между итерациями теста сбрасывать кэш linux и рестартовать экземпляр. Можно не выполнять команды из этого пункта, а посмотреть пример:

```
 postgres@tantor:~$ echo "explain (analyze, timing off) select abalance from
pgbench_accounts where bid=1 limit 1;" > bquery.sql
 postgres@tantor:~$ su -
 Password: root
 root@tantor:~#
  for (( i=1; i <= 25; i+=1 ))
 do
 printf "$i "
 su - postgres -c "pgbench -n -i --partitions=$i > /dev/null 2> /dev/null"
 echo 3 > /proc/sys/vm/drop_caches
 systemctl restart tantor-se-server-16
 sleep 3
 su - postgres -c "pgbench -T 5 -f bquery.sql  2> /dev/null | egrep 'latency|tps'"
 done
 1 latency average = 0.238 ms
 tps = 4200.697407 (without initial connection time)
 2 latency average = 0.260 ms
 tps = 3840.529923 (without initial connection time)
 3 latency average = 0.277 ms
 tps = 3608.223482 (without initial connection time)
 4 latency average = 0.303 ms
 tps = 3302.631956 (without initial connection time)
 5 latency average = 0.321 ms
 tps = 3111.183002 (without initial connection time)
 6 latency average = 0.358 ms
 tps = 2793.766387 (without initial connection time)
 7 latency average = 0.367 ms
 tps = 2724.507091 (without initial connection time)
 8 latency average = 0.400 ms
 tps = 2498.189268 (without initial connection time)
 9 latency average = 0.414 ms
 tps = 2415.747116 (without initial connection time)
 10 latency average = 0.433 ms
 tps = 2310.439740 (without initial connection time)
 11 latency average = 0.459 ms
 tps = 2178.293809 (without initial connection time)
 12 latency average = 0.475 ms
 tps = 2104.057041 (without initial connection time)
 13 latency average = 0.497 ms
 tps = 2013.104397 (without initial connection time)
 14 latency average = 0.518 ms
 tps = 1930.415354 (without initial connection time)
 15 latency average = 0.537 ms
 tps = 1861.375380 (without initial connection time)
 16 latency average = 0.551 ms
```

```
tps = 1815.541104 (without initial connection time)
17 latency average = 0.589 ms
tps = 1697.304151 (without initial connection time)
18 latency average = 0.595 ms
tps = 1680.943106 (without initial connection time)
19 latency average = 0.624 ms
tps = 1603.523026 (without initial connection time)
20 latency average = 0.651 ms
tps = 1536.011902 (without initial connection time)
21 latency average = 0.672 ms
tps = 1487.413340 (without initial connection time)
22 latency average = 0.679 ms
tps = 1473.400025 (without initial connection time)
23 latency average = 0.707 ms
tps = 1414.174861 (without initial connection time)
24 latency average = 0.725 ms
tps = 1379.219506 (without initial connection time)
25 latency average = 0.734 ms
tps = 1361.741860 (without initial connection time)
root@tantor:~# exit
logout
postgres@tantor:~$
```

При проведении тестов стоит обратить внимание на частоту контрольных точек. В конце контрольной точки будет спад производительности.

Можно построить графики latency и tps:

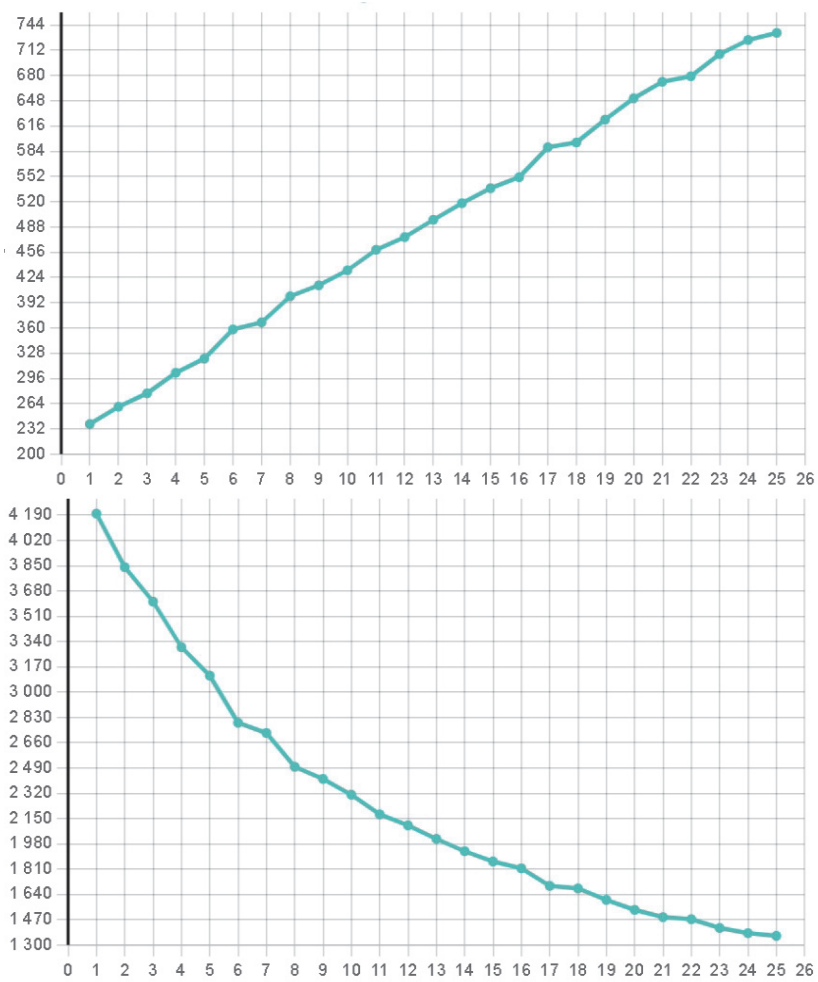


График latency, которая соответствует времени создания плана запроса к секционированной таблице в зависимости от числа секций линейный. При небольшом числе ядер и небольшой нагрузке использование fastpath не оказывает влияние на производительность.

Практика к главе 10

Часть 1. Расширение pg_buffercache

1) Установите расширение pg_buffercache:

```
postgres=# create extension pg_buffercache;
CREATE EXTENSION
```

Расширение полезно для наблюдения за буферным кэшем и получением общей информации. Расширение стандартное и его стоит использовать.

2) Посмотрите какие объекты входят в расширение:

```
postgres=# \dx+ pg_buffercache
 Objects in extension "pg_buffercache"
           Object description
--------------------------------------
 function pg_buffercache_pages()
 function pg_buffercache_summary()
 function pg_buffercache_usage_counts()
 view pg_buffercache
(4 rows)
```

Представление pg_buffercache, является обёрткой над функцией `pg_buffercache_pages()`. Так как представление выбирает все столбцы из результата функции, то использовать функцию `pg_buffercache_pages()` нет смысла.
Две функции без параметров `pg_buffercache_summary()` и `pg_buffercache_usage_counts()` не не устанавливают блокировки на структуры памяти буферного кэша, в отличие от первой функции и представления и работают быстро.

3) Выполните команду:

```
postgres=# \d+ pg_buffercache
                        View "public.pg_buffercache"
      Column      |   Type   | Collation | Nullable | Default | Storage | Description
------------------+----------+-----------+----------+---------+---------+-------------
 bufferid         | integer  |           |          |         | plain   |
 relfilenode      | oid      |           |          |         | plain   |
 reltablespace    | oid      |           |          |         | plain   |
 reldatabase      | oid      |           |          |         | plain   |
 relforknumber    | smallint |           |          |         | plain   |
 relblocknumber   | bigint   |           |          |         | plain   |
 isdirty          | boolean  |           |          |         | plain   |
 usagecount       | smallint |           |          |         | plain   |
 pinning_backends | integer  |           |          |         | plain   |
View definition:
 SELECT bufferid,
    relfilenode,
    reltablespace,
    reldatabase,
    relforknumber,
    relblocknumber,
    isdirty,
    usagecount,
    pinning_backends
   from pg_buffercache_pages() p(bufferid integer, relfilenode oid, reltablespace oid, reldatabase oid,
relforknumber smallint, relblocknumber bigint, isdirty boolean, usagecount smallint, pinning_backends integer);
```

bufferid - порядковый номер буфера, начиная с 1
Следующие 5 столбцов соответствуют структуре BufferTag, в которой указан прямой (самодостаточный, то есть хранящий всё чтобы найти файл и в нём блок) адрес блока на диске:

reltablespace oid табличного пространства - название файла символической ссылки в PGDATA/pg_tblspc)
reldatabase oid базы данных - название поддиректории
relfilenode oid название файла relation. Названия файлов состоят из чисел
relforknumber номер форка (0 - main, 1- fsm, 2 - vm, 4 - init)
relblocknumber номер блока относительно нулевого блока первого файла форка, максимум 4294967294 (0xFFFFFFFE).
В представлении отражаются признаки: закрепления, менялся ли блок с момента чтения с диска (isdirty), число процессов которые закрепили (pin) блок.

4) Как узнать сколько блоков в буферном кэше относится к таблицам и индексам? Выполните команду:

```
postgres=# select relfilenode, count(*) from pg_buffercache group by relfilenode
order by 2 desc limit 5;
 relfilenode | count
-------------+-------
             | 15055
     1255891 |  2802
        1249 |   116
        1259 |    60
     1255486 |  1698
(5 rows)
```

Незанятых (свободных) блоков 15055. Имя таблицы в **pg_buffercache** не отражается, только номер файла.

5) Для получения имени relation нужен более сложный запрос. Выполните команду:

```
postgres=# SELECT n.nspname, c.relname, count(*) AS buffers, count(b.isdirty)
FILTER (WHERE b.isdirty = true) as buffer_dirty
 from pg_buffercache b JOIN pg_class c
 ON b.relfilenode = pg_relation_filenode(c.oid) AND
    b.reldatabase IN (0, (SELECT oid FROM pg_database WHERE datname =
current_database()))
 JOIN pg_namespace n ON n.oid = c.relnamespace
 GROUP BY n.nspname, c.relname
 ORDER BY 3 DESC
 LIMIT 2;
  nspname   |     relname      | buffers | buffer_dirty
------------+------------------+---------+--------------
 pg_catalog | pg_class         |    2802 |            3
 public     | pgbench_accounts |    1698 |            0
(2 rows)
```

Этот запрос удобнее. В запросе не выводятся строки, относящиеся к объектам других баз данных. База данных с номером 0 - глобальные объекты.

6) Еще более информативный запрос:

```
postgres=# select relname, buffers, relpages pages, (buffers*8)/1024 as size_mb,
 usage_count, dirty, pins,
 round(((100*dirty::float8)/nullif(buffers,0)::float8)::numeric,2) as "dirty%",
 round(((100*buffers::float8)/nullif(relpages,0)::float8)::numeric,2) as "cached%"
from
 (select c.relname, count(*) as buffers, sum(usagecount) as usage_count,
   count(*) filter (where isdirty) as dirty, sum(pinning_backends) as pins,
   max(c.relpages) as relpages
  from pg_buffercache b join pg_class c on b.relfilenode =
   pg_relation_filenode(c.oid)
   and b.reldatabase IN (0, (select oid
    from pg_database where datname = current_database()))
group by c.relname)
```

```
order by buffers desc
limit 2;

     relname      | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
------------------+---------+-------+---------+-------+-------+------+--------+---------
 pg_class         |    2802 |  2798 |      21 | 14006 |     3 |    1 |   0.11 |  100.14
 pgbench_accounts |    1698 |  1695 |      13 |  8490 |     0 |    0 |   0.00 |  100.18
(2 rows)
```

Запрос определяет размер relation по статистике, которая может отсутствовать. В этом случае в столбце cached% будет выдано пустое значение, а в pages ноль.
buffers > pages, так как кэшируются блоки слоёв fsm и vm, а также в случаях неактуальной статистики.

7) Запрос большой. Создайте для удобства представление:

```
postgres=# create or replace view buffercache as
select relname, buffers, relpages pages, (buffers*8)/1024 as size_mb,
 usage_count, dirty, pins,
 round(((100*dirty::float8)/nullif(buffers,0)::float8)::numeric,2) as "dirty%",
 round(((100*buffers::float8)/nullif(relpages,0)::float8)::numeric,2) as
"cached%"
from
 (select c.relname, count(*) as buffers, sum(usagecount) as usage_count,
   count(*) filter (where isdirty) as dirty, sum(pinning_backends) as pins,
   max(c.relpages) as relpages
  from pg_buffercache b join pg_class c on b.relfilenode =
   pg_relation_filenode(c.oid)
   and b.reldatabase IN (0, (select oid
    from pg_database where datname = current_database()))
group by c.relname)
order by buffers desc;
select * from buffercache limit 2;
CREATE VIEW
     relname      | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
------------------+---------+-------+---------+-------+-------+------+--------+---------
 pg_class         |    2802 |  2798 |      21 | 14006 |     3 |    1 |   0.11 |  100.14
 pgbench_accounts |    1698 |  1695 |      13 |  8490 |     0 |    0 |   0.00 |  100.18
(2 rows)
```

Представление удобно для просмотра содержимого буферного кэша. Однако, для просмотра используются обращения к заголовкам буферов, что на буферном кэше в сотни гигабайт создаёт небольшую дополнительную нагрузку. Запрос будет медленно работать, не по причине числа буферов, а по причине конкуренции с другими процессами за доступ к заголовку буфера. Выполняя запрос к представлению **pg_buffercache** (по грязным буферам) можно оценить не будет ли процесс контрольной точки сталкиваться с задержками при получении доступа к буферам.
%cache доля кэша, занимаемая блоками объекта.

8) Для более быстрого получения статистики о кэше буферов можно использовать запросы:

```
postgres=# select * from pg_buffercache_summary();
 buffers_used | buffers_unused | buffers_dirty | buffers_pinned |  usagecount_avg
--------------+----------------+---------------+----------------+------------------
         1392 |          14992 |             0 |              0 | 4.586206896551724
(1 row)

postgres=# select * buffercache_usage_counts();
 usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
           0 |   14992 |     0 |      0
           1 |      65 |     0 |      0
           2 |      90 |     0 |      0
           3 |      11 |     0 |      0
```

```
            4 |       20 |      0 |       0
            5 |     1206 |      0 |       0
(6 rows)
```

Если данных в этих представлениях достаточно, лучше использовать их. В буфера с `usage_count=0` входят как свободные блоки (находящиеся в списке свободных), так и содержащие блок, который давно не использовался.

Часть 2. Буферные кольца

1) Выполните команды:

```
postgres=# drop table if exists t;
create table t (id integer, val text storage plain not null) with (autovacuum_enabled =
off);
insert into t select i, 'text number ' || i from generate_series(1, 600000) as i;
select * from pg_size_pretty(pg_relation_size('t'));

drop table if exists t1;
create table t1 (id integer, val text storage plain not null) with (autovacuum_enabled =
off);
insert into t1 select i, 'text number ' || i from generate_series(1, 700000) as i;
select * from pg_size_pretty(pg_relation_size('t1'));

select * buffercache_usage_counts();
select * from buffercache limit 4;
```

```
DROP TABLE
CREATE TABLE
ALTER TABLE
INSERT 0 600000
 pg_size_pretty
----------------
 30 MB
(1 row)

DROP TABLE
CREATE TABLE
ALTER TABLE
INSERT 0 700000
 pg_size_pretty
----------------
 35 MB
(1 row)


 usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
           0 |    6665 |     0 |      0
           1 |     134 |     5 |      0
           2 |      57 |     4 |      0
           3 |      46 |     1 |      0
           4 |     763 |    10 |      0
           5 |    8719 |  8365 |      0
(6 rows)

     relname      | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
------------------+---------+-------+---------+-------+-------+------+--------+---------
 t1               |    4492 |     0 |      35 | 22456 |  4490 |    0 |  99.96 |
 t                |    3850 |     0 |      30 | 19246 |  3848 |    0 |  99.95 |
 pg_class         |    2802 |  2798 |      21 | 14008 |     2 |    1 |   0.07 |  100.14
 pgbench_accounts |    1698 |  1695 |      13 |  8490 |     0 |    0 |   0.00 |  100.18
(4 rows)
```

Команды создают две таблицы. Таблица **t** меньше 1/4 буферного кэша. Таблица **t1** больше 1/4 буферного кэша.

В столбце `cached%` пустое значение, а в столбце `pages` ноль потому, что число строк представление определяет по статистике, а она не собиралась после вставки строк. Статистика собирается командой ANALYZE.

Блоки всех таблиц загружены в буферный кэш и они все грязные, кроме пары блоков. Эта пара блоков относится к форку fsm. Пример:

```
postgres=# select bufferid, relforknumber, relblocknumber, usagecount, pinning_backends from pg_buffercache
where relfilenode=3243243 and isdirty=false;
 bufferid | relforknumber | relblocknumber | usagecount | pinning_backends
----------+---------------+----------------+------------+------------------
     9754 |             1 |              1 |          1 |                0
     9755 |             1 |              0 |          5 |                0
(2 rows)
```

Почему при работе с таблицей **t1**, которая больше 1/4 буферного кэша не использовалось буферное кольцо? Потому что не было условий использования буферного кольца.

Условия использования буферных колец (алгоритм ещё называется "buffer cache replacement strategy" и "Buffer Access Strategy Type"):

a) BULKREAD. Для последовательного чтения (**методом Seq Scan**) блоков **таблиц размер которых больше или равен 1/4 кэша буферов**. Размер кольца 256Кб. Если таблица (размером не меньше 1/4) уже читается в другой сессии, то сессии желающие прочесть блоки таблицы используют одно буферное кольцо. Можно сказать, что **сессии "синхронизируются"**, чтобы повторно не читать те же самые блоки. Остальные сессии начинают чтение не с нулевого блока, а с того который был в буферном кольце первой сессии и потом дочитывают блоки с начала. Синхронизацию можно отключить параметром конфигурации `synchronize_seqscans`, но не стоит этого делать.

b) VACUUM. Грязные страницы не убираются из кольца, а посылаются на запись. Размер кольца задается параметром конфигурации vacuum_buffer_usage_limit. По умолчанию размер кольца 256Кб. Автовакуум также использует буферное кольцо (в агрессивном режиме автовакуума буферное кольцо не используется).

c) BULKWRITE. **Всегда используется командами COPY и CREATE TABLE AS SELECT независимо от объема данных и размеров таблиц**. Размер кольца 16Мб.

Если буфер стал грязным, он исключается из буферного кольца.

В буферном кэше блок может находиться только в одном буфере.

Буфера таблиц имеют **usage_count=5**. Столбец `usage_count` представления **buffercache** неинформативен, он не показывает сколько раз обращались к буферу, это сумма **sum(usagecount)** по каждому блоку, а **usagecount** ограничен числом 5.

2) Выполните команды:

```
postgres=# drop table if exists t2;
create table t2 as select * from t1;
select * from pg_size_pretty(pg_relation_size('t2'));
select * buffercache_usage_counts();
select * from buffercache limit 4;
DROP TABLE
SELECT 700000
 pg_size_pretty
----------------
 36 MB
(1 row)

 usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
           0 |     132 |     0 |      0
           1 |    2125 |  1998 |      0
           2 |      31 |     1 |      0
           3 |    5672 |     8 |      0
           4 |    8416 |  4777 |      0
           5 |       8 |     6 |      0
(6 rows)

 relname  | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
----------+---------+-------+---------+-------+-------+------+--------+---------
```

```
 t1       |    4492 |      0 |       35 | 17965 |  4490 |     0 |  99.96 |
 t        |    3850 |      0 |       30 | 15396 |   271 |     0 |   7.04 |
 pg_class |    2802 |   2798 |       21 |  8410 |     5 |     1 |   0.18 |  100.14
 t2       |    2048 |      0 |       16 |  2048 |  1992 |     0 |  97.27 |
(4 rows)
```

При создании таблицы **t2** использовалось буферное кольцо. Размер кольца 16Мб=2048 блоков. Из них грязных блоков **1992**. Остальные блоки были посланы на запись и вытеснены (evicted) из буферного кольца (и соответственно буферного кэша). Буферные кольца это набор блоков в буферном кэше, а не отдельная структура памяти.

3) Выполните команды:

```
postgres=# VACUUM(ANALYZE) t1;
VACUUM(ANALYZE, BUFFER_USAGE_LIMIT 0) t2;
select * from buffercache limit 4;
VACUUM
VACUUM
 relname  | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
----------+---------+-------+---------+-------+-------+------+--------+---------
 t1       |    4492 |  4488 |      35 |     4 |  4406 |    0 |  98.09 |  100.09
 t        |    3849 |     0 |      30 |     0 |     0 |    0 |   0.00 |
 pg_class |    2800 |  2798 |      21 |  8393 |     4 |    1 |   0.14 |  100.07
 t2       |    2321 |  4488 |      18 |  2117 |     5 |    0 |   0.22 |   51.72
(4 rows)
```

Параметр BUFFER_USAGE_LIMIT устанавливает размер буферного кольца вместо параметра конфигурации vacuum_buffer_usage_limit. В отличие от параметра конфигурации, BUFFER_USAGE_LIMIT можно установить в значение ноль и буферное кольцо не будет использоваться. Значение параметра по умолчанию:

```
postgres=# \dconfig *usage*
 List of configuration parameters
         Parameter          | Value
----------------------------+-------
 vacuum_buffer_usage_limit | 256kB
(1 row)
```

4) Выполните команды:

```
postgres=# copy t1 to '/tmp/t1';
truncate table t2;
select * from buffercache limit 4;
begin;
copy t2 from '/tmp/t1' with FREEZE;
end;
COPY 700000
TRUNCATE TABLE
     relname      | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
------------------+---------+-------+---------+-------+-------+------+--------+---------
 t1               |    4492 |  4488 |      35 |  4491 |     0 |    0 |   0.00 |  100.09
 t                |    3849 |     0 |      30 |     0 |     0 |    0 |   0.00 |
 pg_class         |    2800 |  2798 |      21 | 13990 |     1 |    1 |   0.04 |  100.07
 pgbench_accounts |    1698 |  1695 |      13 |     0 |     0 |    0 |   0.00 |  100.18
(4 rows)

BEGIN
ERROR:  cannot perform COPY FREEZE because the table was not created or truncated in the
current subtransaction
ROLLBACK
```

Команда COPY с опцией FREEZE может выполняться только в транзакции, в которой таблица была создана или усечена.

С точки зрения производительности использование опции FREEZE полезно тем, что строки замораживаются и автовакуум не будет делать через какое-то время лишний проход по всем блокам таблицы. При загрузке строки в TOAST не замораживаются (https://github.com/postgres/postgres/commit/8e03eb92).

5) Выполните команды:

```
postgres=# begin;
truncate t2;
copy t2 from '/tmp/t1' with FREEZE;
commit;
select * from buffercache limit 4;
BEGIN
COPY 700000
COMMIT
 relname   | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
-----------+---------+-------+---------+-------+-------+------+--------+---------
 t1        |    4493 |  4488 |      35 |  4499 |     0 |    0 |   0.00 |  100.11
 t         |    3849 |     0 |      30 |     0 |     0 |    0 |   0.00 |
 pg_class  |    2799 |  2798 |      21 | 13990 |     1 |    1 |   0.04 |  100.04
 t2        |    2049 |     0 |      16 |  2053 |  2009 |    0 |  98.05 |
(4 rows)
```

Команда COPY с опцией FREEZE была выполнена в одной транзакции с командой усечения таблицы. При загрузке использовалось буферное кольцо. В буферном кэше блоки таблицы используют 2049 буферов, а в таблице 4493 блока - столько же, сколько в таблице t1.

6) Проверьте используется ли буферное кольцо BULKWRITE, если размер таблицы меньше 1/4 буферного кэша. Сохраните строки таблицы t, размер которой меньше 1/4 буферного кэша и загрузите их в таблицы. Выполните команды:

```
postgres=# copy t to '/tmp/t1';
truncate table t;
copy t from '/tmp/t1';
begin;
truncate table t2;
copy t2 from '/tmp/t1';
commit;
select * from buffercache limit 4;
COPY 600000
TRUNCATE TABLE
COPY 600000
BEGIN
TRUNCATE TABLE
COPY 600000
COMMIT
 relname   | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
-----------+---------+-------+---------+-------+-------+------+--------+---------
 t1        |    4493 |  4488 |      35 |  4499 |     0 |    0 |   0.00 |  100.11
 pg_class  |    2799 |  2798 |      21 | 13990 |     1 |    1 |   0.04 |  100.04
 t         |    2052 |     0 |      16 |  2068 |  2023 |    0 |  98.59 |
 t2        |    2048 |     0 |      16 |  2048 |  2007 |    0 |  98.00 |
(4 rows)
```

Буферное кольцо BULKWRITE используется и не зависит ни от размера загружаемых данных, ни от размера таблиц. Всегда используется командами COPY и CREATE TABLE AS SELECT.

Часть 3. Расширение pg_prewarm

1) Установите расширение pg_prewarm:

```
postgres=# alter system set shared_preload_libraries='pg_prewarm';
ALTER SYSTEM
postgres=# \q
postgres@tantor:~$ sudo restart
postgres@tantor:~$ psql
postgres=# create extension pg_prewarm;
CREATE EXTENSION
```

2) Посмотрите какие объекты входят в расширение pg_prewarm:

```
postgres=# \dx+ pg_prewarm
           Objects in extension "pg_prewarm"
                 Object description
------------------------------------------------------
 function autoprewarm_dump_now()
 function autoprewarm_start_worker()
 function pg_prewarm(regclass,text,text,bigint,bigint)
(3 rows)
```

3) Расширение имеет параметры конфигурации:

```
postgres=# \dconfig *prewarm*
    List of configuration parameters
            Parameter             | Value
----------------------------------+-------
 pg_prewarm.autoprewarm           | on
 pg_prewarm.autoprewarm_interval  | 5min
(2 rows)
```

После рестарта экземпляра запускается фоновый процесс `autoprewarm leader`, который пока работает экземпляр или при остановке экземпляра сохраняет в фоновом режиме данные в файл autoprewarm.blocks находящийся в корне директории кластера PGDATA. При рестарте экземпляра процесс `leader` сортирует список блоков из файла и запускает `bgworker`, который последовательно подключается к базам и загружает блоки объектов баз в кэш буферов.

Параметром `pg_prewarm.autoprewarm` можно отключить запуск процесса leader. Изменение этого параметра требует перезапуск экземпляра. Запустить или остановить фоновый процесс пока экземпляр работает нельзя, функция `autoprewarm_start_worker()` бесполезна. Частота сохранения данных в файл устанавливается параметром `pg_prewarm.autoprewarm_interval`. По умолчанию 5 минут. Если установить ноль, то файл со списком блоков не будет обновляться.

4) Функция расширения `pg_prewarm(regclass,text,text,bigint,bigint)` позволяет загрузить в кэш буферов блоки объекта:

```
postgres=# select pg_prewarm('pg_class', 'prefetch');
 pg_prewarm
------------
        758
(1 row)
```

При вызове функции достаточно указать один параметр с именем объекта: `pg_prewarm('pg_class')`.

5) Второй параметр функции `pg_prewarm(regclass,text,text,bigint,bigint)` называется mode:

```
postgres=# \sf pg_prewarm
CREATE OR REPLACE FUNCTION public.pg_prewarm(regclass, mode text DEFAULT 'buffer'::text,
fork text DEFAULT 'main'::text, first_block bigint DEFAULT NULL::bigint, last_block
bigint DEFAULT NULL::bigint)
 RETURNS bigint
 LANGUAGE c
 PARALLEL SAFE
AS '$libdir/pg_prewarm', $function$pg_prewarm$function$
```

Параметром `mode text DEFAULT 'buffer'::text` можно указать способ загрузки блоков:
`buffer` (значение по умолчанию) в буферный кэш
`read` только в страничный кэш синхронным вызовом
`prefetch` только в страничный кэш вызовом posix_fadvise(.., POSIX_FADV_WILLNEDED))

Если буферный кэш пуст, то загрузка произойдёт быстро. Если буферный кэш заполнен, то придется выгружать блоки других объектов (eviction) и на их место загружать новые блоки.
Чтобы этого избежать можно использовать `mode=prefetch`. Блоки загрузятся в кэш операционной системы и если к блокам нужен будет доступ, то блоки быстро скопируются в буферный кэш. В файле `autoprewarm.blocks` сохраняются только блоки, загруженнные в буферный кэш. Информация о загрузке блоков в кэш операционной системы нигде не сохраняется.

6) Посмотрите начало файла расширения:

```
postgres=# \! head -3 $PGDATA/autoprewarm.blocks
<<1269>>
0,1664,1255633,0,0
0,1664,1255462,0,0
```

В первой строке файла число блоков, адреса которых сохранены в файле. Адрес блока находится в формате стандартной структуры `BufferTag`. Формат структуры позволяет найти блок на диске не обращаясь ни к каким дополнительным данным. В структуре 5 полей, соответствующих столбцам в файле расширения:
`oid` табличного пространства - название файла символической ссылки в PGDATA/pg_tblspc)
`oid` базы данных - название поддиректории
название файла relation
номер форка (`0 - main, 1- fsm, 2 - vm, 4 - init`)
номер блока относительно нулевого блока первого файла форка, максимум 4294967294 (0xFFFFFFFE).

7) Удалите файл:

```
postgres=# \! rm $PGDATA/autoprewarm.blocks
postgres=# \! head -3 $PGDATA/autoprewarm.blocks
head: cannot open '/var/lib/postgresql/tantor-se-16/data/autoprewarm.blocks' for
reading: No such file or directory
```

8) Вызовите функцию создающую или обновляющую файл:

```
postgres=# select autoprewarm_dump_now();
 autoprewarm_dump_now
----------------------
                 1269
(1 row)
```

Функция обновляет или создает файл autoprewarm.blocks. Это может быть полезно, если процесс leader не запущен, но нужно, чтобы после перезапуска экземпляра блоки были

подгружены в буферный кэш. Функция возвращает количество блоков, имеющихся в файле autoprewarm.blocks.

9) Функция вернула число блоков в файле:

```
postgres=# \! head -3 $PGDATA/autoprewarm.blocks
<<1269>>
0,1664,1255633,0,0
0,1664,1255462,0,0
```

При рестарте экземпляра 1269 блоков будут загружаться в фоновом режиме в буферный кэш.

Часть 4. Процесс фоновой записи bgwriter

Процесс фоновой записи посылает содержимое буфера на запись (но не вытесняет из буферного кэша), если в заголовке буфера есть признаки что буфер:
a) грязный
b) блок не повреждён
c) refcount=0, закреплений блока нет, то есть блок не нужен процессам
d) usage_count=0 (попадает в градацию давно не использовавшихся).
После посылки содержимого буфера на запись, буфер не попадает в список свободных, буфер становится чистым (снимается признак того, что буфер грязный).

1) Посмотрите сколько блоков может заранее (read-ahead) прочесть контроллер с блочного устройства, где лежит директория кластера:

```
postgres@tantor:~$ sudo blockdev --getra /dev/sda
256
```

Контроллер может прочесть 256 блоков. Единица измерения 512 байт. Контроллер может заранее прочесть до 128Кб.
Вакуум, анализ, процесс startup используют предварительное чтение блоков (prefetch). Расширение pg_prewarm может использовать prefetch. Максимальное число буферов устанавливается параметром конфигурации maintenance_io_concurrency.

2) Посмотрите значение параметра `maintenance_io_concurrency`:

```
postgres=# show maintenance_io_concurrency;
 maintenance_io_concurrency
----------------------------
 10
(1 row)
```

Значение по умолчанию 10 блоков по 8Кб, что соответствует 80Кб.
Первый и второй пункт практики нужны, чтобы вы обратили внимание на эти параметры, которые описываются в теоретической части.

4) Посмотрите параметры конфигурации bgwriter:

```
postgres=# select name, setting, context, max_val, min_val from pg_settings
where name ~ 'bgwr';
          name           | setting | context |  max_val   | min_val
-------------------------+---------+---------+------------+---------
 bgwriter_delay          | 200     | sighup  | 10000      | 10
 bgwriter_flush_after    | 64      | sighup  | 256        | 0
 bgwriter_lru_maxpages   | 100     | sighup  | 1073741823 | 0
```

```
 bgwriter_lru_multiplier | 2       | sighup  | 10          | 0
(4 rows)
```

Это удобный запрос, чтобы просматривать минимальное, максимальное значение и другие детали по конфигурационным параметрам. Команды `show` и `\dconfig` этих деталей не выдают.

`bgwriter_delay` на сколько миллисекунд bgwriter засыпает между итерациями.
`bgwriter_flush_after` - число блоков после посылки на запись которых инициируется flush страничного кэша linux. Ноль отключает flush.
Число грязных буферов, записываемых в итерации, зависит от того, сколько блоков подгрузили в буферный кэш серверные процессы (recently allocated) в предыдущих циклах. Усредненное значение умножается на `bgwriter_lru_multiplier` и указывает сколько буферов нужно очистить в текущем цикле. Процесс с максимальной скоростью пытается достичь это значение, но не больше, чем `bgwriter_lru_maxpages`. `bgwriter_lru_maxpages` - максимальное количество блоков, которые записываются в одной итерации, при нулевом значении bgwriter перестает работать.

5) Выполните запрос:

```
postgres=# select * from pg_stat_bgwriter\gx
-[ RECORD 1 ]---------+------------------------------
checkpoints_timed     | 707
checkpoints_req       | 173
checkpoint_write_time | 11877275
checkpoint_sync_time  | 3842836
buffers_checkpoint    | 1418992
buffers_clean         | 88732
maxwritten_clean      | 310
buffers_backend       | 8050175
buffers_backend_fsync | 0
buffers_alloc         | 3158495
stats_reset           | 2035-01-01 01:01:59.220537+03
```

Отражает статистику эффективности работы bgwriter по всему экземпляру. Представление содержит одну строку.
`buffers_clean` - сколько буферов послал на запись ("clean", "очистил") bgwriter.
`maxwritten_clean` - сколько раз bgwriter приостанавливал работу из-за того, что достиг bgwriter_lru_maxpages или расчетного значения количества блоков (с учетом bgwriter_lru_multiplier), если bgwriter_lru_maxpages уже установлен в максимальное значение.
`stats_reset` дата и время обнуления (reset) данных в этом представлении. Обнулить данные в представлении можно функцией **`pg_stat_reset_shared('bgwriter')`**. Вызов функции без параметров `pg_stat_reset()` не обнуляет статистику ни в этом представлении, ни в представлении `pg_stat_io`.
`buffers_backend` - сколько буферов очистили (послали на запись) серверные процессы и автовакуум.

В 17 версии в представлении мало столбцов:

```
postgres=# \! sudo docker exec -it postgres psql -U postgres -c "select * from pg_stat_bgwriter"
 buffers_clean | maxwritten_clean | buffers_alloc |         stats_reset
---------------+-----------------+---------------+------------------------------
             0 |               0 |           176 | 2025-01-03 21:27:38.13624+00
(1 row)
```

Часть столбцов была перенесена в представление `pg_stat_checkpointer`:

```
postgres=# \! sudo docker exec -it postgres psql -U postgres -c "select * from pg_stat_checkpointer"
 num_timed | num_requested | restartpoints_timed | restartpoints_req | restartpoints_done | write_time | sync_time | buffers_written | stats_reset
-----------+---------------+---------------------+-------------------+--------------------+------------+-----------+-----------------+-------------------------------
       785 |             2 |                   0 |                 0 |                  0 |         35 |         9 |               0 | 2035-01-01 01:01:38.13624+00
(1 row)
```

Значения, аналогичные тому, что выводится в столбце `buffers_backend` в 17 версии предлагается смотреть в представлении `pg_stat_io`.

6) Удобно просматривать показатели не в абсолютных значениях, а в процентах:

```
postgres=# create view pgstatbgwriter as select to_char(100*checkpoints_timed::numeric / nullif((checkpoints_timed+checkpoints_req),0),'990D9')
  || ' %' "ckpt by time",
 to_char(100*checkpoints_req::numeric / nullif((checkpoints_timed+checkpoints_req),0),'990D9')
  || ' %' AS "ckpt by size",
 to_char(100*buffers_checkpoint::numeric / nullif((buffers_checkpoint+buffers_clean + buffers_backend),0),'990D9')
  || ' %' "checkpointer",
 to_char(100*buffers_backend::numeric / nullif((buffers_checkpoint+buffers_clean + buffers_backend),0),'990D9')
  || ' %' "backend",
 to_char(100*buffers_backend_fsync::numeric / nullif((buffers_checkpoint+buffers_clean + buffers_backend),0),'990D9')
  || ' %' "backend_fsync",
 to_char(100*buffers_clean::numeric / nullif((buffers_checkpoint+buffers_clean + buffers_backend),0),'990D9')
  || ' %' "bgwriter",
 pg_size_pretty((buffers_checkpoint+buffers_clean+buffers_backend)*8192/ (extract (epoch from current_timestamp - stats_reset))::bigint) || ' / s' "speed"
 FROM pg_stat_bgwriter;
 select * from pgstatbgwriter;
 CREATE VIEW
  ckpt by time | ckpt by size | checkpointer | backend  | backend_fsync | bgwriter |   speed
 --------------+--------------+--------------+----------+---------------+----------+------------
    80.5 %    |   19.5 %     |   14.8 %     |   84.2 % |    0.0 %      |    0.9 % | 338 kB / s
 (1 row)
```

7) Ещё один удобный запрос для 16 версии:

```
SELECT
  clock_timestamp()-pg_postmaster_start_time()                    "Uptime",
  clock_timestamp()-stats_reset                                   "Since stats reset",
  round(100.0*checkpoints_req/total_checkpoints,1)                "Forced checkpoint ratio (%)",
  round(np.min_since_reset/total_checkpoints,2)                   "Minutes between checkpoints",
  round(checkpoint_write_time::numeric/(total_checkpoints*1000),2) "Average write time per checkpoint (s)",
  round(checkpoint_sync_time::numeric/(total_checkpoints*1000),2)  "Average sync time per checkpoint (s)",
  round(total_buffers/np.mp,1)                                    "Total MB written",
  round(buffers_checkpoint/(np.mp*total_checkpoints),2)           "MB per checkpoint",
  round(buffers_checkpoint/(np.mp*np.min_since_reset*60),2)       "Checkpoint MBps",
  round(buffers_clean/(np.mp*np.min_since_reset*60),2)            "Bgwriter MBps",
  round(buffers_backend/(np.mp*np.min_since_reset*60),2)          "Backend MBps",
  round(total_buffers/(np.mp*np.min_since_reset*60),2)            "Total MBps",
  round(1.0*buffers_alloc/total_buffers,3)                        "New buffer allocation ratio",
  round(100.0*buffers_checkpoint/total_buffers,1)                 "Clean by checkpoints (%)",
  round(100.0*buffers_clean/total_buffers,1)                      "Clean by bgwriter (%)",
  round(100.0*buffers_backend/total_buffers,1)                    "Clean by backends (%)",
  round(100.0*maxwritten_clean/(np.min_since_reset*60000/np.bgwr_delay),2) "Bgwriter halt-only length (buffers)",
  coalesce(round(100.0*maxwritten_clean/(nullif(buffers_clean,0)/np.bgwr_maxp),2),0) "Bgwriter halt ratio (%)",
  '------------------'  "--------------------------------",
  bgstats.*
   FROM (
     SELECT bg.*,
  checkpoints_timed + checkpoints_req total_checkpoints,
  buffers_checkpoint + buffers_clean + buffers_backend total_buffers,
  pg_postmaster_start_time() startup,
  current_setting('checkpoint_timeout') checkpoint_timeout,
  current_setting('max_wal_size') max_wal_size,
  current_setting('checkpoint_completion_target') checkpoint_completion_target,
```

```
 current_setting('bgwriter_delay') bgwriter_delay,
 current_setting('bgwriter_lru_maxpages') bgwriter_lru_maxpages,
 current_setting('bgwriter_lru_multiplier') bgwriter_lru_multiplier
    FROM pg_stat_bgwriter bg
 ) bgstats,
 (
    SELECT
 round(extract('epoch' from clock_timestamp() - stats_reset)/60)::numeric min_since_reset,
 (1024 * 1024 / block.setting::numeric) mp,
 delay.setting::numeric bgwr_delay,
 lru.setting::numeric bgwr_maxp
    FROM pg_stat_bgwriter bg
    JOIN pg_settings lru   ON lru.name = 'bgwriter_lru_maxpages'
    JOIN pg_settings delay ON delay.name = 'bgwriter_delay'
    JOIN pg_settings block ON block.name = 'block_size'
 ) np\gx
-[ RECORD 1 ]-------------------------+-----------------------------
Uptime                                | 10:05:12.155806
Since stats reset                     | 2 days 14:55:23.335427
Forced checkpoint ratio (%)           | 19.5
Minutes between checkpoints           | 4.26
Average write time per checkpoint (s) | 13.39
Average sync time per checkpoint (s)  | 4.33
Total MB written                      | 74671.1
MB per checkpoint                     | 12.50
Checkpoint MBps                       | 0.05
Bgwriter MBps                         | 0.00
Backend MBps                          | 0.28
Total MBps                            | 0.33
New buffer allocation ratio           | 0.330
Clean by checkpoints (%)              | 14.8
Clean by bgwriter (%)                 | 0.9
Clean by backends (%)                 | 84.2
Bgwriter halt-only length (buffers)   | 0.03
Bgwriter halt ratio (%)               | 34.94
--------------------------------      | ------------------
checkpoints_timed                     | 714
checkpoints_req                       | 173
checkpoint_write_time                 | 11877275
checkpoint_sync_time                  | 3842836
buffers_checkpoint                    | 1418992
buffers_clean                         | 88732
maxwritten_clean                      | 310
buffers_backend                       | 8050175
buffers_backend_fsync                 | 0
buffers_alloc                         | 3158508
stats_reset                           | 2035-01-01 01:07:59.220537+03
total_checkpoints                     | 887
total_buffers                         | 9557899
startup                               | 2035-01-01 01:58:10.400156+03
checkpoint_timeout                    | 5min
max_wal_size                          | 1GB
checkpoint_completion_target          | 0.9
bgwriter_delay                        | 200ms
bgwriter_lru_maxpages                 | 100
bgwriter_lru_multiplier               | 2
```

8) Выполните запрос с помощью функции расширения `pg_buffercache`:

```
postgres=# select * buffercache_usage_counts();
 usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
           0 |    6294 |     0 |      0
           1 |    4243 |     0 |      0
           2 |      46 |     0 |      0
           3 |      37 |     0 |      0
           4 |      30 |     0 |      0
           5 |    5734 |     0 |      0
(6 rows)
```

Функция даёт моментальную картину того есть ли работа для bgwriter или он будет работать впустую, только обращаясь к буферам. bgwriter может очистить блоки, у которых одновременно по данным представления `pg_buffercache isdirty=true, usagecount=0, pinning_backends=0`. По данным функции **`pg_buffercache_usage_counts()`** такие блоки учитываются в первой строке (`usage_count=0`) - чем больше dirty и чем меньше pinned, тем

более верояно, что процессу bgwriter есть, что очищать. Если в столбце buffers в верхних строках нули, а большинство блоков в последних строках (`usage_count=5` и `usage_count=4`), то вероятность того, что bgwriter есть (будет в ближайшее время) что очищать невелика.

Это представление полезно тем, что в столбце `buffers` можно посмотреть равномерно ли распределение буферов по градациям `usage_count` при текущей активности и настройках экземпляра. Распределение буферов по шести "корзинкам" usage_count не должно иметь явного перекоса в одну или другую стороны. При перекосе алгоритм вытеснения теряет эффективность.

Если в столбце `buffers` есть перекос в сторону `usage_count=5` это означает, что размер буферного кэша недостаточен. Если большая часть буферов в строке с `usage_count=1` размер буферного кэша можно уменьшить. В строке `usage_count=0` учитываются свободные буфера. Свободные буфера присутствуют после рестарта экземпляра или удаления, усечения таблиц, баз данных. Если экземпляр работал долго, то буферный кэш заполняется полностью. Если имеется большое число `buffers` в строке `usage_count=0`, то либо экземпляр только что перегрузился и большая часть буферов свободна, либо размер буферного кэша можно уменьшить.

Настраивать параметры bgwriter лучше после настройки размера кэша буферов и настройки процесса контрольной точки.

Практика к главе 11

Часть 1. Настройка частоты контрольных точек

1) Посмотрите параметры конфигурации процесса checkpointer:

```
postgres=# select name, setting, unit, context, max_val, min_val from
pg_settings where name ~ 'checkpoint' or name='max_wal_size';
             name              | setting | unit | context |  max_val   | min_val
-------------------------------+---------+------+---------+------------+---------
 checkpoint_completion_target | 0.9     |      | sighup  | 1          | 0
 checkpoint_flush_after       | 32      | 8kB  | sighup  | 256        | 0
 checkpoint_timeout           | 300     | s    | sighup  | 86400      | 30
 checkpoint_warning           | 30      | s    | sighup  | 2147483647 | 0
 log_checkpoints              | on      |      | sighup  |            |
 max_wal_size                 | 1024    | MB   | sighup  | 2147483647 | 2
(6 rows)
```

По умолчанию `log_checkpoints=on` и в диагностический журнал записываются сообщения о выполнении контрольных точек.
checkpoint_timeout определяет интервалы между контрольными точками, но контрольная точка начнётся раньше, если объем WAL приблизится к **max_wal_size**.

2) Создайте скрипты для теста:

```
postgres@tantor:~$ mcedit ckpt.sql

\pset tuples_only
\getenv t3 t3
\echo :t3
\o ckpt.tmp
alter system set checkpoint_timeout = :t3;
select pg_reload_conf();
select pg_current_wal_lsn() AS t1 \gset
\! sleep $t3
select pg_current_wal_lsn() AS t2 \gset
\o
select pg_size_pretty((:'t2'::pg_lsn - :'t1'::pg_lsn)/:'t3');
select * buffercache_usage_counts();

postgres@tantor:~$ mcedit ckpt.sh

for (( i=60; i <= 300; i+=60 ))
do
export t3="$i"
psql -f ckpt.sql
done

postgres@tantor:~$ mcedit ckpt-horizon.sh

for (( i=60; i <= 300; i+=60 ))
do
export t3="$i"
psql -f ckpt-horizon.sql
done

postgres@tantor:~$ mcedit ckpt-horizon.sql

\pset tuples_only
```

```
\getenv t3 t3
\echo :t3
\o ckpt.tmp
alter system set checkpoint_timeout = :t3;
select pg_reload_conf();
select pg_current_wal_lsn() AS t1 \gset
select pg_sleep(:t3);
select pg_current_wal_lsn() AS t2 \gset
\o
select pg_size_pretty((:'t2'::pg_lsn - :'t1'::pg_lsn)/:'t3');
```

```
postgres@tantor:~$ chmod +x ckpt.sh
chmod +x ckpt-horizon.sh
```

3) В первом окне терминала запустите стандартный тест:

```
postgres@tantor:~$ pgbench -i
dropping old tables...
creating tables...
generating data (client-side)...
100000 of 100000 tuples (100%) done (elapsed 0.13 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.45 s (drop tables 0.03 s, create tables 0.02 s, client-side generate
0.19 s, vacuum 0.09 s, primary keys 0.11 s).
postgres@tantor:~$ pgbench -c 4 -j 10 -T 6000 -P 30
starting vacuum...end.
progress: 30.0 s, 220.2 tps, lat 18.154 ms stddev 6.414, 0 failed
progress: 60.0 s, 190.7 tps, lat 20.974 ms stddev 7.433, 0 failed
```

Время теста указано с запасом 6000 секунд. Когда нагрузка не будет нужна можно прервать тест комбинацией клавиш <ctrl+c>

4) Во втором окне терминала запустите тест без удержания горизонта.

```
postgres@tantor:~$ ./ckpt.sh
```

Пока тест работает посмотрите текст скрипта, чтобы понять, что делает теста. Тест увеличивает частоту контрольной точки с 30 секунд до 5 минут с интервалами по минуте. Цель теста - найти оптимальный интервал. Поскольку интервал контрольной точки по умолчанию 5 минут, рекомендуемый 20 минут, то реальный тест бы занял долгое время. При реальном тестировании можно сделать интервал 5-10 минут от 5 минут до 1-2 часов, а время нагрузки поставить заведомо долгое, например, 600000 секунд.

Для целей курса важны тенденция метрик (метрика убывает, растет, не меняется), методика тестирования, набор метрик.

Запуск и работа скрипта помогает лучше запомнить тему. Если не запускать скрипты, не набирать на клавиатуре команды, не получать ошибки и не задумываться как их исправить, то изучаемый материал не вызовет эмоций (интереса, радости от того что удалось запустить тест, расстройства из-за опечаток), благодаря которым и запоминается материал.

Скрипт меняет частоту контрольной точки, перечитывает конфигурацию, измеряет **сколько WAL-записей было создано** , выводит статистику по буферному кэшу.

При увеличении интервала между контрольными точками уменьшается объем WAL-записей. Это улучшает производительность.

В процессе выполнения теста можно, при желании, использовать запросы из предыдущей практики.

В окне терминала pgbench tps и latency не меняются:

```
progress: 210.0 s, 192.7 tps, lat 20.754 ms stddev 7.556, 0 failed
```

```
progress: 240.0 s, 193.7 tps, lat 20.653 ms stddev 7.712, 0 failed
progress: 270.0 s, 194.4 tps, lat 20.569 ms stddev 8.251, 0 failed
```

Если tps не меняются, то значит ли это что частота контрольных точек не влияет на производительность? Не не значит. В нагрузочном тесте короткие транзакции, почти все старые строки очищаются HOT cleanup.

Пример результатов теста, которые будут выдаваться в терминал:

```
postgres@tantor:~$ ./ckpt.sh &
[1] 137305
postgres@tantor:~$ Tuples only is on.
60
 339 kB

           0 |    9883 |      0 |      0
           1 |     350 |      0 |      0
           2 |      88 |      0 |      0
           3 |      36 |      0 |      0
           4 |      30 |      1 |      0
           5 |    5997 |   1753 |      2

Tuples only is on.
120
 234 kB

           0 |    9727 |      0 |      0
           1 |     348 |      0 |      0
           2 |      88 |      0 |      0
           3 |      35 |      0 |      0
           4 |      23 |      0 |      0
           5 |    6163 |   2014 |      2

Tuples only is on.
180
 192 kB

           0 |    9471 |      0 |      0
           1 |     348 |      0 |      0
           2 |      86 |      0 |      0
           3 |      37 |      1 |      0
           4 |      22 |      0 |      0
           5 |    6420 |   2186 |      1

Tuples only is on.
240
 167 kB

           0 |    9141 |      0 |      0
           1 |     371 |      0 |      0
           2 |      86 |      0 |      0
           3 |      34 |      0 |      0
           4 |      24 |      0 |      0
           5 |    6728 |   2254 |      2

Tuples only is on.
300
 158 kB

           0 |    8753 |      0 |      0
           1 |     371 |      0 |      0
           2 |      88 |      2 |      0
           3 |      34 |      0 |      0
           4 |      22 |      0 |      0
           5 |    7116 |   2355 |      1

Tuples only is on.
360
 150 kB

           0 |    8281 |      0 |      0
           1 |     371 |      0 |      0
           2 |      87 |      0 |      0
           3 |      34 |      0 |      0
           4 |      22 |      0 |      0
           5 |    7589 |   2462 |      2

Tuples only is on.
420
 154 kB
```

```
  0 |    7735 |      0 |       0
  1 |     371 |      0 |       0
  2 |      87 |      0 |       0
  3 |      34 |      0 |       0
  4 |      22 |      0 |       0
  5 |    8135 |   2584 |       1

Tuples only is on.
480
```

Распределение блоков интереса не представляет: тест простейший и блоки скапливаются в usage_count=5. Понемногу уменьшается число свободных блоков (usage_count=0). Можно сделать вывод, что размер буферного кэша (128Мб) большой для нагрузки (стандартным тестом pgbench).

Интересно то, что при увеличении интервала между контрольными точками уменьшается объем записи в WAL.

Когда закончите знакомиться с текстом скрипта, задания, результатами теста, прервите выполнение теста:

```
postgres@tantor:~$ kill -9 137305
postgres@tantor:~$ ps
    PID TTY          TIME CMD
 137300 pts/1    00:00:00 bash
 137593 pts/1    00:00:00 psql
 137595 pts/1    00:00:00 sleep
 137607 pts/1    00:00:00 ps
[1]+  Killed                  ./ckpt.sh
postgres@tantor:~$ <ctrl+d>
logout
astra@tantor:~$ su - postgres
Password: postgres
postgres@tantor:~$
```

5) Во втором окне терминала запустите скрипт с удержанием горизонта.

```
postgres@tantor:~$ ./ckpt-horizon.sh &
[2] 138802
postgres@tantor:~$ Tuples only is on.
60
```

Горизонт удерживается на каждой итерации цикла и длительность удержания до перечитывания конфигурации. То есть длительность удержания горизонта равна текущему интервалу между контрольноыми точками.

6) В третьем окне терминала можно запустить psql и проверить, что горизонт удерживается:

```
postgres=# select age(backend_xmin), extract(epoch from (clock_timestamp()-
xact_start)) secs, pid, datname database, state from pg_stat_activity where
backend_xmin IS NOT NULL OR backend_xid IS NOT NULL order by
greatest(age(backend_xmin), age(backend_xid)) desc;
  age  |    secs    |  pid   | database | state
-------+------------+--------+----------+--------
 59898 | 338.774153 | 138907 | postgres | active
  8698 |  50.358268 | 138953 | postgres | active
     6 |   0.004147 | 137282 | postgres | active
     6 |   0.007396 | 137284 | postgres | active
     6 |   0.022296 | 137283 | postgres | active
     6 |   0.001173 | 138773 | postgres | active
(6 rows)
```

В примере горизонт удерживается **338** секунд.

7) Значения этого теста не представляют интереса. Представляет интерес то как вставлено ожидание в обоих скриптах. При создании своих тестов нужно обращать внимание на то, что используя SQL команды или запуская команды в psql нужно проверять нет ли побочных эффектов в виде удержания блокировок, создания транзакций, удержания горизонта или моментального снимка.

Примерно с 300 секунд удержания горизонта в окне с тестом pgbench станет заметным снижение tps:

```
progress: 3510.0 s, 173.2 tps, lat 23.092 ms stddev 7.375, 0 failed
progress: 3540.0 s, 172.1 tps, lat 23.240 ms stddev 7.978, 0 failed
progress: 3570.0 s, 171.5 tps, lat 23.311 ms stddev 8.286, 0 failed
progress: 3600.0 s, 172.3 tps, lat 23.211 ms stddev 8.417, 0 failed
progress: 3630.0 s, 174.9 tps, lat 22.871 ms stddev 9.124, 0 failed
progress: 3660.0 s, 170.6 tps, lat 23.442 ms stddev 8.418, 0 failed
```

```
60
 380 kB
120
 282 kB
180
 317 kB
360
 227 kB
240
 210 kB
300
 172 kB
420
 165 kB
360
 152 kB
480
 97 kB
420
 161 kB
540
 168 kB
480
 137 kB
```

8) Остановите тесты в обоих окнах.

9) Для справки результат долгого теста с удержанием горизонта и нагрузкой из одной сессии. В этом тесте минимум объема WAL при checkpoint_timeout равном 20 минутам. С этого интервала грязные буфера начинают удерживаться в буферном кэше.

```
300
 323 kB
           0 |     6664 |      0 |        0
           1 |       75 |      6 |        0
           2 |       34 |      9 |        0
           3 |       55 |     19 |        0
           4 |       55 |     38 |        0
           5 |     9501 |   7878 |        0
600
 299 kB
           0 |     4263 |      0 |        0
           1 |       67 |      0 |        0
           2 |       25 |      0 |        0
           3 |       36 |      0 |        0
           4 |       15 |      0 |        0
           5 |    11978 |   9416 |        0
900
 291 kB
           0 |      683 |      0 |        0
           1 |       67 |      0 |        0
           2 |       25 |      0 |        0
           3 |       36 |      0 |        0
           4 |       14 |      0 |        0
           5 |    15559 |  10646 |        0
1200
 271 kB
           0 |     4081 |      0 |        0
           1 |      279 |    118 |        0
           2 |      878 |    868 |        0
           3 |      962 |    960 |        0
```

```
           4 |    1205 |  1204 |       0
           5 |    8979 |  7697 |       0
1500
 275 kB
           0 |    2543 |     0 |       0
           1 |    1473 |  1438 |       0
           2 |    1448 |  1440 |       0
           3 |    1558 |  1557 |       0
           4 |     384 |   379 |       0
           5 |    8978 |  7711 |       0
1800
 272 kB
           0 |     732 |     0 |       0
           1 |    1403 |  1243 |       0
           2 |    2161 |  2153 |       0
           3 |    2160 |  2158 |       0
           4 |     260 |   253 |       0
           5 |    9668 |  8389 |       0
2100
 282 kB
           0 |     901 |     0 |       0
           1 |    2892 |  2829 |       0
           2 |    2488 |  2478 |       0
           3 |     293 |   289 |       0
           4 |     662 |   655 |       0
           5 |    9148 |  7800 |       0
2400
 271 kB
           0 |    1064 |     0 |       0
           1 |    3872 |  3829 |       0
           2 |      44 |    34 |       0
           3 |     844 |   841 |       0
           4 |    1982 |  1938 |       0
           5 |    8578 |  7344 |       0
2700
 270 kB

           0 |    1070 |     0 |       0
           1 |    1537 |    42 |       0
           2 |     547 |   537 |       0
           3 |    1920 |  1919 |       0
           4 |    1502 |  1501 |       0
           5 |    9808 |  8597 |       0
3000
 293 kB

           0 |     672 |     0 |       0
           1 |    1705 |  1278 |       0
           2 |    2758 |  2743 |       0
           3 |    1659 |  1600 |       0
           4 |    1236 |  1109 |       0
           5 |    8354 |  6985 |       0
3300
 303 kB
           0 |    1034 |     0 |       0
           1 |     563 |   530 |       0
           2 |     583 |   571 |       0
           3 |     920 |   919 |       0
           4 |    2389 |  2388 |       0
           5 |   10895 |  9551 |       0
3600
 302 kB
           0 |    2310 |     0 |       0
           1 |     492 |   463 |       0
           2 |     918 |   904 |       0
           3 |     855 |   851 |       0
           4 |    2879 |  2874 |       0
           5 |    8930 |  7585 |       0
3900
 311 kB
           0 |    3285 |     0 |       0
           1 |     536 |   485 |       0
           2 |     937 |   928 |       0
           3 |     856 |   855 |       0
           4 |    1555 |  1532 |       0
           5 |    9215 |  7969 |       0
4200
 269 kB
           0 |    2195 |     0 |       0
           1 |    1104 |  1062 |       0
           2 |    1189 |  1178 |       0
           3 |     950 |   950 |       0
           4 |    1545 |  1543 |       0
           5 |    9401 |  8179 |       0
```

```
4500
 309 kB
           0 |    1634 |     4 |       0
           1 |    1517 |  1488 |       0
           2 |     991 |   982 |       0
           3 |    1038 |  1038 |       0
           4 |     532 |   531 |       0
           5 |   10672 |  9324 |       0
4800
 313 kB
           0 |    1035 |     0 |       0
           1 |    1680 |  1616 |       0
           2 |    1520 |   181 |       0
           3 |    2453 |  2061 |       0
           4 |    1035 |  1034 |       0
           5 |    8661 |  7432 |       1
```

Часть 2. Задержка при запуске экземпляра. Параметр recovery_init_sync_method

Для проведения теста воспользуемся скриптом из части 6 практики к главе 4.

1) Создайте файл для удаления таблиц `initdrop.sql`:

```
postgres@tantor:~$ mcedit initdrop.sql
```

```
\timing on
DO
$$
begin
 for i in 1..10000 by 100 loop
  for j in 0..99 loop
   execute concat('drop table if exists test.film_summary',i+j);
  end loop;
  commit;
 end loop;
 execute 'drop schema if exists test cascade';
end;
$$
LANGUAGE plpgsql;
```

1) Создайте или проверьте содержимое файла `init.sql`:

```
postgres@tantor:~$ mcedit init.sql
```

```
create schema test;
select format('create table test.film_summary%s (film_id int, title varchar,
release_year smallint) with (autovacuum_enabled=off);', g.id)
 from generate_series(1, 10000) as g(id)
\gexec
```

2) Выполните команды:

```
postgres@tantor:~$
time psql -f initdrop.sql 2> /dev/null
psql -c "checkpoint;"
time psql -f init.sql > /dev/null
pg_ctl stop -m immediate
rm -f $PGDATA/log/*
time sudo systemctl start tantor-se-server-16
Timing is on.
DO
Time: 5920.933 ms (00:05.921)
```

```
real    0m5.938s
user    0m0.009s
sys     0m0.000s
CHECKPOINT
psql:init.sql:1: NOTICE:  schema "test" does not exist, skipping

real    2m38.121s
user    0m0.337s
sys     0m0.185s
waiting for server to shut down.... done
server stopped

real    2m36.460s
user    0m0.016s
sys     0m0.000s
```

Команды выполнили контрольную точку, чтобы время в логе не включало время протяженной контрольной точки.

За **2m38**.121s были созданы 10000 таблиц, 10000 TOAST-таблиц, 10000 индексов на TOAST-таблицы.

Был остановлен экземпляр в режиме **immediate** (симуляция сбоя экземпляра).

За **2m36**.460s был запущен экземпляр.

Список процессов, пока экземпляр запускался был таким:

```
postgres@tantor:~$ ps -ef | grep postgres
postgres   62347    7689  0 00:08 pts/0    00:00:00 pg_ctl start
postgres   62349   62347  0 00:08 ?        00:00:00 /opt/tantor/db/16/bin/postgres
postgres   62350   62349  0 00:08 ?        00:00:00 postgres: logger
postgres   62351   62349  0 00:08 ?        00:00:00 postgres: checkpointer
postgres   62352   62349  0 00:08 ?        00:00:00 postgres: background writer
postgres   62353   62349  4 00:08 ?        00:00:01 postgres: startup
postgres   62361   62319 99 00:08 pts/2    00:00:00 ps -ef
postgres   62362   62319  0 00:08 pts/2    00:00:00 grep postgres
```

3) Пока не завершится работа процесса startup и пока не выполнится контрольная точка после восстановления, подсоединиться к экземпляру не удастся:

```
postgres@tantor:~$ psql
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:
FATAL:  the database system is not yet accepting connections
DETAIL:  Consistent recovery state has not been yet reached.
```

4) После запуска посмотрите содержимое диагностического лога:

```
postgres@tantor:~$ cat $PGDATA/log/postgresql-*
14:13:05.688 MSK [83807] LOG:  starting Tantor Special Edition 16.6.1 a74db619 on x86_64-pc-linux-gnu, compiled
by gcc (Astra 12.2.0-14.astra3+b1) 12.2.0, 64-bit
14:13:05.689 MSK [83807] LOG:  listening on IPv4 address "127.0.0.1", port 5432
14:13:05.689 MSK [83807] LOG:  listening on IPv6 address "::1", port 5432
14:13:05.701 MSK [83807] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
14:13:05.736 MSK [83812] LOG:  database system was interrupted; last known up at 14:10:26 MSK
14:13:16.095 MSK [83812] LOG:  syncing data directory (fsync), elapsed time: 10.00 s, current path:
./base/5/1057586
14:13:26.095 MSK [83812] LOG:  syncing data directory (fsync), elapsed time: 20.00 s, current path:
./base/5/1074286
14:13:36.096 MSK [83812] LOG:  syncing data directory (fsync), elapsed time: 30.00 s, current path:
./base/5/1077650
14:13:46.097 MSK [83812] LOG:  syncing data directory (fsync), elapsed time: 40.00 s, current path:
./base/5/1094421
14:13:56.095 MSK [83812] LOG:  syncing data directory (fsync), elapsed time: 50.00 s, current path:
./base/5/1065705
14:14:06.096 MSK [83812] LOG:  syncing data directory (fsync), elapsed time: 60.00 s, current path:
./base/5/1061386
```

```
14:14:16.096 MSK [83812] LOG:  syncing data directory (fsync), elapsed time: 70.00 s, current path:
./base/5/1093966
14:14:21.053 MSK [83812] LOG:  database system was not properly shut down; automatic recovery in progress
14:14:21.078 MSK [83812] LOG:  redo starts at 3D/2FB2B608
14:14:22.988 MSK [83812] LOG:  invalid record length at 3D/3ABE40D8: expected at least 26, got 0
14:14:22.988 MSK [83812] LOG:  redo done at 3D/3ABE3C50 system usage: CPU: user: 1.89 s, system: 0.00 s,
elapsed: 1.91 s
14:14:23.094 MSK [83810] LOG:  checkpoint starting: end-of-recovery immediate wait
14:15:41.933 MSK [83810] LOG:  checkpoint complete: wrote 15858 buffers (96.8%); 0 WAL file(s) added, 11
removed, 0 recycled; write=1.483 s, sync=77.183 s, total=78.851 s; sync files=30045,
longest=0.049 s, average=0.003 s; distance=180962 kB, estimate=180962 kB; lsn=3D/3ABE40D8, redo
lsn=3D/3ABE40D8
   14:15:41.960 MSK [83807] LOG:  database system is ready to accept connections
```

В лог-файл записывается статус процесса startup с интервалом по умолчанию раз в 10 секунд, который устанавливается параметром конфигурации `log_startup_progress_interval`.

Время синхронизации файлов `14:14:21` - `14:13:05` = 76 секунд.
Время восстановления (наката журналов) `14:15:41` - `14:14:21` = 80 секунд.

5) Посмотрите, что изменится, если перед остановкой экземпляра выполнить контрольную точку командой **checkpoint**:

```
postgres@tantor:~$
time psql -f initdrop.sql 2> /dev/null
psql -c "checkpoint;"
time psql -f init.sql > /dev/null
time psql -c "checkpoint;"
pg_ctl stop -m immediate
time sudo systemctl start tantor-se-server-16
cat $PGDATA/log/postgresql-*
Timing is on.
DO
Time: 5417.500 ms (00:05.418)

real    0m5.454s
user    0m0.014s
sys     0m0.007s
CHECKPOINT
psql:init.sql:1: NOTICE:  schema "test" does not exist, skipping

real    2m21.283s
user    0m0.340s
sys     0m0.173s
CHECKPOINT

real    1m13.241s
user    0m0.004s
sys     0m0.005s
waiting for server to shut down.... done
server stopped

real    1m15.575s
user    0m0.012s
sys     0m0.004s
```

При выполнении **checkpoint** перед остановкой экземпляра контрольная точки и запуск выполнялись **1m13**.241s+**1m15**.575s=2m28.186 секунд. Время простоя **1m15**.575s.
В отсутствие контрольной точки перед остановкой экземпляра запуск экзмпляра занял **2m36**.460s. Время простоя **2m36**.460s. Восстановление заняло чуть больше времени, но сравнимо по скорости с контрольной точкой.

6) Пример содержимого журнала:

```
postgres@tantor:~$ cat $PGDATA/log/postgresql-*

   14:15:41.960 MSK [83807] LOG:  database system is ready to accept connections
   18:32:27.928 MSK [83810] LOG:  checkpoint starting: immediate force wait
   18:32:28.741 MSK [83810] LOG:  checkpoint complete: wrote 5647 buffers (34.5%); 0 WAL file(s) added, 1
removed, 4 recycled; write=0.189 s, sync=0.066 s, total=0.813 s; sync files=15, longest=0.033 s, average=0.005
s; distance=77253 kB, estimate=170591 kB; lsn=3D/3F755900, redo lsn=3D/3F7558B8
   18:34:50.042 MSK [83810] LOG:  checkpoint starting: immediate force wait
   18:36:03.261 MSK [83810] LOG:  checkpoint complete: wrote 11939 buffers (72.9%); 0 WAL file(s) added, 0
removed, 11 recycled; write=0.749 s, sync=72.259 s, total=73.219 s; sync files=30042,
longest=0.034 s, average=0.003 s; distance=179845 kB, estimate=179845 kB; lsn=3D/4B16AEE8, redo
lsn=3D/4A6F6FF8
   18:36:03.268 MSK [83807] LOG:  received immediate shutdown request
   18:36:03.301 MSK [83807] LOG:  database system is shut down
   18:36:03.531 MSK [89269] LOG:  starting Tantor Special Edition 16.6.1 a74db619 on x86_64-pc-linux-gnu,
compiled by gcc (Astra 12.2.0-14.astra3+b1) 12.2.0, 64-bit
   18:36:03.532 MSK [89269] LOG:  listening on IPv4 address "127.0.0.1", port 5432
   18:36:03.532 MSK [89269] LOG:  listening on IPv6 address "::1", port 5432
   18:36:03.540 MSK [89269] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
   18:36:03.570 MSK [89273] LOG:  database system was interrupted; last known up at 18:36:03 MSK
   18:36:13.926 MSK [89273] LOG:  syncing data directory (fsync), elapsed time: 10.00 s, current path:
./base/5/1108344
   18:36:23.925 MSK [89273] LOG:  syncing data directory (fsync), elapsed time: 20.00 s, current path:
./base/5/1111670
   18:36:33.924 MSK [89273] LOG:  syncing data directory (fsync), elapsed time: 30.00 s, current path:
./base/5/1125680
   18:36:43.926 MSK [89273] LOG:  syncing data directory (fsync), elapsed time: 40.00 s, current path:
./base/5/1142589
   18:36:53.926 MSK [89273] LOG:  syncing data directory (fsync), elapsed time: 50.00 s, current path:
./base/5/1127261
   18:37:03.925 MSK [89273] LOG:  syncing data directory (fsync), elapsed time: 60.00 s, current path:
./base/5/1113186
   18:37:13.924 MSK [89273] LOG:  syncing data directory (fsync), elapsed time: 70.00 s, current path:
./base/5/1144321
   18:37:18.586 MSK [89273] LOG:  database system was not properly shut down; automatic recovery in progress
   18:37:18.604 MSK [89273] LOG:  redo starts at 3D/4A6F6FF8
   18:37:18.628 MSK [89273] LOG:  invalid record length at 3D/4B16AF80: expected at least 26, got 0
   18:37:18.628 MSK [89273] LOG:  redo done at 3D/4B16AEE8 system usage: CPU: user: 0.02 s, system: 0.00 s,
elapsed: 0.02 s
   18:37:18.666 MSK [89271] LOG:  checkpoint starting: end-of-recovery immediate wait
   18:37:18.877 MSK [89271] LOG:  checkpoint complete: wrote 1321 buffers (8.1%); 0 WAL file(s) added, 1
removed, 0 recycled; write=0.078 s, sync=0.078 s, total=0.219 s; sync files=13,
longest=0.044 s, average=0.00д6 s; distance=10703 kB, estimate=10703 kB; lsn=3D/4B16AF80, redo
lsn=3D/4B16AF80
   18:37:18.891 MSK [89269] LOG:  database system is ready to accept connections
```

Объем журнальных данных: **distance=179845 kB + 10703 kB** = 190548. Было: **distance=180962 kB**. Небольшое увеличение времени соответствует небольшому увеличению объема журнальных записей.

7) Выполните запрос:

```
postgres=# select * from pg_stat_recovery_prefetch\gx
-[ RECORD 1 ]--+------------------------------
stats_reset    | 2035-01-01 00:08:04.65834+03
prefetch       | 10556
hit            | 435863
skip_init      | 3986
skip_new       | 2
skip_fpw       | 15327
skip_rep       | 432413
wal_distance   | 0
block_distance | 0
io_depth       | 0
```

Представление заполняется процессом startup. Если восстановления при запуске экземпляра не было, значения нулевые.

В представлении отражаются только блоки журналов. Перед остановкой экземпляра создавалось 10000 таблиц и было сгенерировано много журнальных записей.

Описания столбцов представления есть в теоретической части 10 главы.

8) На реплике статистики постоянно увеличиваются и на репликах значения сбрасывают вызовом функции `select pg_stat_reset_shared('recovery_prefetch')`. Выполните сброс статистики:

```
postgres=# select pg_stat_reset_shared('recovery_prefetch');
select * from pg_stat_recovery_prefetch\gx
 pg_stat_reset_shared
----------------------

(1 row)

-[ RECORD 1 ]--+------------------------------
stats_reset    | 2035-01-11 00:09:02.679457+03
prefetch       | 0
hit            | 0
skip_init      | 0
skip_new       | 0
skip_fpw       | 0
skip_rep       | 0
wal_distance   | 0
block_distance | 0
io_depth       | 0
```

Статистика была сброшена.

9) Можно еще раз принудительно остановить экземпляр и снова запустить:

```
postgres@tantor:~$ pg_ctl stop -m immediate
rm -f $PGDATA/log/*
time sudo systemctl start tantor-se-server-16
cat $PGDATA/log/postgresql-*
waiting for server to shut down.... done
server stopped

real    1m13.828s
user    0m0.015s
sys     0m0.000s
03:20:19.094 MSK [14260] LOG:  starting Tantor Special Edition 16.2.0 e12e484f on x86_64-pc-linux-gnu,
compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
03:20:19.094 MSK [14260] LOG:  listening on IPv4 address "127.0.0.1", port 5432
03:20:19.094 MSK [14260] LOG:  listening on IPv6 address "::1", port 5432
03:20:19.105 MSK [14260] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
03:20:19.136 MSK [14265] LOG:  database system was interrupted; last known up at 03:17:41 MSK
03:20:29.582 MSK [14265] LOG:  syncing data directory (fsync), elapsed time: 10.00 s, current path:
./base/5/4066825
03:20:39.584 MSK [14265] LOG:  syncing data directory (fsync), elapsed time: 20.00 s, current path:
./base/5/1903193
03:20:49.583 MSK [14265] LOG:  syncing data directory (fsync), elapsed time: 30.00 s, current path:
./base/5/4075920
03:20:59.584 MSK [14265] LOG:  syncing data directory (fsync), elapsed time: 40.00 s, current path:
./base/5/4059545
03:21:09.583 MSK [14265] LOG:  syncing data directory (fsync), elapsed time: 50.00 s, current path:
./base/5/4048445
03:21:19.582 MSK [14265] LOG:  syncing data directory (fsync), elapsed time: 60.00 s, current path:
./base/5/4079304
03:21:29.583 MSK [14265] LOG:  syncing data directory (fsync), elapsed time: 70.00 s, current path:
./base/5/4081550
03:21:32.276 MSK [14265] LOG:  database system was not properly shut down; automatic recovery in progress
03:21:32.298 MSK [14265] LOG:  redo starts at 37/CC2D07A8
03:21:32.298 MSK [14265] LOG:  invalid record length at 37/CC2D08D0: expected at least 26, got 0
03:21:32.298 MSK [14265] LOG:  redo done at 37/CC2D0888 system usage: CPU: user: 0.00 s, system: 0.00 s,
elapsed: 0.00 s
03:21:32.347 MSK [14263] LOG:  checkpoint starting: end-of-recovery immediate wait
03:21:32.418 MSK [14263] LOG:  checkpoint complete: wrote 4 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0
recycled; write=0.024 s, sync=0.011 s, total=0.079 s; sync files=3, longest=0.005 s, average=0.004 s; distance=0
kB, estimate=0 kB; lsn=37/CC2D08D0, redo lsn=37/CC2D08D0
03:21:32.431 MSK [14260] LOG:  database system is ready to accept connections
```

Из-за наличия 10000 таблиц в одной из баз данных кластера, после остановки экземпляра в режиме immediate, по нехватке памяти или пропаданию питания экземпляр будет запускаться 73 секунды плюс наката файлов журнала и контрольная точка после наката.

Поскольку при предыдущем открытии экземпляра было восстановление (накатился WAL-журнал), выполнена контрольная точка после восстановления и после этого не было изменений, то при запуске экземпляра восстановление и контрольная точка выполнились за долю секунды 03:21:32.418 - 03:21:32.298.

Длительность фазы синхронизации запуска экземпляра зависит от числа файлов в PGDATA кластера.

10) Выполните запрос:

```
postgres=# select * from pg_stat_recovery_prefetch\gx
-[ RECORD 1 ]--+------------------------------
stats_reset    | 2035-01-11 00:16:37.654068+03
prefetch       | 0
hit            | 0
skip_init      | 0
skip_new       | 0
skip_fpw       | 720
skip_rep       | 0
wal_distance   | 0
block_distance | 0
io_depth       | 0
```

При восстановлении процесс startup не накатил ни одного блока журнала.

Как ускорить запуск экземпляра?

11) Выполните команду:

```
postgres=# alter system set recovery_init_sync_method = syncfs;
ALTER SYSTEM
postgres=# \q
```

12) Повторите перезапуск экземпляра:

```
postgres@tantor:~$ pg_ctl stop -m immediate
rm -f $PGDATA/log/*
time sudo systemctl start tantor-se-server-16
cat $PGDATA/log/postgresql-*
waiting for server to shut down.... done
server stopped

real    0m0.511s
user    0m0.015s
sys     0m0.000s
03:27:10.679 MSK [14302] LOG:  starting Tantor Special Edition 16.2.0 e12e484f on x86_64-pc-linux-gnu,
compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
03:27:10.680 MSK [14302] LOG:  listening on IPv4 address "127.0.0.1", port 5432
03:27:10.680 MSK [14302] LOG:  listening on IPv6 address "::1", port 5432
03:27:10.688 MSK [14302] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
03:27:10.717 MSK [14306] LOG:  database system was interrupted; last known up at 03:21:32 MSK
03:27:10.787 MSK [14306] LOG:  database system was not properly shut down; automatic recovery in progress
03:27:10.808 MSK [14306] LOG:  redo starts at 37/CC2D0968
03:27:10.808 MSK [14306] LOG:  invalid record length at 37/CC2D09B0: expected at least 26, got 0
03:27:10.808 MSK [14306] LOG:  redo done at 37/CC2D0968 system usage: CPU: user: 0.00 s, system: 0.00 s,
elapsed: 0.00 s
03:27:10.851 MSK [14304] LOG:  checkpoint starting: end-of-recovery immediate wait
03:27:10.921 MSK [14304] LOG:  checkpoint complete: wrote 4 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0
recycled; write=0.023 s, sync=0.012 s, total=0.079 s; sync files=3, longest=0.005 s, average=0.004 s; distance=0
kB, estimate=0 kB; lsn=37/CC2D09B0, redo lsn=37/CC2D09B0
03:27:10.932 MSK [14302] LOG:  database system is ready to accept connections
```

Экземпляр запустился за долю секунды.

Использование **recovery_init_sync_method = syncfs** не ухудшает отказоустойчивость.

По умолчанию, длительность удержания страниц в страничном кэше linux определяется параметром linux: **vm.dirty_expire_centisecs** - сколько буфер может быть грязным, прежде чем будет помечен для записи, по умолчанию 3000 (30 секунд). Под большой нагрузкой на диск от пометки блока до записи может пройти больше, чем 30 секунд. Настройки страничного кэша linux рассматривались во 2 главе (2-3) на слайде "Длительность удержания грязных страниц в кэше". В примере нагрузки на дисковую систему виртуальной машины не было, а синхронизаций пофайлово заняла 73 секунды. В любом случае и даже в случае если кроме экземпляра диск нагружается другими приложениями, полная синхронизация файловой системы выполнится быстрее, поэтому стоит использовать **recovery_init_sync_method = syncfs**. Это значение не используется по умолчанию, вероятно, из-за опасений наличия ошибок в linux, о чем написано в документации PostgreSQL к параметру конфигурации **recovery_init_sync_method**: "Linux версий до 5.8 может не всегда сообщать экземпляру об ошибках ввода-вывода при записи данных на диск, сообщения можно будет увидеть разве что в журнале ядра". Журнал ядра можно посмотреть командой операционной системы dmesg с привилегиями root.

Оценить время синхронизации файловых систем можно командой операционной системы:

```
postgres@tantor:~$ time sync -f

real    0m0.010s
user    0m0.002s
sys     0m0.000s
```

Из документации linux:

```
   syncfs() is like sync(), but synchronizes just the filesystem containing file referred to by the
open file descriptor fd. sync() is always successful.
   syncfs() can fail for at least the following reasons:
   EBADF fd is not a valid file descriptor.
   EIO An error occurred during synchronization. This error may relate to data written to any file
on the filesystem, or on meta-data related to the filesystem itself.
   ENOSPC Disk space was exhausted while synchronizing.
```

13) У утилиты резервирования есть параметр --no-sync. Проверьте время резервирования без параметра и с параметром:

```
postgres@tantor:~$ rm -rf /var/lib/postgresql/backup/1
time pg_basebackup -c fast -D $HOME/backup/1 -P
rm -rf /var/lib/postgresql/backup/1
time pg_basebackup --no-sync -c fast -D $HOME/backup/1 -P
time sync -f
3783151/3783151 kB (100%), 1/1 tablespace

real    2m50.010s
user    0m52.035s
sys     0m3.515s

real    0m40.807s
user    0m36.967s
sys     0m2.612s

real    0m0.007s
user    0m0.001s
sys     0m0.000s
```

Время резервирования без параметра существенно больше, чем с параметром и командой `sync -f`. В 17 версии появился параметр --sync-method=syncfs.

Параметр **-c fast** позволяет не ждать выполнения контрольной точки, он действует при подключении к мастеру. Параметр используют, чтобы быстрее начать бэкап. Если время начала копирования не критично, лучше дождаться выполнения контрольной точки по времени.

14) Попробуйте удалить созданные 10000 таблиц:

```
postgres=# \timing on \\
drop schema test cascade;
ERROR:  out of shared memory
HINT:  You might need to increase max_locks_per_transaction.
Time: 108.804 ms
```

Либо придется писать скрипт с удалением таблиц по одной, либо увеличить число блокировок.

15) Создайте файл для удаления таблиц, если он ещё не был создан `initdrop.sql`:

```
postgres@tantor:~$ cat initdrop.sql

\timing on
DO
$$
begin
 for i in 1..10000 by 100 loop
  for j in 0..99 loop
   execute concat('drop table if exists test.film_summary',i+j);
  end loop;
  commit;
 end loop;
 execute 'drop schema if exists test cascade';
end;
$$
LANGUAGE plpgsql;
```

16) Выполните скрипт :

```
postgres@tantor:~$ time psql -f initdrop.sql 2> /dev/null

real    0m5.678s
user    0m0.009s
sys     0m0.000s
```

Таблицы были удалены за 5,678 секунд.

Часть 3. Длительность контрольной точки

1) Создайте файл `init1.sql`:

```
postgres@tantor:~$ mcedit init1.sql

  create schema test;
  select format('create table test.film_summary%s (film_id int, title varchar,
release_year smallint) with (autovacuum_enabled=off);', g.id) from
generate_series(1, 10000) as g(id)
  \gexec
  select format('insert into test.film_summary%1$s select i, %2$s || i, i from
generate_series(0, 300) as i;', g.id, E'\'text number \'') from
generate_series(1, 10000) as g(id)
  \gexec
  select format('select * from test.film_summary%s where film_id = 0;', g.id)
from generate_series(1, 10000) as g(id)
   \gexec
```

Скрипт создаст 10000 таблиц в схеме test по 300 строк в каждой таблице в двух блоках и перечитает буфера, чтобы они не грязнились при чтении и это не создавало погрешности в результатах тестов.

2) Установите параметры конфигурации:

```
postgres=# alter system set checkpoint_timeout = 1200;
alter system set recovery_init_sync_method = syncfs;
alter system set shared_buffers = '512MB';
alter system set max_connections = 1000;
alter system set log_min_duration_statement = '15s';
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
postgres=# \q
```

Интервал контрольных точек длиннее, чтобы контрольные точки меньше влияли на результаты и интервал был приближен к значениям, используемым на практике. 5 минут на практике не используются.

Если редактировать числа в тесте, то нужно следить, чтобы тесты не сгенерировали объем журналов больше, чем max_wal_size (1Гб по умолчанию).

recovery_init_sync_method = syncfs, чтобы не ждать лишние ~80 секунд в случае остановки экземпляра в режиме immediate. Эти ~80 секунд не имеют отношения к контрольным точкам и накату журналов и зависят только от числа файлов в PGDATA.

Размер пула буферов увеличен, так как грязных буферов ожидается больше, чем 128Мб. При реальной эксплуатации в кэше буферов почти нет свободных буферов, все буфера заняты блоками объектов.

Число соединений увеличено, чтобы не сталкиваться с ограничением по числу блокировок и приблизиться к к значениям, используемым на практике.

Логирование длинных команд на случай, если такие команды будут выполняться чтобы обратить на них внимание.

3) Проверьте, что содержимое файла параметров `postgresql.auto.conf` такое:

```
postgres@tantor:~$ cat $PGDATA/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
logging_collector = 'on'
checkpoint_timeout = '1200'
recovery_init_sync_method = 'syncfs'
shared_buffers = '1GB'
max_connections = '1000'
log_min_duration_statement = '15s'
```

4) Удалите лог, чтобы не разыскивать среди набора логов самый последний, рестаруйте экземпляр для применения значений параметров конфигурации и очистки кэша буферов, выполните тестовый скрипт:

```
postgres@tantor:~$ rm -f $PGDATA/log/*
psql -c "vacuum full;"
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null

VACUUM
real    3m34.352s
user    0m1.554s
sys     0m0.498s
```

Тестовый скрипт удобен тем, что создает много грязных буферов в 10000 таблиц. При реальной работе также обновляется много блоков в большом числе таблиц и индексов.

Использование одной таблицы в тесте, путь даже большого размера, было бы далеко от реальной работы.

5) Выполните запрос:

```
postgres@tantor:~$ psql -c "select * buffercache_usage_counts();"
 usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
           0 |    4759 |     0 |      0
           1 |   20126 | 10002 |      0
           2 |   10027 |     1 |      0
           3 |      12 |     2 |      0
           4 |      20 |     2 |      0
           5 |   30592 | 28466 |      0
(6 rows)
```

Большая часть буферов из находящихся в буферном кэше (~50тыс.) грязная (38тыс.).
Свободно 4762 буфера.

5) Выполните контрольную точку:

```
postgres@tantor:~$ psql -c "alter system set log_statement = ddl;"
psql -c "select pg_reload_conf();"
time psql -c "checkpoint;"
psql -c "alter system reset log_statement"
psql -c "select pg_reload_conf();"
cat $PGDATA/log/postgresql-*
ALTER SYSTEM
 pg_reload_conf
----------------
 t
(1 row)

CHECKPOINT

real    1m23.284s
user    0m0.010s
sys     0m0.000s
ALTER SYSTEM
 pg_reload_conf
----------------
 t
(1 row)

03:19:20.674 MSK [99626] LOG:  starting Tantor Special Edition 16.2.0 e12e484f on x86_64-pc-linux-gnu,
compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
03:19:20.674 MSK [99626] LOG:  listening on IPv4 address "127.0.0.1", port 5432
03:19:20.674 MSK [99626] LOG:  listening on IPv6 address "::1", port 5432
03:19:20.683 MSK [99626] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
03:19:20.705 MSK [99631] LOG:  database system was shut down at 03:19:19 MSK
03:19:20.719 MSK [99626] LOG:  database system is ready to accept connections
03:25:00.845 MSK [99626] LOG:  received SIGHUP, reloading configuration files
03:25:00.846 MSK [99626] LOG:  parameter "log_statement" changed to "ddl"
03:25:00.866 MSK [99629] LOG:  checkpoint starting: immediate force wait
03:26:24.124 MSK [99629] LOG:  checkpoint complete: wrote 38498 buffers (58.7%); 0 WAL file(s) added,
0 removed, 27 recycled; write=2.811 s, sync=80.146 s, total=83.259 s; sync files=40047,
longest=0.041 s, average=0.002 s; distance=441826 kB, estimate=441826 kB; lsn=3A/E67EE0D0, redo
lsn=3A/E67EE088
03:26:24.135 MSK [99661] LOG:  duration: 83269.646 ms  statement: checkpoint;
03:26:24.149 MSK [99672] LOG:  statement: alter system reset log_statement
03:26:24.184 MSK [99626] LOG:  received SIGHUP, reloading configuration files
03:26:24.185 MSK [99626] LOG:  parameter "log_statement" removed from configuration file, reset to default
```

Контрольная точка выполнялась 83 секунды.
В 03:25:00 началась контрольная точка по команде (immediate force wait).

log_statement = ddl единственное что сделал - вывел строку **duration: 83269.646 ms statement: checkpoint;** в конце контрольной точки. Момент выдачи команды этот параметр конфигурации не вывел. **duration: 83269.646 ms** это длительность работы контрольной точки с максимальной скоростью от выдачи команды checkpoint до завершения контрольной точки. Если работает протяженная контрольная точка, то total=**83.259 s** выдает длительность протяженной контрольной точки и не совпадает с **duration**. В данном примере протяженной контрольной точки не было и значение **duration** похоже на total.

В WAL было создано **distance=441826 kB** = 55228 блоков.

Грязных блоков было 28466+10007=38453. Такое число блоков было записано в режиме full page writes. Поскольку блоки таблиц были почти полностью заполнены, то в WAL они заняли столько же места.

6) Проверьте, что грязных буферов нет:

```
postgres@tantor:~$ psql -c "select * buffercache_usage_counts();"
 usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
           0 |    4748 |     0 |      0
           1 |   20128 |     0 |      0
           2 |   10027 |     0 |      0
           3 |      11 |     0 |      0
           4 |      20 |     0 |      0
           5 |   30602 |     0 |      0
(6 rows)
```

Все грязные буфера были записаны по контрольной точке на диск.

7) В блоках таблиц по 2 блока, в этих блоках свободно 40 и 612 байт:

```
postgres@tantor:~$ psql -c "select upper-lower free from
page_header(get_raw_page('test.film_summary1','main',0));"
  psql -c "select upper-lower free from
page_header(get_raw_page('test.film_summary1','main',1));"
 free
------
   40
(1 row)

 free
------
  612
(1 row)
```

При выдаче команды CHECKPOINT после того как экземпляр проработал checkpoint_timeout времени чтобы читать лог нужно знать следующее.

Если бы запустилась контрольная точка по времени, то команда CHECKPOINT ускоряет контрольную точку по времени, которая начинает работать с максимальной скоростью. После чего запускается еще одна контрольная точка (по команде CHECKPOINT) и она выполняется быстро.

То есть статистику выполнения контрольной точки по команде checkpoint нужно смотреть в сообщении, относящемся к предыдущей контрольной точке - по времени.

8) Рестарт экземпляра пройдет моментально, так как грязных буферов нет:

```
postgres@tantor:~$ time sudo systemctl restart tantor-se-server-16

real    0m0.515s
user    0m0.015s
sys     0m0.000s
```

Поэтому если нужно остановить экземпляр, то рекомендуется выполнить контрольную точку командой CHECKPOINT. Выигрыш в том, что пока команда работает экземпляр доступен и обслуживает клиентов. Если экземпляру дать команду остановки, то экземпляр уже не доступен клиентам и начинается простой обслуживания.

9) Удалите таблицы одной транзакцией:

```
postgres@tantor:~$ psql

postgres=# \timing on \\
DO
$$
begin
 for i in 1..10000 loop
   execute concat('drop table if exists test.film_summary',i);
 end loop;
 execute 'drop schema if exists test cascade';
end;
$$
LANGUAGE plpgsql;

DO
Time: 6399.334 ms (00:06.399)
```

Таблицы удалялись 6399.334 ms.

Часть 4. Длительность финальной контрольной при остановке экземпляра

Проверим, что длительность финальной контрольной точки при постановке экземпляра точно такая же, как по команде CHECKPOINT

1) Повторите создание таблиц:

```
postgres@tantor:~$ rm -f $PGDATA/log/*
psql -c "vacuum full;"
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null
time sudo systemctl stop tantor-se-server-16
cat $PGDATA/log/postgresql-*

VACUUM
real    3m32.800s
user    0m1.545s
sys     0m0.553s

real    1m47.895s
user    0m0.011s
sys     0m0.004s

04:55:01.531 MSK [100981] LOG:  starting Tantor Special Edition 16.2.0 e12e484f on x86_64-pc-linux-gnu, compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
04:55:01.531 MSK [100981] LOG:  listening on IPv4 address "127.0.0.1", port 5432
04:55:01.531 MSK [100981] LOG:  listening on IPv6 address "::1", port 5432
04:55:01.540 MSK [100981] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
04:55:01.561 MSK [100985] LOG:  database system was shut down at 04:55:00 MSK
04:55:01.575 MSK [100981] LOG:  database system is ready to accept connections
05:01:37.224 MSK [100981] LOG:  received fast shutdown request
05:01:37.237 MSK [100981] LOG:  aborting any active transactions
05:01:37.241 MSK [100981] LOG:  background worker "logical replication launcher" (PID 100988) exited with exit code 1
```

```
05:01:37.245 MSK [100983] LOG:  shutting down
05:01:37.252 MSK [100983] LOG:  checkpoint starting: shutdown immediate
05:03:23.824 MSK [100983] LOG:  checkpoint complete: wrote 38495 buffers (58.7%); 0 WAL file(s) added, 0 removed, 27 recycled; write=3.671 s, sync=102.522 s, total=106.580 s; sync files=40047, longest=0.145 s, average=0.003 s; distance=441785 kB, estimate=441785 kB; lsn=3B/28562C40, redo lsn=3B/28562C40
05:03:23.912 MSK [100981] LOG:  database system is shut down
```

Длительность контрольной точки: `05:03:23 - 05:01:37 = 106` секунд.

Остановка командой pg_ctl stop или systemctl stop на длительность остановки не влияет, так как systemctl stop вызывает pg_ctl stop.

`distance=441785 kB` и число файлов `files=40047` такие же, как при выполнении команды CHECKPOINT.

`write=3.671 s` немного больше, чем был: `write=2.811 s.`

Результат теста: финальная контрольная точка не быстрее ручной контрольной точки. В примере финальная контрольная точка выполнялась дольше на 23 секунды: 106 секунд вместо 83 секунд.

3) Запустите экземпляр:

```
postgres@tantor:~$ sudo systemctl start tantor-se-server-16
```

4) Удалите таблицы по 100 штук в транзакции:

```
postgres=# \timing on \\
DO
$$
begin
 for i in 1..10000 by 100 loop
  for j in 0..99 loop
   execute concat('drop table if exists test.film_summary',i+j);
  end loop;
  commit;
 end loop;
 execute 'drop schema if exists test cascade';
end;
$$
LANGUAGE plpgsql;

DO
Time: 6979.210 ms (00:06.979)
```

Удаление заняло `6979 ms`.

В прошлый раз удаляли одной транзакцией и это заняло `6399.334 ms`. Сейчас при удалении по 100 таблиц в 100 транзакциях удаление заняло `6979 ms`. Удаление одной транзакцией более эффективно, но требует больше блокировок.

Часть 5. Длительность контрольной точки после падения экземпляра

В этой части практики проделаем ту же последовательность действий что и в предыдущей части, только без контрольной точки, с остановкой экземпляра в режиме immediate (без контрольной точки).

Цель: определить сколько времени займёт восстановление блоков по WAL процессом startup при запуске экземпляра и изменится ли длительность контрольной точки.

1) Повторите создание таблиц:

```
postgres@tantor:~$ rm -f $PGDATA/log/*
psql -c "vacuum full;"
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null
cat $PGDATA/log/postgresql-* | egrep "checkpoint|ready"

VACUUM
real    3m36.456s
user    0m1.498s
sys     0m0.562s
03:59:21.293 MSK [99814] LOG:  database system is ready to accept connections
```

Таблицы создавались столько же времени, что и в прошлый раз.

2) Остановите экземпляр без контрольной точки и запустите экземпляр:

```
postgres@tantor:~$ pg_ctl stop -m immediate
rm -f $PGDATA/log/*
time sudo systemctl start tantor-se-server-16
tail -n 7  $PGDATA/log/postgresql-*

waiting for server to shut down.... done
server stopped

real    1m34.397s
user    0m0.015s
sys     0m0.000s
04:07:27.828 MSK [100872] LOG:  database system was not properly shut down; automatic recovery in progress
04:07:27.856 MSK [100872] LOG:  redo starts at 3A/EC71EC38
04:07:37.561 MSK [100872] LOG:  invalid record length at 3B/769AFC8: expected at least 26, got 0
04:07:37.561 MSK [100872] LOG:  redo done at 3B/769AF80 system usage: CPU: user: 5.54 s, system:
1.27 s, elapsed: 9.70 s
04:07:37.618 MSK [100870] LOG:  checkpoint starting: end-of-recovery immediate wait
04:09:01.824 MSK [100870] LOG:  checkpoint complete: wrote 48483 buffers (74.0%); 0 WAL file(s) added, 0
removed, 27 recycled; write=3.816 s, sync=80.096 s, total=84.213 s; sync files=40049,
longest=0.087 s, average=0.002 s; distance=441841 kB, estimate=441841 kB; lsn=3B/769AFC8, redo
lsn=3B/769AFC8
04:09:01.860 MSK [100868] LOG:  database system is ready to accept connections
```

Экземпляр запускался 94 секунды.

Сначала был накат журнальных данных, он длился 10 секунд с `04:07:27` до `04:07:37`.

Дальше выполнялась контрольная точка после восстановления, которая длилась `total=84.213 s`. Передача блоков из буферного кэша в страничный кэш linux заняло всего `write=3.816 s`. В WAL-файлы было записано `distance=441841 kB`.

Вызов fsync отправлялся по `40049` файлам и длился `sync=80.096 s`.

По большей части это те же 73 секунды, которые выполнялся fsync по отдельным файлам во 2 части практики. Если бы по файловой системе отправлялся вызов syncfs или sync он бы длился не более 80-73=7 секунд.

Для контрольных точек по требованию коду PostgreSQL было бы оптимальнее выполнять syncfs по файловым системам с PGDATA, вместо fsync по отдельным файлам, хотя бы на контрольной точке при остановке экземпляра. Параметр `recovery_init_sync_method` появился в 14 версии PostgreSQL и пока имеет значение по умолчанию fsync. Параметр `--sync-method=syncfs` у утилиты `pg_basebackup` появился в 17 версии. То есть поддержка полной синхронизации файловых систем `syncfs` в процессе развития.

Можно сравнить с логом из 5 пункта этой части практики, значения почти те же самые.

Результат теста: рестарт экземпляра длился на на 10 секунд больше, чем если бы экземпляр был остановлен корректно. 10 секунд длился накат журналов.

3) Проверим как выполняется команда `drop schema test cascade`. Убедимся, что эта команда не накручивает счетчик транзакций и мультитранзакций. Выполните команды:

```
postgres@tantor:~$ psql -c "SELECT datname, age(datfrozenxid),
mxid_age(datminmxid) FROM pg_database where datname='postgres';"
psql -c "select pg_current_xact_id();"
time psql -c "drop schema test cascade;"
psql -c "SELECT datname, age(datfrozenxid), mxid_age(datminmxid) FROM
pg_database where datname='postgres';"
select pg_current_xact_id();"
 datname  |  age   | mxid_age
----------+--------+----------
 postgres | 594926 |        0
(1 row)

 pg_current_xact_id
--------------------
            6949699
(1 row)

NOTICE:  drop cascades to 10000 other objects
DETAIL:  drop cascades to table test.film_summary1
drop cascades to table test.film_summary2
...
drop cascades to table test.film_summary100
and 9900 other objects (see server log for list)
DROP SCHEMA

real    0m7.658s
user    0m0.011s
sys     0m0.000s
 datname  |  age   | mxid_age
----------+--------+----------
 postgres | 594928 |        0
(1 row)

 pg_current_xact_id
--------------------
            6949706
(1 row)
```

Команда **drop schema test cascade** не накручивают счетчики транзакций.

У значения xid разница в **6949706-6949699**=5 транзакций (6 транзакция была создана вызовом функции). Команда drop schema использовала менее чем, в 5 транзакций или подтранзакций (savepoint).

Результат тестов: Команда `drop schema cascade` выполнялась медленнее (`7.658s`), чем удаление блоком plpgsql в 100 транзакциях по 100 таблиц в каждой транзакции (`6979 ms`) и медленнее, чем одной транзакцией (`6399.334 ms`).

Преимущество удаления в 100 транзакциях в том, что каждая из транзакций длится `6979 ms/100=69ms` и удержание горизонта базы данных в 100 раз короче, чем у других способов удаления таблиц.

Часть 6. Контрольная точка по запросу

Хочется посмотреть как читать лог, если началась контрольная точка по времени и выдана контрольная точка по команде checkpoint.

Также проверить влияет ли **max_connections** на скорость выполнения контрольных точек и создание, удаление объектов. Поэтому уменьшим до значения по умолчанию 100.

Чтобы долго не ждать установим **checkpoint_timeout** в значение по умолчанию 5 минут.

1) Выполните команды (время выполнения примерно 7 минут):

```
postgres@tantor:~$
psql -c "alter system reset checkpoint_timeout;"
psql -c "alter system reset max_connections;"
```

```
psql -c "vacuum full;"
rm -f $PGDATA/log/*
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null
psql -c "select backend_type name, sum(writes) buffers_written, sum(writebacks) writebacks, sum(evictions) evictions, sum(fsyncs) fsyncs, sum(fsync_time) fsync_time from pg_stat_io group by backend_type having sum(writes)> 0 or sum(writebacks)> 0 or sum(fsyncs)>0 or sum(evictions)>0;"
sleep 100
psql -c "select * buffercache_usage_counts();"
sleep 10
psql -c "select * buffercache_usage_counts();"
time psql -c "checkpoint;"
tail -6 $PGDATA/log/postgresql-*
psql -c "select backend_type name, sum(writes) buffers_written, sum(writebacks) writebacks, sum(evictions) evictions, sum(fsyncs) fsyncs, sum(fsync_time) fsync_time from pg_stat_io group by backend_type having sum(writes)> 0 or sum(writebacks)> 0 or sum(fsyncs)>0 or sum(evictions)>0;"
```

```
VACUUM
ALTER SYSTEM
ALTER SYSTEM

real    3m31.394s
user    0m1.400s
sys     0m0.678s

 usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
           0 |    4765 |     0 |      0
           1 |   20125 | 10001 |      0
           2 |   10024 |     0 |      0
           3 |      10 |     0 |      0
           4 |      19 |     0 |      0
           5 |   30593 | 26780 |      0
(6 rows)

 usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
           0 |    4765 |     0 |      0
           1 |   20119 | 10000 |      0
           2 |   10026 |     0 |      0
           3 |      12 |     0 |      0
           4 |      19 |     0 |      0
           5 |   30595 | 25364 |      0
(6 rows)

       name        | buffers_written | writebacks | evictions | fsyncs |fsync_time
-------------------+-----------------+------------+-----------+--------+-----------
 client backend    |              65 |          0 |         0 |      0 |          0
 autovacuum worker |            1259 |          0 |         0 |      0 |          0
 checkpointer      |          640928 |     640928 |           | 591409 |          0
(3 rows)

CHECKPOINT

real    1m22.741s
user    0m0.009s
sys     0m0.000s

07:51:35.353 MSK [104999] LOG:  database system is ready to accept connections
07:56:35.861 MSK [105001] LOG:  checkpoint starting: time
07:58:19.555 MSK [105001] LOG:  checkpoint complete: wrote 38493 buffers (58.7%); 0 WAL file(s) added, 0 removed, 27 recycled; write=23.442 s, sync=79.949 s, total=103.694 s; sync files=40045, longest=0.072 s, average=0.002 s; distance=441819 kB, estimate=441819 kB; lsn=3B/F9ABE3F0, redo lsn=3B/F9ABE3A8
07:58:19.565 MSK [105001] LOG:  checkpoint starting: immediate force wait
07:58:19.604 MSK [105001] LOG:  checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.001 s, sync=0.001 s, total=0.040 s; sync files=0, longest=0.000 s, average=0.000 s; distance=0 kB, estimate=397637 kB; lsn=3B/F9ABE518, redo lsn=3B/F9ABE4D0
07:58:19.604 MSK [105033] LOG:  duration: 82727.854 ms  statement: checkpoint;

       name        | buffers_written | writebacks | evictions | fsyncs | fsync_time
-------------------+-----------------+------------+-----------+--------+-----------
 client backend    |              65 |          0 |         0 |      0 |          0
 autovacuum worker |            1259 |          0 |         0 |      0 |          0
 checkpointer      |          679404 |     679404 |           | 631443 |          0
(3 rows)
  разница с предыдущим запросом:     |       38476 |           |  40034 |
```

Пояснение к командам: была сделана пауза на 100 секунд (команда **sleep 100**), так как создание таблиц длится примерно 3 минуты 30 секунд. 3 минуты 30 секунд плюс 100 секунд равно 5 минут 10 секунд, то есть контрольная точка по времени уже начнет работать.

Значения **duration: 82727.854 ms** равно **real 1m22.741s** (время выполнения команды checkpoint **с максимальной скоростью**).

Сразу после запуска экземпляра 07:51:35.353 контрольная точка не запускается. При этом процесс checkpointer накапливает от других процессов списки блоков на writeback и fsync.

writebacks в исходном коде эта статистика называется IOOP_WRITEBACK, увеличиватся на число блоков в очереди на запись nr_pending и схематично выполняется так:

```
  * Executing the writes in-order can make them a lot faster, and allows to
  * merge writeback requests to consecutive blocks into larger writebacks.
  sort_pending_writebacks(wb_context->pending_writebacks, wb_context->nr_pending);
  * Coalesce neighbouring writes, but nothing else. For that we iterate
  * through the, now sorted, array of pending flushes, and look forward to
  * find all neighbouring (or identical) writes.
  /* and finally tell the kernel to write the data to storage */
  reln = smgropen(currlocator, InvalidBackendId);
  smgrwriteback(reln, BufTagGetForkNum(&tag), tag.blockNum, nblocks); -- Trigger kernel
writeback for the supplied range of blocks.
  pgstat_count_io_op_time(IOOBJECT_RELATION, io_context, IOOP_WRITEBACK, io_start,
wb_context->nr_pending);
```

pg_stat_io.fsyncs - статистика в исходном коде называется IOOP_FSYNC, выполняется системным вызовом и увеличивается на 1 по каждому файлу: fsync(**fd**).

статистика pg_stat_io.fsyncs - это всего лишь число файлов данных, в которые checkpointer записывал грязные блоки. За одну контрольную точку выполняется один fsync по одному файлу.

Статистика fsyncs не представляет интереса, так как определяется числом файлов данных. О трудоёмкости выполнения и задержки на получение результата из этой статистики узнать нельзя, так как к моменту вызова fsync(fd) linux мог записать блоки файла из страничного кэша на диск, а мог не записать.

Контрольная точка запустилась в **07:56:35** и начала посылать грязные блоки на запись (writeback). В соответствии со значение параметра checkpoint_completion_target контрольная точка должна writeback грязные блоки за 90% отведённого ей интервала **checkpoint_timeout=**5 минут. Число грязных буферов постепенно уменьшается, в примере **dirty=25364**, через 10 секунд **26780**.

Параметр checkpoint_completion_target не влияет на частоту контрольных точек, он влияет на то, какую часть времени занимает фаза writeback файлов. Если все файлы посланы на запись и активности нет, то контрольная точка по времени не завершится. Контрольная точка завершится либо по checkpoint_timeout, либо если будет вызвана контрольная точка по требованию (команда администратора, команда остановки экземпляра, превышение max_wal_size). После завершения одной контрольной точки, сразу же начинается следующая.

Когда начинается посылка fsync по диапазонам блоков отдельных файлов? Это определяется параметром конфигурации checkpoint_flush_after.

В процессе выполнения контрольной точки обновляются столбцы pg_stat_io.writes и pg_stat_io.writebacks. Значение столбцов pg_stat_io.fsyncs и pg_stat_io.fsync_time в тесте обновлялось после завершения контрольной точки. Но значения pg_stat_io.fsyncs обновляются и в течение контрольной точки. Скорее всего, статистика fsyncs передается в представление один раз - после завершения контрольной точки.

Для протяженной контрольной точки значения **write=23.442 s, sync=79.949 s, total=103.694 s**, а также ее начало (07:56:35.861) и конец (07:58:19.555) **не имеют смысла.**

Пример показателей для разных значений sleep:

```
360 - 1m32.562s (write=269.924 s, sync= 98.376 s, total=368.681 s;) 6:08 duration:  92547.992 ms
310 - 1m39.601s (write=242.044 s, sync= 98.750 s, total=341.174 s;) 5:41 duration:  99587.851 ms
260 - 1m41.145s (write=180.222 s, sync= 99.321 s, total=279.923 s;) 4:40 duration: 101131.358 ms
110 - 1m44.681s (write= 30.342 s, sync=100.731 s, total=131.473 s;) 2:11 duration: 104667.304 ms
    real = duration
```

Значения перед duration это время прошедшее от начала протяженной контрольной точки по времени до ее завершения, оно не имеет смысла, так как контрольная точка по времени работает протяженно во времени, а когда дается команда выполнить контрольную точку по требованию, то контрольная точка по времени дорабатывает без задержек, момент доработки и является окончанием контрольной точки по времени. Значения `sync= 98.376 s` в примере почти одинаковы и соответствует реальной длительности системных вызовов sync_file_range, выполняемых с максимальной скоростью.

Результаты запросов к представлению **pg_stat_io** будут рассматриваться в следующих главах. В этом представлении интересны столбцы `*_time` с метриками времени, но для заполнения этих столбцов нужно менять значение параметра конфигурации. Результаты запросов в примере показывают, что checkpointer отправил *38476* writebacks и *40034* fsyncs (отправляются по диапазонам страниц операционной системы). Число соответствует числу грязных блоков и числу **files=40045**, так как каждая таблица состоит из 2 блоков (в них по 300 строк).

2) Проверьте, что является началом последней контрольной точки по управляющему файлу:

```
postgres@tantor:~$ pg_controldata | grep checkpoint | grep location
Latest checkpoint location:           3B/F9ABE518
Latest checkpoint's REDO location:    3B/F9ABE4D0
```

Значения соответствуют записи **lsn=3B/F9ABE518, redo lsn=3B/F9ABE4D0** в логе.
Начало (**lsn=**) отстоит от конца (**redo lsn=**) не далеко потому, что на кластере не было активности от момента команды checkpoint до ее завершения.

3) Удалите таблицы:

```
postgres@tantor:~$ time psql -f initdrop.sql 2> /dev/null
DO
Time: 6339.328 ms (00:06.339)
```

Удаление прошло заметно быстрее: было **6979 ms** стало **6339 ms**.

Результат этого пункта: после уменьшения `max_connections` удаление таблиц стало работать быстрее. На создание таблиц изменение значения параметра `max_connections` не повлияло. Излишнее увеличение `max_connections` снижает производительность.

4) Этот пункт выполнять необязательно, так как время на выполнение команд большое. Достаточно посмотреть результаты команд. Тестируется как влияет на время выполнения команд увеличение `max_connections` и `max_files_per_process`:

```
echo "сброс параметров в значения по умолчанию"
psql -c "alter system reset max_locks_per_transaction;"
psql -c "alter system reset log_statement;"
psql -c "alter system reset autovacuum_naptime;"
psql -c "alter system reset max_locks_per_transaction;"
psql -c "alter system reset shared_buffers;"
psql -c "alter system reset checkpoint_timeout;"
echo "max_connections=100, max_files_per_process=1000"
psql -c "alter system reset max_connections;"
psql -c "alter system reset max_files_per_process;"
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null
sudo systemctl restart tantor-se-server-16
time psql -f initdrop.sql 2> /dev/null
echo "max_connections=1000, max_files_per_process=1000"
psql -c "alter system set max_connections=1000;"
```

```
psql -c "alter system reset max_files_per_process;"
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null
sudo systemctl restart tantor-se-server-16
time psql -f initdrop.sql 2> /dev/null
echo "max_connections=100, max_files_per_process=40000"
psql -c "alter system reset max_connections;"
psql -c "alter system set max_files_per_process=40000;"
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null
sudo systemctl restart tantor-se-server-16
time psql -f initdrop.sql 2> /dev/null
echo "max_connections=1000, max_files_per_process=40000"
psql -c "alter system set max_connections=1000;"
psql -c "alter system set max_files_per_process=40000;"
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null
sudo systemctl restart tantor-se-server-16
time psql -f initdrop.sql 2> /dev/null
psql -c "alter system reset max_connections;"
psql -c "alter system reset max_files_per_process;"
```

```
сброс параметров в значения по умолчанию
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
max_connections=100, max_files_per_process=1000
ALTER SYSTEM
ALTER SYSTEM

real    3m27.062s
user    0m1.163s
sys     0m0.462s
Timing is on.
DO
Time: 6692.016 ms (00:06.692)

real    0m6.710s
user    0m0.008s
sys     0m0.001s
max_connections=1000, max_files_per_process=1000
ALTER SYSTEM
ALTER SYSTEM

real    3m47.364s
user    0m1.240s
sys     0m0.483s
Timing is on.
DO
Time: 7257.740 ms (00:07.258)

real    0m7.277s
user    0m0.009s
sys     0m0.000s
max_connections=100, max_files_per_process=40000
ALTER SYSTEM
ALTER SYSTEM

real    3m47.401s
user    0m1.259s
sys     0m0.395s
Timing is on.
DO
Time: 6617.566 ms (00:06.618)

real    0m6.636s
user    0m0.009s
sys     0m0.000s
max_connections=1000, max_files_per_process=40000
ALTER SYSTEM
ALTER SYSTEM
```

```
real    3m47.462s
user    0m1.239s
sys     0m0.453s
Timing is on.
DO
Time: 7232.038 ms (00:07.232)

real    0m7.250s
user    0m0.009s
sys     0m0.000s

ALTER SYSTEM
ALTER SYSTEM
```

Увеличение `max_files_per_process с 1000 до 40000` не повлияло на результаты теста.

Практика к главе 12

Часть 1. Параметры команды vacuum

1) Посмотрите сколько транзакций назад замораживалась TOAST-таблицы с названием pg_toast_3394:

```
postgres=# select (select b.oid::regclass from pg_class b where
reltoastrelid=a.oid::regclass) table, relname, relfrozenxid, age(relfrozenxid),
relminmxid, mxid_age(relminmxid) from pg_class a where relfrozenxid<>0 and
relname='pg_toast_3394';
      table      |    relname    | relfrozenxid |   age   | relminmxid | mxid_age
-----------------+---------------+--------------+---------+------------+---------
 pg_init_privs | pg_toast_3394 |          730 | 6748505 |          1 |        0
(1 row)
```

Таблица `pg_toast_3394`, как и ее основная таблица `pg_init_privs` давно не замораживались: 6748505 транзакций, мультитранзакций 1.

2) Выполните заморозку строк в таблице pg_toast_3394:

```
postgres=# vacuum (freeze, verbose) pg_toast.pg_toast_3394;
INFO:  aggressively vacuuming "postgres.pg_toast.pg_toast_3394"
INFO:  finished vacuuming "postgres.pg_toast.pg_toast_3394": index scans: 0
pages: 0 removed, 0 remain, 0 scanned (100.00% of total)
tuples: 0 removed, 0 remain, 0 are dead but not yet removable
removable cutoff: 6749235, which was 0 XIDs old when operation ended
new relfrozenxid: 6749235, which is 6748505 XIDs ahead of previous value
frozen: 0 pages from table (100.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (100.00% of total) had 0 dead item
identifiers removed
avg read rate: 29.260 MB/s, avg write rate: 29.260 MB/s
buffer usage: 25 hits, 1 misses, 1 dirtied
WAL usage: 1 records, 1 full page images, 8177 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

3) Посмотрите как поменялся номер транзакции на которую была выполнена заморозка и возраст заморозки у TOAST и ее основной таблицы:

```
postgres=# select (select b.oid::regclass from pg_class b where
reltoastrelid=a.oid::regclass) table, relname, relfrozenxid, age(relfrozenxid),
relminmxid, mxid_age(relminmxid) from pg_class a where relfrozenxid<>0 and
relname in ('pg_toast_3394', 'pg_init_privs');
      table      |    relname    | relfrozenxid |   age   | relminmxid | mxid_age
-----------------+---------------+--------------+---------+------------+----------
 pg_init_privs | pg_toast_3394 |      6749235 |       0 |          1 |        0
               | pg_init_privs |          730 | 6748505 |          1 |        0
(1 row)
```

У TOAST-таблицы номер транзакции, на которую таблица была заморожена, стал `6749235`, возраст `0`.

Можно замораживать TOAST таблицы отдельно от основных таблиц.

Вакуум с опцией freezee относится к "агрессивному" режиму вакуумирования.

3) К агрессивному режиму вакуумирования относится режим DISABLE_PAGE_SKIPPING.

```
postgres=# vacuum (DISABLE_PAGE_SKIPPING, verbose) pg_init_privs;
INFO:  aggressively vacuuming "postgres.pg_catalog.pg_init_privs"
```

```
INFO:  finished vacuuming "postgres.pg_catalog.pg_init_privs": index scans: 0
pages: 0 removed, 3 remain, 3 scanned (100.00% of total)
tuples: 4 removed, 227 remain, 0 are dead but not yet removable
removable cutoff: 6749235, which was 0 XIDs old when operation ended
new relfrozenxid: 6749235, which is 6748505 XIDs ahead of previous value
frozen: 1 pages from table (33.33% of total) had 4 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
avg read rate: 158.056 MB/s, avg write rate: 135.477 MB/s
buffer usage: 28 hits, 7 misses, 6 dirtied
WAL usage: 7 records, 6 full page images, 46887 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  aggressively vacuuming "postgres.pg_toast.pg_toast_3394"
INFO:  finished vacuuming "postgres.pg_toast.pg_toast_3394": index scans: 0
pages: 0 removed, 0 remain, 0 scanned (100.00% of total)
tuples: 0 removed, 0 remain, 0 are dead but not yet removable
removable cutoff: 6749235, which was 0 XIDs old when operation ended
frozen: 0 pages from table (100.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (100.00% of total) had 0 dead item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 1 hits, 0 misses, 0 dirtied
WAL usage: 0 records, 0 full page images, 0 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

Блоки, отмеченные в карте заморозки по умолчанию пропускаются. Этот параметр **DISABLE_PAGE_SKIPPING** отключает пропуск любых блоков, в том числе которые отмечены в карте заморозки. Параметр может использоваться, если есть подозрения, что карта видимости (в ней хранися карта заморозки) повреждена и карту нужно пересоздать. Также параметр может использоваться для тестов: чтобы проверить сколько будет длиться сканирование всех блоков таблицы.

По умолчанию, TOAST таблица была еще раз вакуумирована и заморожена этой командой. Для отключения вакуумирования TOAST таблиц есть опция: vaccum (process_toast off).

4) Режим DISABLE_PAGE_SKIPPING замораживает строки таблиц:

```
postgres=# select (select b.oid::regclass from pg_class b where
reltoastrelid=a.oid::regclass) table, relname, relfrozenxid, age(relfrozenxid),
relminmxid, mxid_age(relminmxid) from pg_class a where relfrozenxid<>0 and
relname in ('pg_toast_3394', 'pg_init_privs');
      table      |    relname    | relfrozenxid | age | relminmxid | mxid_age
-----------------+---------------+--------------+-----+------------+----------
 pg_init_privs | pg_toast_3394 |      6749235 |   0 |          1 |        0
               | pg_init_privs |      6749235 |   0 |          1 |        0
(2 rows)
```

Зачем вакуумировать или замораживать отдельно таблицы и отдельно TOAST? В TOAST выносятся большие поля и сохраняются в наборе строк TOAST-таблицы. Размер таблицы TOAST может быть намного больше основной таблицы. Обычно поля, хранящиеся в TOAST меняются реже, чем поля, хранящиеся в основной таблице. При появлении новых версий строк в основной таблице новые версии строк в TOAST могут не порождаться.

Все таблицы, обычно, обрабатываются автовакуумом, но можно выполнять вакуумирование отдельно, чтобы следующий цикл автовакуума работал быстрее - пропускал блоки, отмеченные в карте видимости таблицы. Если TOAST-таблиц разрослась, а порог срабатывания автовакуума не превышен, то можно отдельно вакуумировать TOAST-таблицу.

5) Выполните команды:

```
postgres=# select relname, relfrozenxid, relminmxid from pg_class where
relminmxid<>0 and relname = 't3' order by relfrozenxid::text::numeric desc limit
100;
 relname | relfrozenxid | relminmxid
---------+--------------+------------
```

```
 t3       |     21854345 |      108925
(1 row)

postgres=# vacuum (verbose) t3;
INFO:  vacuuming "postgres.public.t3"
INFO:  finished vacuuming "postgres.public.t3": index scans: 0
pages: 0 removed, 0 remain, 0 scanned (100.00% of total)
tuples: 0 removed, 0 remain, 0 are dead but not yet removable, oldest xmin: 21883901
removable cutoff: 21883901, which was 0 XIDs old when operation ended
new relfrozenxid: 21883901, which is 29556 XIDs ahead of previous value
frozen: 0 pages from table (100.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (100.00% of total) had 0 dead item identifiers removed
I/O timings: read: 0.000 ms, write: 0.000 ms
avg read rate: 0.000 MB/s, avg write rate: 65.104 MB/s
buffer usage: 6 hits, 0 misses, 1 dirtied
WAL usage: 1 records, 1 full page images, 7715 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  vacuuming "postgres.pg_toast.pg_toast_3243313"
INFO:  finished vacuuming "postgres.pg_toast.pg_toast_3243313": index scans: 0
pages: 0 removed, 0 remain, 0 scanned (100.00% of total)
tuples: 0 removed, 0 remain, 0 are dead but not yet removable, oldest xmin: 21883901
removable cutoff: 21883901, which was 0 XIDs old when operation ended
new relfrozenxid: 21883901, which is 29556 XIDs ahead of previous value
frozen: 0 pages from table (100.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (100.00% of total) had 0 dead item identifiers removed
I/O timings: read: 0.037 ms, write: 0.000 ms
avg read rate: 104.167 MB/s, avg write rate: 52.083 MB/s
buffer usage: 28 hits, 2 misses, 1 dirtied
WAL usage: 1 records, 1 full page images, 8103 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
postgres=# select relname, relfrozenxid, relminmxid from pg_class where
relminmxid<>0 and relname = 't3' order by relfrozenxid::text::numeric desc limit
100;
 relname | relfrozenxid | relminmxid
---------+--------------+------------
 t3      |     21883901 |      108925
(1 row)
```

Если таблица t3 отсутствует, можно использовать любую другую таблицу.

Команды иллюстрируют, что обычный вакуум и автовакуум по возможности (если обработает все блоки которые не обрабатывал раньше и в сумме окажутся замороженными все строки во всех блоках) выполняет заморозку.

Также, вакуум и автовакуум в конце своей работы обновляют `pg_database.datfrozenxid` - самый старый не замороженный XID в объектах базы данных.

Для получения нового значения для `pg_database.datfrozenxid` выполняется выборка XID по всем relations таблицы **pg_class**, относящихся к базе данных. Если в **pg_class** много строк, то такой запрос нагружает экземпляр. Нагружает потому, что таблица pg_class может активно обновляться. Например, созданием-усечением-удалением-изменением временных и постоянных таблиц. В таблице может быть большое число версий строк, неэффективный индекс, конкуренция за доступ к буферам таблицы в буферном кэше.

6) У команды VACUUM есть опция SKIP_DATABASE_STATS, которая отключает выполнение запроса и обновление значения pg_database.datfrozenxid. Также есть параметр, который ничего не очищает, а только выполняет запрос и обновляет значение:

```
postgres=# \timing on \\
VACUUM (ONLY_DATABASE_STATS VERBOSE);
ERROR:  only_database_stats requires a Boolean value
Time: 0.263 ms
```

Результат этой команды позволяет оценить скорость доступа к **pg_class**. В примере скорость доступа 0,263 миллисекунды и проблем нет.

7) Посмотрите параметры конфигурации автовакуума:

```
postgres=# select name, setting||coalesce(unit,'') unit, context, min_val,
max_val from pg_settings where category='Autovacuum'  or name like '%autovacuum%'
order by 1;
```

```
                name                 |    unit     |  context   | min_val |       max_val
-------------------------------------+-------------+------------+---------+---------------------
 autovacuum                          | on          | sighup     |         |
 autovacuum_analyze_scale_factor     | 0.1         | sighup     | 0       | 100
 autovacuum_analyze_threshold        | 50          | sighup     | 0       | 2147483647
 autovacuum_freeze_max_age           | 10000000000 | postmaster | 100000  | 9223372036854775807
 autovacuum_max_workers              | 5           | postmaster | 1       | 262143
 autovacuum_multixact_freeze_max_age | 20000000000 | postmaster | 10000   | 9223372036854775807
 autovacuum_naptime                  | 60s         | sighup     | 1       | 2147483
 autovacuum_vacuum_cost_delay        | 2ms         | sighup     | -1      | 100
 autovacuum_vacuum_cost_limit        | -1          | sighup     | -1      | 10000
 autovacuum_vacuum_insert_scale_factor | 0.2       | sighup     | 0       | 100
 autovacuum_vacuum_insert_threshold  | 1000        | sighup     | -1      | 2147483647
 autovacuum_vacuum_scale_factor      | 0.2         | sighup     | 0       | 100
 autovacuum_vacuum_threshold         | 50          | sighup     | 0       | 2147483647
 autovacuum_work_mem                 | -1kB        | sighup     | -1      | 2147483647
 log_autovacuum_min_duration         | 600000ms    | sighup     | -1      | 2147483647
(15 rows)
```

На версих PostgreSQL с 32-разрядным счетчиком транзакций:

```
postgres@tantor:~$ psql -h 172.18.0.2 -p 5432 -U postgres
postgres=# select name, setting||coalesce(unit,'') unit, context, min_val,
max_val from pg_settings where category='Autovacuum'  or name like '%autovacuum%'
order by 1;
```

```
                name                 |   unit    |  context   | min_val |  max_val
-------------------------------------+-----------+------------+---------+------------
 autovacuum                          | on        | sighup     |         |
 autovacuum_analyze_scale_factor     | 0.1       | sighup     | 0       | 100
 autovacuum_analyze_threshold        | 50        | sighup     | 0       | 2147483647
 autovacuum_freeze_max_age           | 200000000 | postmaster | 100000  | 2000000000
 autovacuum_max_workers              | 3         | postmaster | 1       | 262143
 autovacuum_multixact_freeze_max_age | 400000000 | postmaster | 10000   | 2000000000
 autovacuum_naptime                  | 60s       | sighup     | 1       | 2147483
 autovacuum_vacuum_cost_delay        | 2ms       | sighup     | -1      | 100
 autovacuum_vacuum_cost_limit        | -1        | sighup     | -1      | 10000
 autovacuum_vacuum_insert_scale_factor | 0.2     | sighup     | 0       | 100
 autovacuum_vacuum_insert_threshold  | 1000      | sighup     | -1      | 2147483647
 autovacuum_vacuum_scale_factor      | 0.2       | sighup     | 0       | 100
 autovacuum_vacuum_threshold         | 50        | sighup     | 0       | 2147483647
 autovacuum_work_mem                 | -1kB      | sighup     | -1      | 2147483647
 log_autovacuum_min_duration         | 600000ms  | sighup     | -1      | 2147483647
(15 rows)
```

Часть 2. Наблюдение за вакуумом

1) Создайте 10000 таблиц.Для этого выполните команды:

```
postgres@tantor:~$ time psql -f init1.sql > /dev/null

real    3m45.019s
user    0m1.262s
sys     0m0.438s
```

2) Выполните команды:

```
postgres@tantor:~$
time psql -c "vacuum (DISABLE_PAGE_SKIPPING);" &
time psql -c "vacuum (DISABLE_PAGE_SKIPPING);" &
time psql -c "vacuum (DISABLE_PAGE_SKIPPING);" &

for (( i=1; i <= 10; i+=1 ))
do
sleep 0.7
```

```
  psql -c "select pid, relid::regclass, phase, heap_blks_total total,
heap_blks_scanned scanned, max_dead_tuples max_dead, num_dead_tuples dead from
pg_stat_progress_vacuum;"
  done
[1] 142424
[2] 142425
[3] 142427
  pid   |    relid     |     phase     | total | scanned | max_dead | dead
--------+--------------+---------------+-------+---------+----------+------
 142431 | pg_attribute | scanning heap |  3757 |    1242 |  1093287 | 2562
 142432 | bencher      | scanning heap | 54055 |    2805 |      291 |    0
 142433 | t2           | scanning heap |  3847 |    1202 |      291 |    0
(3 rows)

  pid   |  relid  |     phase     | total | scanned | max_dead | dead
--------+---------+---------------+-------+---------+----------+------
 142432 | bencher | scanning heap | 54055 |   17732 |      291 |    0
(1 row)

  pid   |        relid         |     phase     | total | scanned | max_dead | dead
--------+----------------------+---------------+-------+---------+----------+------
 142431 | bencher              | scanning heap | 54055 |    7192 |      291 |    0
 142432 | test.film_summary561 | scanning heap |     2 |       2 |      291 |    0
(2 rows)

  pid   |         relid         |          phase           | total | scanned | max_dead | dead
--------+-----------------------+--------------------------+-------+---------+----------+------
 142431 | bencher               | scanning heap            | 54055 |   21602 |      291 |    0
 142432 | test.film_summary2137 | performing final cleanup |     2 |       2 |      291 |    0
(2 rows)


  pid   |           relid            |     phase     | total | scanned | max_dead | dead
--------+----------------------------+---------------+-------+---------+----------+------
 142431 | pg_toast.pg_toast_7245268 | initializing  |     0 |       0 |        0 |    0
 142432 | test.film_summary7072     | scanning heap |     2 |       0 |      291 |    0
 142433 | bencher                   | scanning heap | 54055 |    7625 |      291 |    0
(3 rows)

postgres@tantor:~$ VACUUM

real    0m7.666s
user    0m0.009s
sys     0m0.000s
VACUUM

real    0m10.108s
user    0m0.010s
sys     0m0.000s
VACUUM

real    0m12.971s
user    0m0.011s
sys     0m0.000s
```

Были одновременно запущены три вакуума по базе данных. Опция **DISABLE_PAGE_SKIPPING** использовалась, чтобы вакуум сканировали все блоки таблиц, то есть работали какое-то время.

Периодичесие обращения к представлению **pg_stat_progress_vacuum** показывают какие процессы вакуума или автовакуума в момент запроса выполняют вакуумирование. В примере видно, что последовательно 3,1,2,2,3 команды обрабатывали объекты.

Для получения имени таблицы удобно использовать приведение типов: **relid::regclass**.

3) Нажмите на клавиатуре клавишу <Enter>, чтобы вернуть промпт:

```
<Enter>
[1]   Done                    time psql -c "vacuum (DISABLE_PAGE_SKIPPING);"
[2]-  Done                    time psql -c "vacuum (DISABLE_PAGE_SKIPPING);"
[3]+  Done                    time psql -c "vacuum (DISABLE_PAGE_SKIPPING);"

postgres@tantor:~$
```

Представление **pg_stat_progress_vacuum** не показывает работу команды VACUUM FULL. Работу команды VACUUM_FULL показывает представление pg_stat_progress_cluster:

```
postgres=# select * from pg_stat_progress_cluster\gx
-[ RECORD 1 ]-------+-------------
pid                 | 140712
datid               | 5
datname             | postgres
relid               | 7251678
command             | VACUUM FULL
phase               | initializing
cluster_index_relid | 0
heap_tuples_scanned | 0
heap_tuples_written | 0
heap_blks_total     | 0
heap_blks_scanned   | 0
index_rebuild_count | 0
```

VACUUM FULL является разновидностью команды CLUSTER.

4) Не удобно то, что не показывается время, которое длится команда и нет данных не заблокирован ли процесс. Для получения этих данных можно соединить запрос с представлением `pg_stat_activity`. Выполните команды:

```
postgres@tantor:~$
time psql -c "vacuum (DISABLE_PAGE_SKIPPING);" &
sleep 3

psql -c "SELECT p.pid, clock_timestamp() - a.xact_start AS duration,
coalesce(wait_event_type ||'.'|| wait_event, 'f') AS waiting,
CASE
WHEN a.query ~*'^autovacuum.*to prevent wraparound' THEN 'wraparound'
WHEN a.query ~*'^vacuum' THEN 'user' ELSE 'regular' END AS mode,
p.datname AS database, p.relid::regclass AS table, p.phase,
pg_size_pretty(p.heap_blks_total * current_setting('block_size')::int) AS
table,
pg_size_pretty(pg_total_relation_size(relid)) AS total,
pg_size_pretty(p.heap_blks_scanned * current_setting('block_size')::int) AS
scanned,
pg_size_pretty(p.heap_blks_vacuumed * current_setting('block_size')::int) AS
vacuumed,
round(100.0 * p.heap_blks_scanned / p.heap_blks_total, 1) AS scan_pct,
round(100.0 * p.heap_blks_vacuumed / p.heap_blks_total, 1) AS vacuum_pct,
p.index_vacuum_count Loops,
round(100.0 * p.num_dead_tuples / p.max_dead_tuples,1) AS dead_pct
FROM pg_stat_progress_vacuum p
JOIN pg_stat_activity a using (pid)
ORDER BY clock_timestamp()-a.xact_start desc;"
```

```
 pid   |duration|waiting|mode|database|  table   |      phase      | table  | total  | scanned | vacuumed | scan_pct | vacuum_pct | loops | dead_pct
-------+--------+-------+----+--------+----------+-----------------+--------+--------+---------+----------+----------+------------+-------+----------
142669|00:00:03| f     |user|postgres| bencher | scanning heap | 422 MB | 422 MB | 402 MB  | 0 bytes  |     95.2 |        0.0 |     0 |      0.0
(1 row)
```

5) Выполните запрос, который показывает требуется ли вакуумировать таблицу или анализировать в соответствии с параметрами конфигурации:

```
postgres=# WITH s AS (
 SELECT
  current_setting('autovacuum_analyze_scale_factor')::float8 AS analyze_factor,
  current_setting('autovacuum_analyze_threshold')::float8    AS analyze_threshold,
```

```
  current_setting('autovacuum_vacuum_scale_factor')::float8  AS vacuum_factor,
  current_setting('autovacuum_vacuum_threshold')::float8     AS vacuum_threshold
)
SELECT nspname, relname, n_dead_tup, v_threshold, n_mod_since_analyze, a_threshold,
  CASE WHEN n_dead_tup > v_threshold THEN 'yes' ELSE 'no' END AS do_vacuum,
  CASE WHEN n_mod_since_analyze > a_threshold THEN 'yes' ELSE 'no' END AS do_analyze,
  pg_relation_size(relid) AS relsize,
  pg_total_relation_size(relid) AS total
FROM (SELECT n.nspname, c.relname, c.oid AS relid, t.n_dead_tup, t.n_mod_since_analyze,
  trunc(c.reltuples * s.vacuum_factor + s.vacuum_threshold) AS v_threshold,
  trunc(c.reltuples * s.analyze_factor + s.analyze_threshold) AS a_threshold
 FROM s, pg_class c
  JOIN pg_namespace n ON c.relnamespace = n.oid
  JOIN pg_stat_all_tables t ON c.oid = t.relid
 WHERE c.relkind = 'r')
WHERE n_dead_tup > v_threshold OR n_mod_since_analyze > a_threshold
ORDER BY nspname, relname limit 5;
```

```
  nspname   |     relname     | n_dead_tup | v_threshold | n_mod_since_analyze | a_threshold | do_vacuum | do_analyze |  relsize  |   total
------------+-----------------+------------+-------------+---------------------+-------------+-----------+------------+-----------+-----------
 pg_catalog | pg_statistic    |          0 |         142 |               19709 |          96 | no        | yes        |    229376 |    376832
 public     | bencher         |          0 |     2000050 |            60000000 |     1000050 | no        | yes        | 442818560 | 442966016
 test       | film_summary1   |          0 |         110 |                 301 |          80 | no        | yes        |     16384 |     57344
 test       | film_summary10  |          0 |         110 |                 301 |          80 | no        | yes        |     16384 |     57344
 test       | film_summary100 |          0 |         110 |                 301 |          80 | no        | yes        |     16384 |     57344
(5 rows)
```

Запрос не учитывает настройки на уровне таблиц, только параметры уровня кластера, поэтому список таблиц, которые будут вакуумированы или анализированы не точный.

6) Посмотрите пример команд:

```
postgres=# VACUUM FULL FREEZE VERBOSE ANALYZE t3;
INFO:  vacuuming "public.t3"
INFO:  "public.t3": found 0 removable, 0 nonremovable row versions in 0 pages
DETAIL:  0 dead row versions cannot be removed yet.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  analyzing "public.t3"
INFO:  "t3": scanned 0 of 0 pages, containing 0 live rows and 0 dead rows; 0 rows in sample, 0 estimated
total rows
VACUUM
postgres=# VACUUM VERBOSE FULL t3;
ERROR:  syntax error at or near "FULL"
LINE 1: VACUUM VERBOSE FULL t3;
                       ^
postgres=# VACUUM (VERBOSE, FULL) t3;
INFO:  vacuuming "public.t3"
INFO:  "public.t3": found 0 removable, 0 nonremovable row versions in 0 pages
DETAIL:  0 dead row versions cannot be removed yet.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

Команда без круглых скобок требует указания параметров только в том порядке, который указан в первом примере. При использовании круглых скобок и запятых порядок не важен.

Регистр ключевых слов не важен.

7) Удалите созданные таблицы, чтобы они не занимали место:

```
postgres@tantor:~$ psql -f initdrop.sql 2> /dev/null
```

Часть 3. Расширение для просмотра карты видимости и заморозки pg_visibility

1) Установите расширение:

```
postgres=# create extension pg_visibility;
CREATE EXTENSION
```

Позволяет просматривать карту видимости и карту заморозки и пересоздавать их.
Эти карты хранятся в файле слоя vm.

2) Посмотрите какие объекты есть в составе расширения:

```
postgres=# \dx+ pg_visibility
     Objects in extension "pg_visibility"
              Object description
----------------------------------------------
 function pg_check_frozen(regclass)
 function pg_check_visible(regclass)
 function pg_truncate_visibility_map(regclass)
 function pg_visibility_map(regclass)
 function pg_visibility_map(regclass,bigint)
 function pg_visibility_map_summary(regclass)
 function pg_visibility(regclass)
 function pg_visibility(regclass,bigint)
(8 rows)
```

В расширении есть 8 функций.

3) Создайте временную таблицу и посмотрите как это повлияет на карту видимости таблицы pg_class:

```
postgres=# select * from pg_visibility_map_summary('pg_class');
create temp table temp1 (n numeric);
select * from pg_visibility_map_summary('pg_class');
drop table temp1;
 all_visible | all_frozen
-------------+------------
         792 |          6
(1 row)

CREATE TABLE
 all_visible | all_frozen
-------------+------------
         790 |          6
(1 row)

DROP TABLE
```

Карта видимости изменилась. Число блоков, в которых все строки видимы уменьшилось.

4) Выполните запрос:

```
postgres=# select pd_all_visible, count(all_visible) filter (where
all_visible=true) true, count(all_visible) filter (where all_visible=false) false
from pg_visibility('pg_class') group by pd_all_visible;
 pd_all_visible | true | false
----------------+------+-------
 f              |    0 |    27
 t              |  786 |     0
(2 rows)
```

Используя этот запрос можно получить детальную информацию по блокам таблицы из карты видимости. Группировка выполняется по биту PD_ALL_VISIBLE. Функция читает не только карту видимости, но и все блоки таблицы, так как бит PD_ALL_VISIBLE есть только в заголовках блоков.

Есть похожая функция, в которой отсутствует **pd_all_visible** и которая использует карту видимости: pg_visibility_map().

5) Выполните команды:

```
  postgres=# select pd_all_visible, count(all_visible) filter (where
all_visible=true) true, count(all_visible) filter (where all_visible=false) false
from pg_visibility('pgbench_accounts') group by pd_all_visible;
  select pg_truncate_visibility_map('pgbench_accounts');
  select pd_all_visible, count(all_visible) filter (where all_visible=true) true,
count(all_visible) filter (where all_visible=false) false from
pg_visibility('pgbench_accounts') group by pd_all_visible;
  vacuum pgbench_accounts;
  select pd_all_visible, count(all_visible) filter (where all_visible=true) true,
count(all_visible) filter (where all_visible=false) false from
pg_visibility('pgbench_accounts') group by pd_all_visible;

   pd_all_visible | true | false
  ----------------+------+-------
   t              | 1640 |     0
  (1 row)

   pg_truncate_visibility_map
  ----------------------------

  (1 row)

   pd_all_visible | true | false
  ----------------+------+-------
   t              |    0 |  1640
  (1 row)

  VACUUM
   pd_all_visible | true | false
  ----------------+------+-------
   t              | 1640 |     0
  (1 row)
```

Функция **pg_truncate_visibility_map()** усекла карту видимости. После усечения все блоки считались **не-**"all_visible". Команда VACUUM просканировала **все блоки** таблицы и пересоздала карту видимости.

Часть 4. Интервал между циклами автовакуума, параметр autovacuum_naptime

1) Выполните команды:

```
postgres@tantor:~$
rm -f $PGDATA/log/*
psql -c "alter system set log_autovacuum_min_duration = 0;"
psql -c "alter system set logging_collector = on;"
psql -c "alter system set autovacuum_naptime = '3s';"
sudo systemctl restart tantor-se-server-16
psql -c "drop table if exists test;"
time psql -c "CREATE TABLE test AS SELECT * FROM generate_series(1, 10000000) x(id);"
time psql -c "CREATE INDEX ON test(id);"
cat $PGDATA/log/postgresql-* | tail -3
sleep 35
cat $PGDATA/log/postgresql-* | tail -20
ALTER SYSTEM
ALTER SYSTEM
```

```
ALTER SYSTEM
DROP TABLE
SELECT 10000000

real    0m14.780s
user    0m0.009s
sys     0m0.000s
CREATE INDEX

real    0m8.712s
user    0m0.009s
sys     0m0.000s
00:19:25.571 MSK [174830] LOG:  skipping vacuum of "test" --- lock not available
00:19:28.573 MSK [174831] LOG:  skipping vacuum of "test" --- lock not available
00:19:31.575 MSK [174832] LOG:  skipping vacuum of "test" --- lock not available
00:19:10.454 MSK [174811] LOG:  database system is ready to accept connections
00:19:25.571 MSK [174830] LOG:  skipping vacuum of "test" --- lock not available
00:19:28.573 MSK [174831] LOG:  skipping vacuum of "test" --- lock not available
00:19:31.575 MSK [174832] LOG:  skipping vacuum of "test" --- lock not available
00:20:00.777 MSK [174835] LOG:  automatic vacuum of table "postgres.public.test": index scans: 0
  pages: 0 removed, 44248 remain, 44248 scanned (100.00% of total)
  tuples: 0 removed, 10000000 remain, 0 are dead but not yet removable, oldest xmin: 22095797
  removable cutoff: 22095797, which was 0 XIDs old when operation ended
  frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
  index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
     I/O timings: read: 419.394 ms, write: 529.628 ms
     avg read rate: 12.566 MB/s, avg write rate: 12.567 MB/s
     buffer usage: 46403 hits, 42137 misses, 42141 dirtied
     WAL usage: 44252 records, 5 full page images, 2917445 bytes
     system usage: CPU: user: 2.19 s, system: 1.68 s, elapsed: 26.19 s
00:20:02.498 MSK [174835] LOG:  automatic analyze of table "postgres.public.test"
     I/O timings: read: 202.989 ms, write: 0.289 ms
     avg read rate: 129.678 MB/s, avg write rate: 0.014 MB/s
     buffer usage: 1526 hits, 28583 misses, 3 dirtied
     system usage: CPU: user: 0.37 s, system: 0.46 s, elapsed: 1.72 s
```

Команды устанавливают вывод всех сообщений автовакуума в диагностический лог, интервал запуска циклов автовакуума 3 секунды.

2) Была создана таблица с большим числом строк. В соотвествии с параметрами:

```
postgres=# \dconfig *autovac*inse*
       List of configuration parameters
              Parameter               | Value
---------------------------------------+-------
 autovacuum_vacuum_insert_scale_factor | 0.2
 autovacuum_vacuum_insert_threshold    | 1000
(2 rows)
```

рабочий процесс [174830] запустился в течение 3 секунд после создания таблицы.

3) Рабочие процессы постоянно не работают, постоянно работает только autovacuum launcher:

```
ps -ef | grep vacuum
postgres    1019     929  0 Jan12 ?        00:00:01 postgres: 15/main: autovacuum launcher
postgres  174817  174811  0 00:19 ?        00:00:00 postgres: autovacuum launcher
postgres  175144  140533  0 00:31 pts/0    00:00:00 grep vacuum
```

Один из рабочих процессов направляется в базу данных с таблицей test пытается её вакуумировать. Таблица была заблокирована командой создания индекса на 8.712s.

в соответствии с параметром раз в 3 секунды запускался новый процесс autovacuum worker и сообщал о том, что пытался вакуумировать таблицу, но не смог и пропустил её, после чего останавливался. Таких сообщений может быть 2 или 3, так как цикл запуска автовакуума может начаться в любой момент от нуля до 3 секунд после создания таблицы. Может быть и одно сообщение, если бы процесс ваккумировал дольше 8 секунд. В данном примере, кроме таблицы

test ничего не должно было вакуумироваться и процессы автовакуума после запуска тут же останавливальсь.

В четвертый раз, в 00:19:**31** + 3 секунды запустился autovacuum worker с номером процесса [**174835**] он вакуумировал и проанализировал таблицу, отработал **26.19 s** + **1.72 s** и создал сообщение в логе через 26 секунд, в **00:20:00**.

Часть 5. Сравнение вакуума 17 версии с предыдущими версиями

Алгоритм работы вакуума и автовакуума существенно улучшился начиная с 17 версии. В виртуальной машине в контейнере docker был установлен PostgreSQL 17 версии. Проверим какие показатели вакуума улучшились.

1) Подсоединитесь к PostgreSQL 17 в контейнере:

```
postgres@tantor:~$
sudo docker restart postgres
sudo docker inspect postgres | grep 172
                  "Gateway": "172.18.0.1",
                  "IPAddress": "172.18.0.2",
```

Если команда ничего не выдала перейдите к следующему пункту.
Если команда выдала IPAdderess подсоединитесь по нему:

```
postgres@tantor:~$ psql -h 172.18.0.2 -p 5432 -U postgres
psql (16.2, server 17.2 (Debian 17.2-1.pgdg120+1))
WARNING: psql major version 16, server major version 17.
         Some psql features might not work.
Type "help" for help.
postgres=# \timing on
Timing is on.
```

Если подсоединение прошло успешно, пропустите следующий пункт и перейдите к 3 пункту.

2) Команды для пересоздания контейнера, если в предыдущем пункте не удаслось подсоединиться к PostgreSQL 17:

```
postgres@tantor:~$
sudo docker rm -f postgres
sudo docker network create postgres
sudo docker run -d --init  --network postgres -e DB_HOST=127.0.0.1 -e
POSTGRES_USER=postgres -e POSTGRES_PASSWORD=postgres -e POSTGRES_INITDB_ARGS="--
data-checksums" -e POSTGRES_HOST_AUTH_METHOD=trust -p 5434:5434 -e
PGDATA=/var/lib/postgresql/data -d -v /root/data:/var/lib/postgresql/data --name
postgres postgres
sudo docker inspect postgres | grep 172
                  "Gateway": "172.18.0.1",
                  "IPAddress": "172.18.0.2",
postgres@tantor:~$ psql -h 172.18.0.2 -p 5432 -U postgres
psql (16.2, server 17.2 (Debian 17.2-1.pgdg120+1))
WARNING: psql major version 16, server major version 17.
         Some psql features might not work.
Type "help" for help.
postgres=# \timing on
Timing is on.
```

3) Выполните тест:

```
postgres=# alter system set max_wal_size = '2GB';
alter system set checkpoint_timeout='30min';
select pg_reload_conf();
```

```
  drop table if exists test;
  create table test with (autovacuum_enabled=off) as select * from
generate_series(1, 10000000) x(id);
  create index on test(id);
  update test set id = id - 1;
  checkpoint;
  vacuum verbose test;
  drop table test;
```

Увеличение **max_wal_size** нужно, чтобы в процессе теста не выполнилась контрольная точка по требованию и не нагрузила ввод-вывод, что повлияет на показатели.

Таблица создается с отключенным автовакуумом для простоты получения результата теста: командой vacuum verbose. С автовакуумом результат был бы тот же, только его нужно было смотреть в диагностическом журнале кластера.

Результат выполнения команд:

```
  ALTER SYSTEM
  Time: 23.768 ms
  ALTER SYSTEM
  Time: 19.945 ms
   pg_reload_conf
  ----------------
   t
  (1 row)

  Time: 4.246 ms
  DROP TABLE
  Time: 153.615 ms
  SELECT 10000000
  Time: 13811.083 ms (00:13.811)
  CREATE INDEX
  Time: 7457.785 ms (00:07.458)
  UPDATE 10000000
  Time: 67207.156 ms (01:07.207)
  CHECKPOINT
  Time: 3631.462 ms (00:03.631)
  INFO:  vacuuming "postgres.public.test"
  INFO:  finished vacuuming "postgres.public.test": index scans: 1
  pages: 0 removed, 88535 remain, 88535 scanned (100.00% of total)
  tuples: 10000000 removed, 10000000 remain, 0 are dead but not yet removable
  removable cutoff: 757, which was 0 XIDs old when operation ended
  new relfrozenxid: 757, which is 3 XIDs ahead of previous value
  frozen: 44249 pages from table (49.98% of total) had 10000000 tuples frozen
  index scan needed: 44248 pages from table (49.98% of total) had 10000000 dead item identifiers
removed
  index "test_id_idx": pages: 54840 in total, 0 newly deleted, 0 currently deleted, 0 reusable
  avg read rate: 45.658 MB/s, avg write rate: 49.977 MB/s
  buffer usage: 106711 hits, 169524 misses, 185560 dirtied
  WAL usage: 364397 records, 143159 full page images, 1005718014 bytes
  system usage: CPU: user: 8.41 s, system: 1.05 s, elapsed: 29.00 s
  VACUUM
  Time: 29007.776 ms (00:29.008)
  DROP TABLE
   data_checksums
  ----------------
   on
  (1 row)
```

4) Подсоединитесь к PostgreSQL 16 или более ранней версии, которые установлены в виртуальной машине:

```
  postgres@tantor:~$ sudo systemctl restart postgresql
  postgres@tantor:~$ sudo systemctl restart tantor-se-server-16
  postgres@tantor:~$ psql -p 5435
  psql (16.2, server 15.6 (Debian 15.6-astra.se2))
  Type "help" for help.
  postgres=# \timing on
  Timing is on.
```

5) Выполните тест:

```
postgres=# alter system set max_wal_size = '2GB';
alter system set checkpoint_timeout='30min';
select pg_reload_conf();
drop table if exists test;
create table test with (autovacuum_enabled=off) as select * from
generate_series(1, 10000000) x(id);
create index on test(id);
update test set id = id - 1;
checkpoint;
vacuum verbose test;
drop table test;
show data_checksums;
ALTER SYSTEM
Time: 36.365 ms
ALTER SYSTEM
Time: 21.414 ms
 pg_reload_conf
----------------
 t
(1 row)

Time: 3.999 ms
ОШИБКА:  таблица "test" не существует
Time: 1.948 ms
SELECT 10000000
Time: 19315.446 ms (00:19.315)
CREATE INDEX
Time: 13423.928 ms (00:13.424)
UPDATE 10000000
Time: 81646.528 ms (01:21.647)
CHECKPOINT
Time: 3838.599 ms (00:03.839)
ИНФОРМАЦИЯ:  очистка "postgres.public.test"
ИНФОРМАЦИЯ:  закончена очистка "postgres.public.test": сканирований индекса: 1
страниц удалено: 0, осталось: 108109, просканировано: 108109 (100.00% от общего числа)
версий строк: удалено: 10000000, осталось: 10000000, «мёртвых», но ещё не подлежащих удалению: 0
XID отсечки удаления: 1098324, на момент завершения операции он имел возраст: 0 XID
новое значение relfrozenxid: 1098323, оно продвинулось вперёд от предыдущего значения на 2 XID
сканирование индекса требуется: на страницах таблицы (54055, 50.00% от общего числа) удалено
мёртвых идентификаторов элементов: 10000000
индекс "test_id_idx": всего страниц: 77011, сейчас удалено: 0, удалено на данный момент: 0,
свободно: 0
средняя скорость чтения: 37.580 МБ/с, средняя скорость записи: 40.359 МБ/с
использование буфера: попаданий: 126331, промахов: 221000, «грязных» записей: 237341
использование WAL: записей: 292850, полных образов страниц: 130689, байт: 362929097
нагрузка системы: CPU: пользов.: 6.53 с, система: 1.09 с, прошло: 45.94 с
VACUUM
Time: 45947.234 ms (00:45.947)
DROP TABLE
Time: 218.371 ms
 data_checksums
----------------
 off
(1 row)
```

В примере подсчет контрольных сумм выключен, поэтому сравнивать результаты тестов с кластерами с включенными контрольными суммами нельзя: вакуум записывает в WAL в 2.77 раза меньше `байт: 362929097` , а не `1005718014 bytes`. Даже при этом вакуум версии 17 отработал быстрее: за `29007.776 ms`, по сравнению с 15 версией, которая отработала за `45947.234 ms`.

Из теста следует, что 17 версия в докере выполняла команды **в целом быстрее**, чем 15 версия **не** в докере.

6) Выполните тест, подсоединившись к кластеру с включенными контрольными суммами:

```
postgres@tantor:~$ sudo systemctl stop postgresql
```

```
postgres@tantor:~$ sudo systemctl restart tantor-se-server-16
postgres@tantor:~$ psql
postgres=# \timing on \\
alter system set max_wal_size = '2GB';
alter system set checkpoint_timeout='30min';
select pg_reload_conf();
drop table if exists test;
create table test with (autovacuum_enabled=off) as select * from
generate_series(1, 10000000) x(id);
create index on test(id);
update test set id = id - 1;
checkpoint;
vacuum verbose test;
drop table test;
show data_checksums;

ALTER SYSTEM
Time: 36.604 ms
ALTER SYSTEM
Time: 22.717 ms
 pg_reload_conf
----------------
 t
(1 row)

Time: 3.678 ms
NOTICE:  table "test" does not exist, skipping
DROP TABLE
Time: 2.644 ms
SELECT 10000000
Time: 14866.174 ms (00:14.866)
CREATE INDEX
Time: 10817.373 ms (00:10.817)
UPDATE 10000000
Time: 80073.956 ms (01:20.074)
CHECKPOINT
Time: 3870.459 ms (00:03.870)
INFO:  vacuuming "postgres.public.test"
INFO:  finished vacuuming "postgres.public.test": index scans: 1
pages: 0 removed, 88496 remain, 88496 scanned (100.00% of total)
tuples: 10000000 removed, 10000000 remain, 0 are dead but not yet removable, oldest xmin:
22095818
removable cutoff: 22095818, which was 0 XIDs old when operation ended
new relfrozenxid: 22095818, which is 3 XIDs ahead of previous value
frozen: 44249 pages from table (50.00% of total) had 10000000 tuples frozen
index scan needed: 44248 pages from table (50.00% of total) had 10000000 dead item identifiers
removed
index "test_id_idx": pages: 54840 in total, 0 newly deleted, 0 currently deleted, 0 reusable
I/O timings: read: 2326.127 ms, write: 1499.600 ms
avg read rate: 21.517 MB/s, avg write rate: 23.612 MB/s
buffer usage: 106917 hits, 169199 misses, 185676 dirtied
WAL usage: 364398 records, 143159 full page images, 1008665968 bytes
system usage: CPU: user: 11.12 s, system: 6.48 s, elapsed: 61.43 s
VACUUM
Time: 61437.534 ms (01:01.438)
DROP TABLE
Time: 181.806 ms
 data_checksums
----------------
 on
(1 row)

Time: 0.117 ms
```

Из теста следует, что 17 версия в докере выполняла команды в целом быстрее, чем 16 версия **не** в докере. Контрольная точка выполнялась в докере немного быстрее `3631.462 ms` против `3870.459 ms` в обеих СУБД вне докера. Время выполнения вакуума 17 версии в докере: `29.00 s` против `61.43 s` в 16 версии вне докера. Вероятно, работа в докере меняет взаимодействие с

вводом-выводом. Меньшее расхождение во времени выполнения контрольных точек позволяет надеяться, что работа в докере не ухудшает отказоустойчивость.

Результат этого теста: сравнивать СУБД, работающие в докере и вне докера нельзя. Для сравнения нужно одинаковое окружение, в котором выполняются программы и одинаковые настройки.

Например, включение контрольных суммы в 15 версии можно сравнить работу вакуума с 16 версией.

7) Устраним недостаток предыдущего теста. Включите подсчет контрольных сумм для блоков данных у кластера 15 версии и повторите тест:

```
postgres@tantor:~$
sudo systemctl stop postgresql
/usr/lib/postgresql/15/bin/pg_checksums -e -D /var/lib/postgresql/15/main
sudo systemctl start postgresql
psql -p 5435
Checksum operation completed
Files scanned:   1262
Blocks scanned:  6290
Files written:  1045
Blocks written: 6290
pg_checksums: syncing data directory
pg_checksums: updating control file
Checksums enabled in cluster
postgres@tantor:~$ psql -p 5435
psql (16.2, server 15.6 (Debian 15.6-astra.se2))
Type "help" for help.
postgres=# \timing on \\
drop table if exists test;
create table test with (autovacuum_enabled=off) as select * from
generate_series(1, 10000000) x(id);
create index on test(id);
update test set id = id - 1;
checkpoint;
vacuum verbose test;
drop table test;

SELECT 10000000
Time: 15648.954 ms (00:15.649)
CREATE INDEX
Time: 9550.452 ms (00:09.550)
UPDATE 10000000
Time: 74725.766 ms (01:14.726)
CHECKPOINT
Time: 4463.721 ms (00:04.464)
ИНФОРМАЦИЯ:  очистка "postgres.public.test"
ИНФОРМАЦИЯ:  закончена очистка "postgres.public.test": сканирований индекса: 1
страниц удалено: 0, осталось: 108109, просканировано: 108109 (100.00% от общего числа)
версий строк: удалено: 10000000, осталось: 10000000, «мёртвых», но ещё не подлежащих удалению: 0
XID отсечки удаления: 1098418, на момент завершения операции он имел возраст: 0 XID
новое значение relfrozenxid: 1098417, оно продвинулось вперёд от предыдущего значения на 2 XID
сканирование индекса требуется: на страницах таблицы (54055, 50.00% от общего числа) удалено
мёртвых идентификаторов элементов: 10000000
индекс "test_id_idx": всего страниц: 77011, сейчас удалено: 0, удалено на данный момент: 0,
свободно: 0
средняя скорость чтения: 26.511 МБ/с, средняя скорость записи: 28.472 МБ/с
использование буфера: попаданий: 126332, промахов: 220999, «грязных» записей: 237349
использование WAL: записей: 400976, полных образов страниц: 184760, байт: 1229390628
нагрузка системы: CPU: пользов.: 7.23 с, система: 1.90 с, прошло: 65.12 с
VACUUM
Time: 65130.002 ms (01:05.130)
```

Объем записанных данных `1229386288` больше `1005718014 bytes` потому, что размер заголовка каждой строки в Astralinux PostgreSQL больше на 8 байт, которые используются для хранения метки доступа. Время ваккумирования `65.12 с` с учетом поправки на объем записи

соответствует 61.43 s. Изменений в коде вакуумирования между 15 и 16 версиями не было. Увеличение длительности контрольной точки пропорционально увеличению объема журнальных записей (с точностью до разброса результатов тестов, если несколько раз повторить тесты): 4463.721 ms / 3838.599 ms = 1.1628, 1229390628 / 1005718014 = 1.2224.

Часть 6. Число сканирований индексов при вакуумировании

Основное преимущество вакуума 17 версии не в уменьшении времени работы, а в существенно меньших: потреблении памяти, вероятности повторного прохода по всем индексам таблицы. Установим для вакуума 2Мб памяти и проверим сколько вакуум сделает проходов по индексу в 17 версии и предыдущей.

1) Подсоединитесь к 17 версии PosqgreSQL:

```
postgres@tantor:~$ psql -h 172.18.0.2 -p 5432 -U postgres
psql (16.2, server 17.2 (Debian 17.2-1.pgdg120+1))
WARNING: psql major version 16, server major version 17.
         Some psql features might not work.
Type "help" for help.
postgres=#
```

2) Выполните тест с уменьшенным размером памяти на вакуумирование:

```
postgres=# \timing on \\
alter system set maintenance_work_mem='2MB';
select pg_reload_conf();
drop table if exists test;
create table test with (autovacuum_enabled=off) as select * from
generate_series(1, 10000000) x(id);
create index on test(id);
update test set id = id - 1;
checkpoint;
vacuum verbose test;
drop table test;
alter system reset maintenance_work_mem;
alter system reset max_wal_size;
select pg_reload_conf();
checkpoint;
ALTER SYSTEM
Time: 24.098 ms
 pg_reload_conf
----------------
 t
(1 row)

Time: 1.005 ms
SELECT 10000000
Time: 13332.139 ms (00:13.332)
CREATE INDEX
Time: 11931.761 ms (00:11.932)
UPDATE 10000000
Time: 74425.280 ms (01:14.425)
CHECKPOINT
Time: 4748.177 ms (00:04.748)
INFO:  vacuuming "postgres.public.test"
INFO:  finished vacuuming "postgres.public.test": index scans: 2
pages: 0 removed, 88535 remain, 88535 scanned (100.00% of total)
tuples: 10000000 removed, 10000000 remain, 0 are dead but not yet removable
removable cutoff: 765, which was 0 XIDs old when operation ended
new relfrozenxid: 765, which is 3 XIDs ahead of previous value
frozen: 44249 pages from table (49.98% of total) had 10000000 tuples frozen
```

```
   index scan needed: 44248 pages from table (49.98% of total) had 10000000 dead item identifiers
removed
   index "test_id_idx": pages: 54840 in total, 0 newly deleted, 0 currently deleted, 0 reusable
   avg read rate: 57.308 MB/s, avg write rate: 49.041 MB/s
   buffer usage: 114465 hits, 216614 misses, 185368 dirtied
   WAL usage: 364398 records, 143159 full page images, 1005723308 bytes
   system usage: CPU: user: 8.90 s, system: 1.04 s, elapsed: 29.53 s
   VACUUM
   Time: 29543.681 ms (00:29.544)
   DROP TABLE
   ALTER SYSTEM
   ALTER SYSTEM
    pg_reload_conf
   ----------------
    t
   (1 row)
```

Выполнено 2 прохода по индексу.

3) Выполните тот же тест на 16 версии:

```
   postgres@tantor:~$ psql
   postgres=# \timing on \\
   alter system set maintenance_work_mem='2MB';
   select pg_reload_conf();
   drop table if exists test;
   create table test with (autovacuum_enabled=off) as select * from
generate_series(1, 10000000) x(id);
   create index on test(id);
   update test set id = id - 1;
   checkpoint;
   vacuum verbose test;
   drop table test;
   alter system reset max_wal_size;
   select pg_reload_conf();
   checkpoint;
   ALTER SYSTEM
   ALTER SYSTEM
   Time: 28.164 ms
    pg_reload_conf
   ----------------
    t
   (1 row)

   Time: 9.007 ms
   SELECT 10000000
   Time: 15005.658 ms (00:15.006)
   CREATE INDEX
   Time: 14352.266 ms (00:14.352)
   UPDATE 10000000
   Time: 73097.796 ms (01:13.098)
   CHECKPOINT
   Time: 3982.094 ms (00:03.982)
   INFO:  vacuuming "postgres.public.test"
   INFO:  finished vacuuming "postgres.public.test": index scans: 29
   pages: 0 removed, 88496 remain, 88496 scanned (100.00% of total)
   tuples: 10000000 removed, 10000000 remain, 0 are dead but not yet removable, oldest xmin:
22095822
   removable cutoff: 22095822, which was 0 XIDs old when operation ended
   new relfrozenxid: 22095822, which is 3 XIDs ahead of previous value
   frozen: 44249 pages from table (50.00% of total) had 10000000 tuples frozen
   index scan needed: 44248 pages from table (50.00% of total) had 10000000 dead item identifiers
removed
   index "test_id_idx": pages: 54840 in total, 0 newly deleted, 0 currently deleted, 0 reusable
   I/O timings: read: 10293.486 ms, write: 1492.231 ms
   avg read rate: 142.216 MB/s, avg write rate: 17.738 MB/s
   buffer usage: 326195 hits, 1485553 misses, 185284 dirtied
   WAL usage: 364426 records, 143159 full page images, 1008723444 bytes
   system usage: CPU: user: 19.79 s, system: 20.94 s, elapsed: 81.60 s
   VACUUM
```

```
Time: 81608.506 ms (01:21.609)
DROP TABLE
ALTER SYSTEM
```

Число проходов по индексу **index scans: 29** , а на 17 версии **index scans: 2**.

4) Сколько памяти нужно, чтобы использовалось 2 прохода на версиях до 17? Подсоединитесь на порт 5435 версии и выполните тест с **maintenance_work_mem='29MB'**:

```
postgres@tantor:~$ psql -p 5435
psql (16.2, server 15.6 (Debian 15.6-astra.se2))
Type "help" for help.
postgres=# \timing on \\
alter system set maintenance_work_mem='29MB';
select pg_reload_conf();
drop table if exists test;
create table test with (autovacuum_enabled=off) as select * from
generate_series(1, 10000000) x(id);
create index on test(id);
update test set id = id - 1;
checkpoint;
vacuum verbose test;
alter system reset max_wal_size;
select pg_reload_conf();
checkpoint;
ALTER SYSTEM
 pg_reload_conf
----------------
 t
(1 row)

SELECT 10000000
Time: 15607.619 ms (00:15.608)
CREATE INDEX
Time: 10548.036 ms (00:10.548)
UPDATE 10000000
Time: 73894.497 ms (01:13.894)
CHECKPOINT
Time: 5350.487 ms (00:05.350)
ИНФОРМАЦИЯ:  очистка "postgres.public.test"
ИНФОРМАЦИЯ:  закончена очистка "postgres.public.test": сканирований индекса: 2
страниц удалено: 0, осталось: 108109, просканировано: 108109 (100.00% от общего числа)
версий строк: удалено: 10000000, осталось: 10000000, «мёртвых», но ещё не подлежащих удалению: 0
XID отсечки удаления: 1098442, на момент завершения операции он имел возраст: 0 XID
новое значение relfrozenxid: 1098441, оно продвинулось вперёд от предыдущего значения на 2 XID
сканирование индекса требуется: на страницах таблицы (54055, 50.00% от общего числа) удалено
мёртвых идентификаторов элементов: 10000000
индекс "test_id_idx": всего страниц: 77011, сейчас удалено: 0, удалено на данный момент: 0,
свободно: 0
средняя скорость чтения: 34.614 МБ/с, средняя скорость записи: 28.333 МБ/с
использование буфера: попаданий: 134768, промахов: 289578, «грязных» записей: 237034
использование WAL: записей: 400977, полных образов страниц: 184760, байт: 1229393586
нагрузка системы: CPU: пользов.: 7.57 с, система: 1.94 с, прошло: 65.35 с
VACUUM
Time: 65358.960 ms (01:05.359)
DROP TABLE
Time: 214.219 ms
ALTER SYSTEM
Time: 29.511 ms
```

Если установить **maintenance_work_mem='28MB'**, то будет 3 прохода по индексу:

```
ИНФОРМАЦИЯ:  закончена очистка "postgres.public.test": сканирований индекса: 3
средняя скорость чтения: 41.495 МБ/с, средняя скорость записи: 27.448 МБ/с
использование буфера: попаданий: 143212, промахов: 358149, «грязных» записей: 236911
нагрузка системы: CPU: пользов.: 7.97 с, система: 2.27 с, прошло: 67.43 с
Time: 67433.238 ms (01:07.433)
```

При выполнении вакуумирования можно вывести в лог кластера распределение памяти серверного процесса или autovacuum worker. Основная часть памяти будет выделяться в транзакционном контексте **TopTransactionContext**. Пример:

```
LOG:  logging memory contexts of PID 464865
[464865] LOG:  level: 0; TopMemoryContext: 151040 total in 7 blocks; 14032 free (29 chunks); 137008 used
[464865] LOG:  level: 1; TopTransactionContext: 33562672 total in 2 blocks; 2904 free (0 chunks); 33559768 used
...
[464865] LOG:  Grand total: 35502216 bytes in 233 blocks; 727992 free (221 chunks); 34774224 used
```

При повторных выполнениях вывода в лог пока работает вакуум будет меняться только занятая память транзакционного контекста. Выделенная память пока транзакция работает не меняется.

```
level: 1; TopTransactionContext: 33587248 total in 4 blocks; 27400 free (325 chunks); 33559848 used
level: 1; TopTransactionContext: 33587248 total in 4 blocks; 31528 free (592 chunks); 33555720 used
```

Вакуум и автовакуум обрабатывают каждую таблицу (в том числе TOAST) в отдельной транзакции. Перед обработкой каждой таблицы создается моментальный снимок, который удерживает горизонт базы данных до завершения обработки таблицы и её TOAST таблицы.

Вывод в лог кластера памяти процесса рассматривался в части 4 практики к главе 9. Для этого используется функция **pg_log_backend_memory_contexts(PID)**, где - номер процесса, локальную память которого нужно вывести в лог. Номер процесса, обслуживающего текущую сессию можно получить функцией **pg_backend_pid()**.

5) По умолчанию ваккумирование использует опцию INDEX_CLEANUP AUTO. Это означает, что если блоков, в которых есть хоть одна строка dead наберется меньше 2% от всех блоков в таблице и одновременно память под TID (6 байт) dead строк будет меньше, чем 32Мб (2^25/6=5592400 строк **до** 17 версии), то остальные фазы не выполняются и индексы не сканируются. Обновление 1 млн.строк превысит эти границы.

Пример команд:

```
postgres=# set maintenance_work_mem='3MB';
UPDATE test SET id = id - 1 where id>9000000;
vacuum verbose test;
SET
Time: 0.215 ms
UPDATE 999999
Time: 7499.271 ms (00:07.499)
ИНФОРМАЦИЯ:  очистка "postgres.public.test"
ИНФОРМАЦИЯ:  закончена очистка "postgres.public.test": сканирований индекса: 2
страниц удалено: 0, осталось: 108109, просканировано: 10813 (10.00% от общего числа)
версий строк: удалено: 999814, осталось: 9999812, «мёртвых», но ещё не подлежащих удалению: 0
из-за конфликта блокировки очистки пропущено версий строк: 185, на страницах: 1
XID отсечки удаления: 1098464, на момент завершения операции он имел возраст: 0 XID
сканирование индекса требуется: на страницах таблицы (5408, 5.00% от общего числа) удалено мёртвых идентификаторов элементов: 999814
индекс "test_id_idx": всего страниц: 77011, сейчас удалено: 0, удалено на данный момент: 0, свободно: 0
средняя скорость чтения: 525.782 МБ/с, средняя скорость записи: 18.311 МБ/с
использование буфера: попаданий: 37880, промахов: 143223, «грязных» записей: 4988
использование WAL: записей: 29297, полных образов страниц: 0, байт: 7572755
нагрузка системы: CPU: пользов.: 1.54 с, система: 0.00 с, прошло: 2.12 с
VACUUM
Time: 2128.807 ms (00:02.129)
postgres=# set maintenance_work_mem='2MB';
UPDATE test SET id = id - 1 where id>9000000;
vacuum verbose test;
SET
Time: 0.280 ms
UPDATE 999998
Time: 7492.749 ms (00:07.493)
ИНФОРМАЦИЯ:  очистка "postgres.public.test"
ИНФОРМАЦИЯ:  закончена очистка "postgres.public.test": сканирований индекса: 3
```

```
   страниц удалено: 0, осталось: 108109, просканировано: 10813 (10.00% от общего числа)
   версий строк: удалено: 999998, осталось: 9999469, «мёртвых», но ещё не подлежащих удалению: 0
   XID отсечки удаления: 1098465, на момент завершения операции он имел возраст: 0 XID
   сканирование индекса требуется: на страницах таблицы (5409, 5.00% от общего числа) удалено
мёртвых идентификаторов элементов: 1000183
   индекс "test_id_idx": всего страниц: 77011, сейчас удалено: 0, удалено на данный момент: 0,
свободно: 0
   средняя скорость чтения: 642.501 МБ/с, средняя скорость записи: 16.255 МБ/с
   использование буфера: попаданий: 45705, промахов: 212417, «грязных» записей: 5374
   использование WAL: записей: 29303, полных образов страниц: 0, байт: 7576494
   нагрузка системы: CPU: пользов.: 1.99 с, система: 0.03 с, прошло: 2.58 с
   VACUUM
   Time: 2583.433 ms (00:02.583)
   postgres=# set maintenance_work_mem='1MB';
   UPDATE test SET id = id - 1 where id>9000000;
   vacuum verbose test;
   SET
   Time: 0.213 ms
   UPDATE 999997
   Time: 7490.491 ms (00:07.490)
   ИНФОРМАЦИЯ:  очистка "postgres.public.test"
   ИНФОРМАЦИЯ:  закончена очистка "postgres.public.test": сканирований индекса: 6
   страниц удалено: 0, осталось: 108109, просканировано: 10813 (10.00% от общего числа)
   версий строк: удалено: 999997, осталось: 9999327, «мёртвых», но ещё не подлежащих удалению: 0
   XID отсечки удаления: 1098466, на момент завершения операции он имел возраст: 0 XID
   сканирование индекса требуется: на страницах таблицы (5409, 5.00% от общего числа) удалено
мёртвых идентификаторов элементов: 999997
   индекс "test_id_idx": всего страниц: 77011, сейчас удалено: 0, удалено на данный момент: 0,
свободно: 0
   средняя скорость чтения: 820.235 МБ/с, средняя скорость записи: 10.130 МБ/с
   использование буфера: попаданий: 68842, промахов: 420324, «грязных» записей: 5191
   использование WAL: записей: 29303, полных образов страниц: 0, байт: 7574017
   нагрузка системы: CPU: пользов.: 3.41 с, система: 0.02 с, прошло: 4.00 с
   VACUUM
   Time: 4003.828 ms (00:04.004)
```

Увеличение **сканирований** одного **индекса** с **2** до **6** увеличило время вакуумирования в 2 раза: с **2.12 с** до **4.00 с**, что существенно.

Важно при настройке автовакуума, чтобы число сканирований индексов было 1, то есть индексы сканировались за один проход. Для этого устанавливают значение параметра `autovacuum_work_mem` таким, чтобы структура с идентификаторами помеченных к удалению строк таблицы, которую заполняет автовакуум, помещалась в выделенную этим параметром память. Так как объем помечаемых к удалению старых версий строк (в примере **`UPDATE test SET id = id - 1 where id>9000000`** порождает 1 млн. версий строк) обычно зависит от времени (скорость их появления обычно одинакова в течение рабочего времени), то также следят за тем, чтобы цикл автовакуума не работал существенно дольше чем обычно. Если цикл автовакуума увеличится, то за время цикла накопится больше изменившихся строк. Также таблица может быть заблокирована и автовакумм пропустит обработку в цикле. Поэтому, команды, уровень блокировки которых мешает работать автовакуму (пример рассматривался в 4 части этой практики, ошибка: `LOG:  skipping vacuum of "test" --- lock not available`) стараются давать нечасто. Пример такой команды: `ANALYZE test`.

6) **Как рассчитывать число проходов по индексам в зависимости от числа строк, подлежащих чистке до 17 версии?**

Как расчитывать `maintenance_work_mem` или `autovacuum_work_mem`, чтобы был один проход по индексам, если известно число строк, которые будут вычищены до 17 версии?

Число строк, которые могут быть удалены заполняются при работе процессов в столбце `pg_stat_all_tables.n_dead_tup`. Это верхняя оценка для числа удаляемых строк, то есть максимум сколько может быть удалено.

Память под хранение идентификаторов строк рассчитывается по формуле: `maintenance_work_mem = n_dead_tup` * 6. **Но не больше 1 гигабайта.**

Проверьте соответствует ли формула данным предыдущего пункта. Число удалённых строк показано в выводе команды vacuum. Пример:

"**удалено мёртвых идентификаторов элементов: 999997**" * 6 байт = 5999982 байт для одно прохода. В последней команде предыдущего пункта "**`set maintenance_work_mem='1MB'`**",

значит потребуется 5999982 байт / 1 мегабайт = 5999982 / 1048576 = 5.722 округляем в большую сторону и получаем "**сканирований индекса: 6**".

При превышении какого числа удаляемых строк на 16 версии будет 2 прохода по индексам независимо от выделенной памяти? 179 миллиона строк (1Гб/6=178956970 строк).

В 17 версии память кореллирует с **n_dead_tup**, но точной зависимости нет. Требуемый объем памяти меньше в ~20 раз.

7) Удалите таблицу test в тех базах, где вы её создавали, так как таблица занимает много места и уменьшите **max_wal_size до значения по умолчанию**:

```
postgres=# drop table test;
alter system reset max_wal_size;
checkpoint;
select pg_reload_conf();
DROP TABLE
```

8) Остановите контейнер докер и удалите директорию кластера в докере:

```
postgres@tantor:~$ sudo docker rm -f postgres
sudo rm -rf /root/data
postgres
```

Часть 7. Логирование автовакуума

Параметр `log_autovacuum_min_duration` устанавливает длительность автовакумма, по превышении которой будут записываться сообщения об автовакууме в диагностический журнал.

1) Выполните команды:

```
postgres@tantor:~$ rm -f $PGDATA/log/*
psql -c "alter system set log_autovacuum_min_duration = 0;"
psql -c "alter system set logging_collector = off;"
psql -c "alter system set autovacuum_naptime = '1s';"
psql -c "alter system set max_wal_size = '2GB';"
psql -c "alter system set checkpoint_timeout='30min';"
psql -c "alter system set log_min_duration_statement = '15s';"
psql -c "drop table if exists t;"
psql -c "create table t (a text storage external);"
pg_ctl stop
pg_ctl start

ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
DROP TABLE
CREATE TABLE
...
 done
server stopped
...
 done
server started
```

Создаётся таблица, в которую будут вставляться строки. Чтобы сообщения о контрольных точках не выводились, устанавливается интервал для контрольной точки по времени 30 минут и объем 2Гб для контрольной точки по размеру WAL.

2) Создайте скрипт для теста:

```
postgres@tantor:~$ mcedit t.sql
```

```
select format(E'insert into t values (repeat(\'a\',2010))') from
generate_series(1, 1000) as g(id)
\gexec
```

Размер значения столбца задан так, что с учётом **storage external** значения будут вытесняться в TOAST.

3) Запустите тест и наблюдайте за выводом диагностический сообщений в терминал:

```
postgres@tantor:~$ time psql -f t.sql > /dev/null
```

```
22:16:51.736 [238943] LOG:  automatic analyze of table "postgres.public.t"
        I/O timings: read: 0.511 ms, write: 0.000 ms
        avg read rate: 125.000 MB/s, avg write rate: 7.812 MB/s
        buffer usage: 103 hits, 32 misses, 2 dirtied
        system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
22:16:52.741 [238944] LOG:  automatic analyze of table "postgres.public.t"
        I/O timings: read: 0.000 ms, write: 0.000 ms
        avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
        buffer usage: 133 hits, 0 misses, 0 dirtied
        system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
22:16:52.754 [238944] LOG:automatic vacuum of table "postgres.pg_toast.pg_toast_11129217": index scans: 0
        pages: 0 removed, 222 remain, 222 scanned (100.00% of total)
        tuples: 0 removed, 1332 remain, 0 are dead but not yet removable, oldest xmin: 22106885
        removable cutoff: 22106885, which was 2 XIDs old when operation ended
        new relfrozenxid: 22106218, which is 1 XIDs ahead of previous value
        frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
        index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
        I/O timings: read: 0.000 ms, write: 0.040 ms
        avg read rate: 0.000 MB/s, avg write rate: 3.500 MB/s
        buffer usage: 476 hits, 0 misses, 3 dirtied
        WAL usage: 224 records, 3 full page images, 39250 bytes
        system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
22:16:53.738 [238945] LOG:  automatic analyze of table "postgres.public.t"
        I/O timings: read: 0.000 ms, write: 0.000 ms
        avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
        buffer usage: 134 hits, 0 misses, 0 dirtied
        system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
22:16:54.739 [238946] LOG:  automatic analyze of table "postgres.public.t"
        I/O timings: read: 0.000 ms, write: 0.000 ms
        avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
        buffer usage: 136 hits, 0 misses, 0 dirtied
        system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s

real    0m4.807s
user    0m0.031s
sys     0m0.030s

22:16:55.745 [238947] LOG:  automatic analyze of table "postgres.public.t"
        I/O timings: read: 0.000 ms, write: 0.000 ms
        avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
        buffer usage: 136 hits, 0 misses, 0 dirtied
        system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
22:16:55.757 [238947] LOG:automatic vacuum of table "postgres.pg_toast.pg_toast_11129217": index scans: 0
        pages: 0 removed, 400 remain, 179 scanned (44.75% of total)
        tuples: 0 removed, 2400 remain, 0 are dead but not yet removable, oldest xmin: 22107423
        removable cutoff: 22107423, which was 0 XIDs old when operation ended
        frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
        index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
        I/O timings: read: 0.000 ms, write: 0.000 ms
        avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
        buffer usage: 391 hits, 0 misses, 0 dirtied
        WAL usage: 180 records, 0 full page images, 11837 bytes
        system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
```

Были запущены автоанализ (**TOAST не могут анализироваться**) и автовакуум (таблиц t и её TOAST). Автовакуум ничего не очистил, так как старых версий строк не было, но и не заморозили ни одной строки.

Частота срабатывания автовакумма, автоанализа для вставок и изменений определяется параметрами:

```
postgres=# \dconfig *scale_factor*
       List of configuration parameters
               Parameter                | Value
----------------------------------------+-------
 autovacuum_analyze_scale_factor        | 0.1
 autovacuum_vacuum_insert_scale_factor | 0.2
 autovacuum_vacuum_scale_factor         | 0.2
(3 rows)
```

Для автоанализа достаточно изменить 10% строк. Если в цикле автовакуума должен запускаться и вакуум и анализ, то сначала запускается вакуум по каждой таблице+TOAST и только потом анализ.

4) Попробуйте проанализировать TOAST-таблицу:

```
postgres=# analyze pg_toast.pg_toast_11129217;
WARNING:  skipping "pg_toast.pg_toast_11129217" --- cannot analyze non-tables
or special system tables
ANALYZE
```

TOAST-таблицы не анализируются, по ним статистика не нужна.

5) Выполните обновление всех строк:

```
postgres@tantor:~$ psql -c "update t set a=(repeat('b',2010));"
UPDATE 1200

22:28:52.815 [239757] LOG:  automatic vacuum of table "postgres.public.t": index scans: 0
        pages: 0 removed, 16 remain, 16 scanned (100.00% of total)
        tuples: 1200 removed, 1200 remain, 0 are dead but not yet removable, oldest xmin: 22107424
        removable cutoff: 22107424, which was 0 XIDs old when operation ended
        new relfrozenxid: 22107423, which is 1206 XIDs ahead of previous value
        frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
        index scan not needed: 8 pages from table (50.00% of total) had 1200 dead item identifiers removed
        I/O timings: read: 0.000 ms, write: 0.024 ms
        avg read rate: 0.000 MB/s, avg write rate: 29.444 MB/s
        buffer usage: 54 hits, 0 misses, 3 dirtied
        WAL usage: 35 records, 3 full page images, 31621 bytes
        system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
22:28:52.819 [239757] LOG:  automatic analyze of table "postgres.public.t"
        I/O timings: read: 0.000 ms, write: 0.000 ms
        avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
        buffer usage: 129 hits, 0 misses, 0 dirtied
        system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
22:28:52.852 [239757] LOG: automatic vacuum of table "postgres.pg_toast.pg_toast_11129217": index scans: 1
        pages: 0 removed, 800 remain, 800 scanned (100.00% of total)
        tuples: 2400 removed, 2400 remain, 0 are dead but not yet removable, oldest xmin: 22107425
        removable cutoff: 22107425, which was 0 XIDs old when operation ended
        new relfrozenxid: 22107423, which is 1205 XIDs ahead of previous value
        frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
        index scan needed: 400 pages from table (50.00% of total) had 2400 dead item identifiers removed
        index "pg_toast_11129217_index": pages: 16 in total, 6 newly deleted, 6 currently deleted, 0 reusable
        I/O timings: read: 0.000 ms, write: 0.000 ms
        avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
        buffer usage: 2084 hits, 0 misses, 0 dirtied
        WAL usage: 1622 records, 0 full page images, 113050 bytes
        system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.02 s
```

Был выполнен один цикл автовакуума и автоанализа. По таблицам TOAST автоанализ не работает, так как статистика по TOAST для планировщика не нужна. Для доступа к строкам TOAST всегда используется доступ с помощью TOAST-индекса.

6) Выполните команды:

```
postgres@tantor:~$
psql -c "alter system set logging_collector = on;"
psql -c "alter system reset log_autovacuum_min_duration;"
psql -c "alter system reset max_wal_size;"
```

```
psql -p 5435 -c "alter system reset max_wal_size;"
psql -p 5435 -c "checkpoint;"
psql -p 5435 -c "alter system reset max_wal_size;"
psql -p 5435 -c "checkpoint;"
psql -h 172.18.0.2 -p 5432 -U postgres "alter system reset max_wal_size;"
psql -h 172.18.0.2 -p 5432 -U postgres "checkpoint;"
psql -c "drop table if exists t;"
pg_ctl stop
sudo systemctl restart tantor-se-server-16
```

Часть 8. Расширение pgststtuple

1) Установите расширение pgstattuple:

```
postgres=# create extension if not exists pgstattuple;
NOTICE:  extension "pgstattuple" already exists, skipping
CREATE EXTENSION
```

2) С помощью расширения можно получить плотность заполнения листовых блоков индекса:

```
postgres=# select relname name, index_size size, leaf_pages leaf, empty_pages
empty, deleted_pages del, avg_leaf_density density from pg_class,
pgstatindex(oid) b WHERE relkind='i' and  avg_leaf_density<>'NaN' and
leaf_pages>10 order by 6 limit 5;
               name                |  size   | leaf | empty | del | density
-----------------------------------+---------+------+-------+-----+---------
 pg_type_typname_nsp_index         | 2580480 |   14 |     0 | 297 |   27.24
 pg_class_relname_nsp_index        | 3547136 |   11 |     0 | 418 |   35.39
 pg_depend_reference_index         | 4055040 |   14 |     0 | 477 |   45.09
 pg_depend_depender_index          | 4718592 |   15 |     0 | 557 |   61.95
 pg_attribute_relid_attnum_index | 6742016 |   18 |     0 | 801 |   81.56
(5 rows)
```

3) С помощью расширения можно получить примерный процент строк, которые можно очистить:

```
postgres=# select relname name, approx_tuple_count count, dead_tuple_count
dead, trunc(dead_tuple_percent) "dead%", approx_tuple_len len,
trunc(approx_free_percent) "free%", trunc(scanned_percent) "sample%" from
pg_class, pgstattuple_approx(oid) b WHERE relkind='r' order by 4 desc limit 5;
      name      | count | dead | dead% |  len   | free% | sample%
----------------+-------+------+-------+--------+-------+---------
 pg_statistic   |   415 |  104 |    15 | 174085 |    14 |      70
 pg_namespace   |     9 |   10 |    11 |   1076 |    73 |     100
 pg_index       |   279 |   23 |     6 |  50562 |     4 |      14
 pg_depend      |  2985 |  121 |     3 | 186380 |     0 |       8
 pg_type        |   898 |   28 |     2 | 170024 |     2 |       9
(5 rows)
```

4) Для оценки раздувания таблиц и индексов можно использовать запросы к существующей статистике. Статистика должна быть собрана. Пример:

```
postgres@tantor:~$ wget
https://raw.githubusercontent.com/NikolayS/postgres_dba/refs/heads/master/sql/b1_
table_estimation.sql

Saving to: ‘b1_table_estimation.sql’
```

```
postgres@tantor:~$ wget
https://raw.githubusercontent.com/NikolayS/postgres_dba/refs/heads/master/sql/b2_
btree_estimation.sql

Saving to: 'b2_btree_estimation.sql'

postgres@tantor:~$ psql -f b1_table_estimation.sql
     Table       |    Size    |      Extra       |  Bloat estimate  |    Live     |Last Vaccuum
-----------------+------------+------------------+------------------+-------------+-------------
pgbench_tellers  | 8192 bytes | ~0 bytes (0.00%) | ~0 bytes (0.00%) | ~8192 bytes |13:29:02
shmem_reference  | 8192 bytes | ~0 bytes (0.00%) | ~0 bytes (0.00%) | ~8192 bytes |12:50:27
pgbench_branches | 8192 bytes | ~0 bytes (0.00%) | ~0 bytes (0.00%) | ~8192 bytes |13:29:02
t2               | 8192 bytes | ~0 bytes (0.00%) | ~0 bytes (0.00%) | ~8192 bytes |13:29:32 (auto)
pgbench_history  | 0 bytes    | ~0 bytes (0.00%) | ~0 bytes (0.00%) | ~0 bytes    |12:50:31
test2            | 592 kB     | ~0 bytes (0.00%) | ~0 bytes (0.00%) | ~592 kB     |12:50:31
t1               | 8192 bytes | ~0 bytes (0.00%) | ~0 bytes (0.00%) | ~8192 bytes |13:29:02 (auto)
t                | 3264 kB    |                  |                  | ~4864 kB    |
pgbench_accounts | 13 MB      | ~184 kB (1.40%)  |                  | ~126 MB     |15:25:50
(9 rows)
```

Часть 9. Условие обработки индексов: 2% строк

1) Проверьте сколько свободного места в файловой системе директории PGDATA:

```
postgres@tantor:~$ df -B M | grep sda1
Filesystem     1M-blocks   Used Available Use% Mounted on
udev               1921M     0M      1921M   0% /dev
tmpfs               393M     3M       391M   1% /run
/dev/sda1         47190M 31412M     13349M  71% /
tmpfs              1964M     4M      1960M   1% /dev/shm
tmpfs                 5M     0M         5M   0% /run/lock
tmpfs               393M     1M       393M   1% /run/user/102
tmpfs               393M     1M       393M   1% /run/user/1000
```

Если меньше 3400MB, то следующие пункты не надо выполнять.
Место может быть занято директориями WAL и большими таблицами.

2) Включите измерение времени. Время полезно для оценки времени выполнения команд.

```
postgres=# \timing on
Timing is on.
```

Если блоков, в которых есть хоть одна dead строка наберется меньше 2% от всех блоков в таблице и одновременно память под TID (6 байт) мертвых строк будет меньше, чем 32Мб (2^25/6=5592400 строк до 17 версии), то индексы не сканируются.
То есть большая таблица это 5592400/0.02 = 279620000 строк.
Чтобы в таблице обновилось меньше 2% строк и при этом вакуум сканировал индексы (по причине того, что TID мертвых строк превысит 32Мб) нужно, чтобы в таблице было больше 279620000 строк. Сколько места займёт такая таблица? Максимальное число строк в таблице 226, таблица с одним столбцом типа int4 или int8 займёт 10135609344 байт=9,44Гб.
Мы это тестировать не будем.

3) Создайте таблицу с 10млн. строк.

```
postgres=# drop table if exists test;
create table test with (autovacuum_enabled = off) as select * from
generate_series(1, 10000000) x(id);
CREATE INDEX ON test(id);

DROP TABLE
Time: 362.244 ms
SELECT 10000000
```

```
Time: 14903.225 ms (00:14.903)
CREATE INDEX
Time: 13010.045 ms (00:13.010)
```

4) Обновите 2% строк:

```
postgres=# UPDATE test SET id = id - 1 where id<=200000;

UPDATE 200000
Time: 1352.337 ms (00:01.352)
```

5) Посмотрите сколько места занимают таблица и индекс:

```
postgres=# select pg_total_relation_size('test');
select pg_size_pretty(pg_total_relation_size('test'));

 pg_total_relation_size
------------------------
              599302144
(1 row)

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

Они занимают `572Мб`, что равно `599302144` байт.

6) Посмотрите долю и процент мертвых строк:

```
postgres=# select relname, n_dead_tup, n_live_tup,
round(100.0*n_dead_tup/n_live_tup, 2) pct1,
round(100.0*n_dead_tup/(n_live_tup+n_dead_tup),2) pct2  from pg_stat_user_tables
where relname = 'test';

 relname | n_dead_tup | n_live_tup | pct1 | pct2
---------+------------+------------+------+------
 test    |     200000 |   10000000 | 2.00 | 1.96
(1 row)
```

Какой из "процентов" использует вакуум и автовакуум чтобы решить стоит ли обрабатывать индексы? Ответ будет дальше.

7) Посмотрите как считают процент функции pgstattuple и насколько долго они считают:

```
postgres=# select * from pgstattuple_approx('test')\gx
-[ RECORD 1 ]--------+--------------------
table_len            | 370049024
scanned_percent      | 100
approx_tuple_count   | 10000000
approx_tuple_len     | 280000000
approx_tuple_percent | 75.66565018152838
dead_tuple_count     | 200000
dead_tuple_len       | 5600000
dead_tuple_percent   | 1.5133130036305675
approx_free_space    | 862236
approx_free_percent  | 0.23300588410685824

Time: 778.598 ms
```

```
postgres=# select * from pgstattuple('test')\gx
-[ RECORD 1 ]------+----------
table_len          | 370049024
tuple_count        | 10000000
tuple_len          | 280000000
tuple_percent      | 75.67
dead_tuple_count   | 200000
dead_tuple_len     | 5600000
dead_tuple_percent | 1.51
free_space         | 543684
free_percent       | 0.15

Time: 1857.216 ms (00:01.857)
```

Разница во времени выполнения функций невелика.
dead_tuple_percent = 1.51% функции считают от размера блока (файлов).

8) Выполните вакуумирование:

```
postgres=# vacuum (verbose) test;

INFO:  vacuuming "postgres.public.test"
INFO:  finished vacuuming "postgres.public.test": index scans: 0
pages: 0 removed, 45172 remain, 45172 scanned (100.00% of total)
tuples: 200000 removed, 10000000 remain, 0 are dead but not yet removable, oldest xmin: 22113906
removable cutoff: 22113906, which was 0 XIDs old when operation ended
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan bypassed: 885 pages from table (1.96% of total) have 200000 dead item identifiers
I/O timings: read: 190.718 ms, write: 246.465 ms
avg read rate: 26.787 MB/s, avg write rate: 34.300 MB/s
buffer usage: 59491 hits, 30026 misses, 38448 dirtied
WAL usage: 45137 records, 44255 full page images, 365378454 bytes
system usage: CPU: user: 1.43 s, system: 1.08 s, elapsed: 8.75 s
VACUUM
Time: 8757.911 ms (00:08.758)
```

Ваккум считает процент мертвых строк по формуле: `n_dead_tup/(n_live_tup+n_dead_tup)`

9) 5592400 мертвых строк займут ровно 32Мб. Проверим, что помимо массива TID `maintenance_work_mem` ограничивает память и под другие структуры.
Установите maintenance_work_mem = '32MB' и обновите 5592200 строк, чтобы массив TID занял чуть меньше 32Мб и запустите вакуум:

```
postgres=# set maintenance_work_mem = '32MB';
UPDATE test SET id = id - 1 where id<=5592200;
vacuum (verbose) test;

SET
UPDATE 5592200
Time: 41250.795 ms (00:41.251)
INFO:  vacuuming "postgres.public.test"
INFO:  finished vacuuming "postgres.public.test": index scans: 2
pages: 0 removed, 69877 remain, 50376 scanned (72.09% of total)
tuples: 5592200 removed, 9909829 remain, 0 are dead but not yet removable, oldest xmin: 22113907
removable cutoff: 22113907, which was 0 XIDs old when operation ended
frozen: 24748 pages from table (35.42% of total) had 5592548 tuples frozen
index scan needed: 25630 pages from table (36.68% of total) had 5792200 dead item identifiers
removed
index "test_id_idx": pages: 43304 in total, 0 newly deleted, 0 currently deleted, 0 reusable
I/O timings: read: 933.086 ms, write: 762.870 ms
avg read rate: 32.593 MB/s, avg write rate: 23.729 MB/s
buffer usage: 76928 hits, 136100 misses, 99088 dirtied
WAL usage: 204558 records, 102164 full page images, 525140763 bytes
system usage: CPU: user: 6.67 s, system: 3.78 s, elapsed: 32.62 s
```

```
VACUUM
Time: 32625.933 ms (00:32.626)
```

Было выполнено 2 прохода по индексам, памяти **maintenance_work_mem** на один проход не хватило.

10) При установке с небольшим (1Мб) запасом памяти хватит и будет выполнен один проход:

```
postgres=# set maintenance_work_mem = '33MB';
UPDATE test SET id = id - 1 where id<=5592500;
vacuum (verbose) test;

SET
Time: 0.166 ms
UPDATE 5592500
Time: 43902.661 ms (00:43.903)
INFO:  vacuuming "postgres.public.test"
INFO:  finished vacuuming "postgres.public.test": index scans: 1
pages: 0 removed, 69877 remain, 49494 scanned (70.83% of total)
tuples: 5592500 removed, 8483454 remain, 0 are dead but not yet removable, oldest xmin: 22113908
removable cutoff: 22113908, which was 0 XIDs old when operation ended
frozen: 21270 pages from table (30.44% of total) had 4806279 tuples frozen
index scan needed: 24749 pages from table (35.42% of total) had 5592500 dead item identifiers
removed
index "test_id_idx": pages: 43304 in total, 0 newly deleted, 0 currently deleted, 0 reusable
I/O timings: read: 642.541 ms, write: 746.983 ms
avg read rate: 25.280 MB/s, avg write rate: 23.980 MB/s
buffer usage: 67906 hits, 99172 misses, 94072 dirtied
WAL usage: 196037 records, 91225 full page images, 517619377 bytes
system usage: CPU: user: 5.36 s, system: 3.31 s, elapsed: 30.64 s
VACUUM
Time: 30649.388 ms (00:30.649)
```

10) Удалите таблицу, чтобы не занимала место на диске:

```
postgres=# drop table test;
DROP TABLE
```

Практика к главе 13

Часть 1. Чтение сообщений вакуума и автовакуума

1) Посмотрите значение и описание параметра `log_autovacuum_min_duration`:

```
postgres=# select name, setting, unit, context, min_val, short_desc, extra_desc
from pg_settings where name ~ 'um_mi'\gx
-[ RECORD 1 ]----------------------------------------------------
name       | log_autovacuum_min_duration
setting    | 0
unit       | ms
context    | sighup
min_val    | -1
short_desc | Sets the minimum execution time above which autovacuum actions will be logged.
extra_desc | Zero prints all actions. -1 turns autovacuum logging off.
```

Значение по умолчанию 10 минут, это разумное значение. В практике ранее было установлено значение 0, что означает логирование всех действий автовакуума.

Сообщение записывается в лог, если по таблице или TOAST-таблице значение "elapsed:" из лога автовакуума превышает `log_autovacuum_min_duration`.

При возникновении таких сообщений стоит выяснять причину долгого вакуумирования таблицы.

Сообщение записывается в лог после завершения обработки таблицы и ее индексов, до этого наблюдать за прогрессом вакуумирования можно через представление `pg_stat_progress_vacuum`, а анализа через `pg_stat_progress_analyze`.

2) Выполните команду или посмотрите любые пример лога автовакуума или команд vacuum:

```
postgres=# vacuum verbose pg_class;
INFO:  vacuuming "postgres.pg_catalog.pg_class"
INFO:  launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO:  finished vacuuming "postgres.pg_catalog.pg_class": index scans: 1
pages: 0 removed, 29 remain, 29 scanned (100.00% of total)
tuples: 6 removed, 759 remain, 0 are dead but not yet removable, oldest xmin:
22113909
removable cutoff: 22113909, which was 0 XIDs old when operation ended
new relfrozenxid: 21894381, which is 3 XIDs ahead of previous value
frozen: 4 pages from table (13.79% of total) had 46 tuples frozen
index scan needed: 21 pages from table (72.41% of total) had 143 dead item
identifiers removed
index "pg_class_oid_index": pages: 151 in total, 0 newly deleted, 145 currently
deleted, 145 reusable
index "pg_class_relname_nsp_index": pages: 433 in total, 0 newly deleted, 418
currently deleted, 418 reusable
index "pg_class_tblspc_relfilenode_index": pages: 151 in total, 0 newly
deleted, 144 currently deleted, 144 reusable
I/O timings: read: 37.648 ms, write: 0.088 ms
avg read rate: 94.197 MB/s, avg write rate: 6.524 MB/s
buffer usage: 707 hits, 592 misses, 41 dirtied
WAL usage: 85 records, 44 full page images, 273017 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.04 s
VACUUM
```

Результат команды аналогичен тому, который выдаётся автовакуумом в лог.

посмотрим на что обращать внимание в первую очередь.

`index scans: 1` число проходов по индексам. 1 - идеальный случай, один проход. 0 - индексов нет или изменений было мало и они не сканировались. Причина несканирования в сообщении "`72.41% of total`" строки "`index scan needed`". Как оценивать и настраивать число сканирований описано в части 9 предыдущей практики.

`launched 2 parallel vacuum workers` означает, что индексы обрабатывались двумя параллельными процессами и тем, который вакуумирует. Степень параллелизма обычно рассчитывается корректно.

`elapsed: 0.04 s` время вакуумирования таблицы `postgres.pg_catalog.pg_class`. `user: 0.00 s` и `system: 0.00 s` не должны сильно отклоняться от пропорции 60/40. Сумма `user+system` может быть меньше, чем `elapsed`. Это означает, что вакуум сталкивался с получением блокировок при доступе к буферам разделяемого пула.

`tuples: 6 removed, 759 remain, 0 are dead but not yet removable` - эффективность этого цикла вакуума. Если число `6 removed` невелико, то вакуум работал впустую и нужно смотреть `0 are dead but not yet removable`. Если оно велико, то значит удерживался горизонт базы. Стоит оценить, что удерживает горизонт и если есть возможность уменьшить длительность удержания только потом настраивать автовакуум.

`frozen: 4 pages from table (13.79% of total)` указывает на то, что на части блоков после вакуумирования не осталось мертвых строк, все строки актуальные и удалось заморозить все строки в этом числе блоков. Этим значениям не нужно уделять внимание, они зависят от удержания горизонта моментальными снимками.

`WAL usage:` показывает сколько журнальных записей было создано, объем в конце строки `273017 bytes`. На объем наибольшее влияние оказывает запись полных образов страниц `44 full page images`.

По значениям `avg read rate: 94.197 MB/s, avg write rate: 6.524 MB/s` ничего сказать о работае ввода-вывода нельзя, так как на эти значения в большей степени оказывает влияние скорость ядра процессора, а не ввода-вывода.

`buffer usage`: вакуум, как и все процессы экземпляра, работает только через буферный кэш. Использование кэша буферов делает работу процессов более эффективной. Автовакуум использует буферное кольцо типа VACUUM, в котором грязные страницы не убираются из кольца, а посылаются на запись. Размер кольца задается параметром конфигурации `vacuum_buffer_usage_limit`. По умолчанию 256Кб, диапазон от 128Кб до 16Мб. В команде `VACUUM (BUFFER_USAGE_LIMIT 0)` можно установить значение 0, что означает запрет на использование буферного кольца и блоки всех обрабатываемых командой объектов как при очистке, так и при анализе могут занять все буфера. Это ускорит выполнение ваккумирования и если кэш буферов большой, загрузит обрабатываемые блоки в него. Для автовакуума значение 0 установить нельзя, но если автовакуум запускается для защиты от переполнения счетчика транзакций, то буферное кольцо не используется и автоочистка выполняется в агрессивном режиме.

Часть 2. Чтение сообщений о контрольной точке

1) Выполнять этот пункт не обязательно. Для создания записей о контрольной точке выполняются следующие действия:

В первом терминале удаляются лог-файлы, чтобы было проще искать последний файл. Устанавливается частота контрольных точек в 2 минуты, чтобы долго не ждать. Рестартует экземпляр. Запускается небольшая нагрузка:

```
postgres@tantor:~$ rm -f $PGDATA/log/postgresql*
psql -c "alter system set checkpoint_timeout=120;"
sudo systemctl restart tantor-se-server-16
while :; do pgbench -i; sleep 10; done
```

Во втором терминале запускаются команды:

```
postgres@tantor:~$ sleep 300
time psql -c "drop table if exists test; create table test with
(autovacuum_enabled = off) as select * from generate_series(1, 10000000) x(id);
drop table if exists test;"
sleep 260
psql -c "alter system reset checkpoint_timeout;"
```

```
psql -c "select pg_reload_conf();"
cat $PGDATA/log/postgresql*
```

```
real    0m14.942s
user    0m0.014s
sys     0m0.008s

12:00:04.950 MSK [331789] LOG:  starting Tantor Special Edition 16.2.0 e12e484f on x86_64-pc-
linux-gnu, compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
12:00:04.951 MSK [331789] LOG:  listening on IPv4 address "127.0.0.1", port 5432
12:00:04.951 MSK [331789] LOG:  listening on IPv6 address "::1", port 5432
12:00:04.959 MSK [331789] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
12:00:04.980 MSK [331793] LOG:  database system was shut down at 12:00:04 MSK
12:00:04.993 MSK [331789] LOG:  database system is ready to accept connections
12:02:05.079 MSK [331791] LOG:  checkpoint starting: time
12:03:53.791 MSK [331791] LOG:  checkpoint complete: wrote 174 buffers (1.1%); 0 WAL file(s)
added, 10 removed, 0 recycled; write=107.775 s, sync=0.630 s, total=108.712 s; sync files=293,
longest=0.009 s, average=0.003 s; distance=154031 kB, estimate=154031 kB; lsn=248/63E6F100, redo
lsn=248/5C112AF0
12:04:05.804 MSK [331791] LOG:  checkpoint starting: time
12:05:15.109 MSK [331791] LOG:  checkpoint complete: wrote 145 buffers (0.9%); 0 WAL file(s)
added, 8 removed, 0 recycled; write=67.819 s, sync=1.343 s, total=69.306 s; sync files=133,
longest=0.928 s, average=0.011 s; distance=141390 kB, estimate=152767 kB; lsn=248/864CD020, redo
lsn=248/64B26448
12:05:15.109 MSK [331791] LOG:  checkpoint starting: wal
12:07:03.773 MSK [331791] LOG:  checkpoint complete: wrote 117 buffers (0.7%); 0 WAL file(s)
added, 0 removed, 34 recycled; write=107.923 s, sync=0.365 s, total=108.664 s; sync files=152,
longest=0.014 s, average=0.003 s; distance=557039 kB, estimate=557039 kB; lsn=248/9854BCB8, redo
lsn=248/86B22090
12:07:15.783 MSK [331791] LOG:  checkpoint starting: time
12:09:03.659 MSK [331791] LOG:  checkpoint complete: wrote 2140 buffers (13.1%); 0 WAL file(s)
added, 0 removed, 19 recycled; write=107.211 s, sync=0.430 s, total=107.876 s; sync files=179,
longest=0.009 s, average=0.003 s; distance=301854 kB, estimate=531520 kB; lsn=248/A1BE70A8, redo
lsn=248/991E9908
12:09:15.667 MSK [331791] LOG:  checkpoint starting: time
12:11:03.524 MSK [331791] LOG:  checkpoint complete: wrote 156 buffers (1.0%); 0 WAL file(s)
added, 0 removed, 9 recycled; write=107.323 s, sync=0.394 s, total=107.857 s; sync files=168,
longest=0.016 s, average=0.003 s; distance=154209 kB, estimate=493789 kB; lsn=248/AA5D0EA0, redo
lsn=248/A2882090
12:11:15.537 MSK [331791] LOG:  checkpoint starting: time
12:13:03.507 MSK [331791] LOG:  checkpoint complete: wrote 125 buffers (0.8%); 0 WAL file(s)
added, 0 removed, 9 recycled; write=107.521 s, sync=0.312 s, total=107.971 s; sync files=153,
longest=0.009 s, average=0.003 s; distance=141196 kB, estimate=458530 kB; lsn=248/B23578D8, redo
lsn=248/AB265368
12:13:15.529 MSK [331791] LOG:  checkpoint starting: time
12:15:03.241 MSK [331791] LOG:  checkpoint complete: wrote 1792 buffers (10.9%); 0 WAL file(s)
added, 0 removed, 7 recycled; write=107.491 s, sync=0.107 s, total=107.712 s; sync files=45,
longest=0.011 s, average=0.003 s; distance=115797 kB, estimate=424256 kB; lsn=248/B237A898, redo
lsn=248/B237A808
12:15:38.289 MSK [331789] LOG:  received SIGHUP, reloading configuration files
12:15:38.290 MSK [331789] LOG:  parameter "checkpoint_timeout" removed from configuration file,
reset to default
12:20:15.392 MSK [331791] LOG:  checkpoint starting: time
12:23:25.949 MSK [331791] LOG:  checkpoint complete: wrote 163 buffers (1.0%); 0 WAL file(s)
added, 0 removed, 19 recycled; write=189.294 s, sync=1.032 s, total=190.557 s; sync files=501,
longest=0.009 s, average=0.003 s; distance=320376 kB, estimate=413868 kB; lsn=248/D24C4778, redo
lsn=248/C5C589C0
12:25:16.016 MSK [331791] LOG:  checkpoint starting: time
12:28:25.985 MSK [331791] LOG:  checkpoint complete: wrote 1813 buffers (11.1%); 0 WAL file(s)
added, 1 removed, 12 recycled; write=189.692 s, sync=0.113 s, total=189.970 s; sync files=51,
longest=0.010 s, average=0.003 s; distance=205385 kB, estimate=393020 kB; lsn=248/D24EB1C0, redo
lsn=248/D24EB178
12:41:21.274 MSK [331791] LOG:  checkpoint starting: immediate force wait
12:41:21.322 MSK [331791] LOG:  checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added,
0 removed, 0 recycled; write=0.001 s, sync=0.001 s, total=0.049 s; sync files=0, longest=0.000 s,
average=0.000 s; distance=0 kB, estimate=353718 kB; lsn=248/D24EB2E8, redo lsn=248/D24EB2A0
```

Через 520 секунд во втором терминале выдается содержимое лога и можно в первом терминале прервать <ctrl+c> цикл, создающий нагрузку.

Описание содержимого журнала:

```
12:00:04.950 MSK [331789] LOG:  starting Tantor Special Edition 16.2.0
12:00:04.951 MSK [331789] LOG:  listening on IPv4 address "127.0.0.1", port 5432
12:00:04.951 MSK [331789] LOG:  listening on IPv6 address "::1", port 5432
12:00:04.959 MSK [331789] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
12:00:04.980 MSK [331793] LOG:  database system was shut down at 12:00:04 MSK
12:00:04.993 MSK [331789] LOG:  database system is ready to accept connections
```

Экземпляр запустился в `12:00:04`. Контрольная точка началась через две минуты (через checkpoint_timeout) в `12:02:05`:

```
12:02:05.079 MSK [331791] LOG:  checkpoint starting: time
```

Контрольная точка завершилась через 90% интервала, в соответствии с checkpoint_completion_target=0.9. Расчет времени: 120*0.9=108 секунд. `12:02:05` + 108= `12:03:53`

```
12:03:53.791 MSK [331791] LOG:  checkpoint complete: wrote 174 buffers (1.1%); 0 WAL file(s)
added, 10 removed, 0 recycled; write=107.775 s, sync=0.630 s, total=108.712 s; sync files=293,
longest=0.009 s, average=0.003 s; distance=154031 kB, estimate=154031 kB; lsn=248/63E6F100, redo
lsn=248/5C112AF0
```

Ровно через две минуты после начала первой контрольной точки, следующая контрольная точка началась в в `12:04:05`:

```
12:04:05.804 MSK [331791] LOG:  checkpoint starting: time
```

Примерно через 300 секунд после запуска экземпляра запустилась команда, создающая больше 600Мб журнальных записей и проработавшая `14.942s`. Как только набралось больше половины от max_wal_size журнальных записей, контрольная точка по времени стала завершаться и завершилась за `sync=1.343 s` проработав `total=69.306 s`:

```
12:05:15.109 MSK [331791] LOG:  checkpoint complete: wrote 145 buffers (0.9%); 0 WAL file(s)
added, 8 removed, 0 recycled; write=67.819 s, sync=1.343 s, total=69.306 s; sync files=133,
longest=0.928 s, average=0.011 s; distance=141390 kB, estimate=152767 kB; lsn=248/864CD020, redo
lsn=248/64B26448
```

Тут же была создана запись о начале контрольной точки по размеру:

```
12:05:15.109 MSK [331791] LOG:  checkpoint starting: wal
```

Поскольку объем журнальных записей созданных командой отработавшей 14 секунд не превысил max_wal_size=1Гб, контрольная точка завершилась не с максимальной скоростью и проработала `108.664 s`. Если бы объем превысил 1Гб, то контрольная точка типа `wal` завершилась бы немного быстрее, началась бы следующая контрольная точка типа `wal` и в журнал бы были выданы предупреждающие сообщения сразу после окончании второй контрольной точки во размеру, выполнившейся с максимальной скоростью за 16 секунд:.

```
 LOG:  checkpoint complete: wrote 28 buffers (0.2%); 0 WAL file(s) added, 0 removed, 33 recycled;
write=14.989 s, sync=0.136 s, total=15.966 s; sync files=20, longest=0.083 s, average=0.007 s;
 LOG:  checkpoints are occurring too frequently (16 seconds apart)
 HINT:  Consider increasing the configuration parameter "max_wal_size".
```

Но в примере контрольная точка по размеру завершилась за `108.664 s` и записала в WAL `295869480` байт: `select '248/9854BCB8'::text::pg_lsn - '248/86B22090'::text::pg_lsn written;`

```
   written
-----------
 295869480
(1 row)
```

```
12:07:03.773 MSK [331791] LOG:  checkpoint complete: wrote 117 buffers (0.7%); 0 WAL file(s)
added, 0 removed, 34 recycled; write=107.923 s, sync=0.365 s, total=108.664 s; sync files=152,
longest=0.014 s, average=0.003 s; distance=557039 kB, estimate=557039 kB; lsn=248/9854BCB8, redo
lsn=248/86B22090
```

Контрольные точки по времени начались через 2 минуты (через checkpoint_timeout) после начала `12:05:15` контрольной точки по размеру:

```
 12:07:15.783 MSK [331791] LOG:  checkpoint starting: time
 12:09:03.659 MSK [331791] LOG:  checkpoint complete: wrote 2140 buffers (13.1%); 0 WAL file(s)
added, 0 removed, 19 recycled; write=107.211 s, sync=0.430 s, total=107.876 s; sync files=179,
longest=0.009 s, average=0.003 s; distance=301854 kB, estimate=531520 kB; lsn=248/A1BE70A8, redo
lsn=248/991E9908
```

Контрольные точки по времени начинаются через checkpoint_timeout после начала предыдущей контрольной точки:

```
 12:09:15.667 MSK [331791] LOG:  checkpoint starting: time
 12:11:03.524 MSK [331791] LOG:  checkpoint complete: wrote 156 buffers (1.0%); 0 WAL file(s)
added, 0 removed, 9 recycled; write=107.323 s, sync=0.394 s, total=107.857 s; sync files=168,
longest=0.016 s, average=0.003 s; distance=154209 kB, estimate=493789 kB; lsn=248/AA5D0EA0, redo
lsn=248/A2882090
 12:11:15.537 MSK [331791] LOG:  checkpoint starting: time
 12:13:03.507 MSK [331791] LOG:  checkpoint complete: wrote 125 buffers (0.8%); 0 WAL file(s)
added, 0 removed, 9 recycled; write=107.521 s, sync=0.312 s, total=107.971 s; sync files=153,
longest=0.009 s, average=0.003 s; distance=141196 kB, estimate=458530 kB; lsn=248/B23578D8, redo
lsn=248/AB265368
 12:13:15.529 MSK [331791] LOG:  checkpoint starting: time
 12:15:03.241 MSK [331791] LOG:  checkpoint complete: wrote 1792 buffers (10.9%); 0 WAL file(s)
added, 0 removed, 7 recycled; write=107.491 s, sync=0.107 s, total=107.712 s; sync files=45,
longest=0.011 s, average=0.003 s; distance=115797 kB, estimate=424256 kB; lsn=248/B237A898, redo
lsn=248/B237A808
 12:15:38.289 MSK [331789] LOG:  received SIGHUP, reloading configuration files
 12:15:38.290 MSK [331789] LOG:  parameter "checkpoint_timeout" removed from configuration file,
reset to default
```

2) Контрольная точка может быть вызвана командой checkpoint. В этом случае в логе будет сообщение:

```
 13:10:36.200 MSK [331791] LOG:  checkpoint starting: time
 13:12:38.670 MSK [331791] LOG:  checkpoint complete: wrote 137 buffers (0.8%); 0 WAL file(s)
added, 0 removed, 24 recycled; write=121.546 s, sync=0.645 s, total=122.470 s; sync files=301,
longest=0.009 s, average=0.003 s; distance=402997 kB, estimate=530375 kB; lsn=249/5740D278, redo
lsn=249/4DD94270
 13:12:38.670 MSK [331791] LOG:  checkpoint starting: immediate force wait
 13:12:39.004 MSK [331791] LOG:  checkpoint complete: wrote 1790 buffers (10.9%); 0 WAL file(s)
added, 0 removed, 10 recycled; write=0.077 s, sync=0.121 s, total=0.334 s; sync files=46,
longest=0.028 s, average=0.003 s; distance=154084 kB, estimate=492746 kB; lsn=249/5740D358, redo
lsn=249/5740D310
 13:17:38.106 MSK [331791] LOG:  checkpoint starting: time
 13:20:48.009 MSK [331791] LOG:  checkpoint complete: wrote 178 buffers (1.1%); 0 WAL
file(s) added, 1 removed, 21 recycled; write=188.370 s, sync=1.278 s,
total=189.904 s; sync files=560, longest=0.042 s, average=0.003 s;
distance=358909 kB, estimate=479362 kB; lsn=249/7C088840, redo lsn=249/6D28C9A0
```

Контрольная точка по времени начнется `13:17:38` через checkpoint_timeout после начала предыдущей контрольной точки `13:12:38`.

3) Пример последнего сообщения из предыдущего пункта, если бы они выдавались на русском языке:

```
 СООБЩЕНИЕ:  контрольная точка завершена: записано буферов: 178 (1.1%); добавлено
файлов WAL 0, удалено: 1, переработано: 21; запись=188.370 сек., синхр.=1.278
сек., всего=189.904 сек.; синхронизировано_файлов=560, самая_долгая_синхр.=0.042
сек., средняя=0.003 сек.; расстояние=358909 kB, ожидалось=479362 kB
```

Описание значений:

`wrote 178 buffers (1.1%); записано буферов` - число грязных буферов, которые записаны по контрольной точке. Одновременно с checkpointer грязные блоки могут записывать серверные процессы и bgwriter. В скобках процент от общего количества буферов буферного кэша, задаваемых параметром shared_buffers

`0 WAL file(s) added, добавлено файлов` - число созданных WAL-файлов

`1 removed, удалено` - число удалённых WAL-файлов
`21 recycled; переработано` - число повторно использованных WAL-файлов
`write=188.370 s, запись` - общая длительность команд на запись в страничный кэш linux
`sync=1.278 s, синхр.` - общая длительность вызовов fdatasync по WAL-файлам
`total=189.904 s; всего` - сумма `write+sync`
`sync files=560, синхронизировано_файлов` - число обработанных файлов данных (файлам в табличных пространствах и их блоков в буферном кэше)
`longest=0.042 s, самая_долгая_синхр.` - наибольшая длительность обработки fsync одного файла
`average=0.003 s; средняя` - средняя скорость выполнения fsync по всем синхронизированным файлам
`distance=358909 kB, расстояние` - объем записей WAL между **началом предыдущей** контрольной точки и **началом текущей к которой относится запись**
`estimate=479362 kB; ожидалось` - рассчитывается, чтобы оценить сколько WAL сегментов будет использовано в следующей контрольной точке. Формула расчёта:

```
Distance = nbytes;
if (Estimate < nbytes)
 Estimate = nbytes;
 else
  CheckPointDistanceEstimate =
   (0.90 * CheckPointDistanceEstimate + 0.10 * (double) nbytes);
```

Комментарий к формуле:

```
/*
 * To estimate the number of segments consumed between checkpoints, keep a
 * moving average of the amount of WAL generated in previous checkpoint
 * cycles. However, if the load is bursty, with quiet periods and busy
 * periods, we want to cater for the peak load. So instead of a plain
 * moving average, let the average decline slowly if the previous cycle
 * used less WAL than estimated, but bump it up immediately if it used
 * more.
 *
 * When checkpoints are triggered by max_wal_size, this should converge to
 * CheckpointSegments * wal_segment_size,
 *
 * Note: This doesn't pay any attention to what caused the checkpoint.
 * Checkpoints triggered manually with CHECKPOINT command, or by e.g.
 * starting a base backup, are counted the same as those created
 * automatically. The slow-decline will largely mask them out, if they are
 * not frequent. If they are frequent, it seems reasonable to count them
 * in as any others; if you issue a manual checkpoint every 5 minutes and
 * never let a timed checkpoint happen, it makes sense to base the
 * preallocation on that 5 minute interval rather than whatever
 * checkpoint_timeout is set to.
 */
```

Следствие формулы: если уменьшили `max_wal_size`, то число WAL-файлов уменьшается постепенно: через ~10 контрольных точек (следствие `0.90 * предыдущий estimate`).

`lsn=249/7C088840,` - LSN записи об окончании контрольной точки. Если это последняя контрольная точка, то это значение присутствует в управляющем файле.
`redo lsn=249/6D28C9A0` - LSN записи с которой началась контрольная точка. Эта запись формируется не checkpointer, а любым процессом.

Часть 3. Чтение сообщений о контрольной точке утилитой `pg_waldump`

1) В управляющем файле есть LSN записи об окончании и начале последней контрольной точки. Выполните команду:

```
postgres@tantor:~$ pg_controldata | grep check | head -n 3
```

```
Latest checkpoint location:           7/F7F28E08
Latest checkpoint's REDO location:    7/F1BF0018
Latest checkpoint's REDO WAL file:    000000010000000700000F1
```

2) Посмотрите, что содержится в журнальных записях. Выведите записи, а которых присутствует буквосочетание CHECKPOINT:

```
postgres@tantor:~$ pg_waldump -s 7/F1BF0018 -e 7/F7FFFFFF | grep CHECKPOINT

 rmgr: XLOG len (rec/tot): 148/148, tx: 0, lsn: 7/F1373F70, prev 7/F1373F20, desc:
CHECKPOINT_ONLINE redo 7/EBA5F1A8; tli 1; prev tli 1; fpw true; xid 7140020; oid 33402; multi 1;
offset 0; oldest xid 723 in DB 1; oldest multi 1 in DB 5; oldest/newest commit timestamp xid: 0/0;
oldest running xid 7140019; online
 rmgr: XLOG len (rec/tot): 148/148, tx: 0, lsn: 7/F7F28E08, prev 7/F7F28DB8, desc:
CHECKPOINT_ONLINE redo 7/F1BF0018; tli 1; prev tli 1; fpw true; xid 7325493; oid 33402; multi 1;
offset 0; oldest xid 723 in DB 1; oldest multi 1 in DB 5; oldest/newest commit timestamp xid: 0/0;
oldest running xid 7325493; online
```

Параметр -e можно не указывать, результат будет тем же самым:

```
pg_waldump -s 7/F1BF0018| grep CHECKPOINT
```

Разряды чисел, относящиеся к смещению внутри WAL-файла можно заменить нулями, результат будет тем же самым:

```
pg_waldump -s 7/F1000000| grep CHECKPOINT
```

3) LSN в журнале будут соответствовать lsn в логе. Посмотрите сообщения в диагностическом логе:

```
postgres@tantor:~$ tail -n 4 $PGDATA/log/postgresql* | grep checkpoint

LOG:  checkpoint complete: wrote 3031 buffers (18.5%); 0 WAL file(s) added, 0 removed, 6 recycled;
write=269.899 s, sync=0.008 s, total=269.926 s; sync files=13, longest=0.003 s, average=0.001 s;
distance=100908 kB, estimate=101792 kB; lsn=7/EB059E10, redo lsn=7/E5712C48
LOG:  checkpoint starting: time
LOG:  disconnection: session time: 0:28:19.585 user=postgres database=postgres host=[local]
LOG:  checkpoint complete: wrote 3215 buffers (19.6%); 0 WAL file(s) added, 0 removed, 6 recycled;
write=269.992 s, sync=0.014 s, total=270.027 s; sync files=14, longest=0.009 s, average=0.001 s;
distance=101681 kB, estimate=101781 kB; lsn=7/F1373F70, redo lsn=7/EBA5F1A8
LOG:  checkpoint starting: time
LOG:  checkpoint complete: wrote 3018 buffers (18.4%); 0 WAL file(s) added, 0 removed, 6 recycled;
write=269.877 s, sync=0.007 s, total=269.901 s; sync files=13, longest=0.003 s, average=0.001 s;
distance=99907 kB, estimate=101593 kB; lsn=7/F7F28E08, redo lsn=7/F1BF0018
LOG:  checkpoint starting: time
```

4) Выполнять не нужно. Посмотрите пример:

```
pg_controldata | grep check | head -n 3
Latest checkpoint location:           8/AA004C8
Latest checkpoint's REDO location:    8/5177990
Latest checkpoint's REDO WAL file:    000000010000000800000005
```

Из-за того, что ведущие нули в LSN не печатаются 8/05177990, можно ошибиться:

```
pg_waldump -s 8/51000000 | grep CHECKPOINT
pg_waldump: error: could not find file "000000010000000800000051": No such file or directory
```

потому что в реальности значение LSN такое:

```
pg_waldump -s 8/05000000 | grep CHECKPOINT
 rmgr: XLOG len (rec/tot): 148/148, tx: 0, lsn: 8/0AA004C8, prev 8/0AA00478, desc:
CHECKPOINT_ONLINE redo 8/5177990; tli 1; prev tli 1; fpw true; xid 7880789; oid 33402; multi 1;
```

```
offset 0; oldest xid 723 in DB 1; oldest multi 1 in DB 5; oldest/newest commit timestamp xid: 0/0;
oldest running xid 7880788; online
 pg_waldump: error: error in WAL record at 8/F9D83B8: invalid record length at 8/F9D83E0: expected
at least 26, got 0
```

При указании полного LSN команда успешно проходит:

```
pg_waldump -s 8/5177990 | grep CHECKPOINT
 rmgr: XLOG len (rec/tot): 148/148, tx: 0, lsn: 8/0AA004C8, prev 8/0AA00478, desc:
CHECKPOINT_ONLINE redo 8/5177990; tli 1; prev tli 1; fpw true; xid 7880789; oid 33402; multi 1;
offset 0; oldest xid 723 in DB 1; oldest multi 1 in DB 5; oldest/newest commit timestamp xid: 0/0;
oldest running xid 7880788; online
```

В выводе **pg_waldump** в строках с `lsn` и `prev` ведущие нули **сохраняются,** а в поле `redo` **не сохраняются**, что создаёт путаницу.
В диагностическом логе и выводе утилиты `pg_controldata` ведущие нули **не сохраняются.**

5) При использовании 16-мегабайтных файлов WAL последние два символа в названии файла совпадают с двумя символами, если добавить ведущий ноль. Пример выполнения команды:

```
pg_controldata | grep check | head -n 3

Latest checkpoint location:           7/F7F28E08
Latest checkpoint's REDO location:    7/F1BF0018
Latest checkpoint's REDO WAL file:    0000000100000007000000F1
Latest checkpoint location:           8/1164B2E8
Latest checkpoint's REDO location:    8/0BC98978
Latest checkpoint's REDO WAL file:    00000001000000080000000B
Latest checkpoint location:           7/FE08BAC0
Latest checkpoint's REDO location:    7/F878C390
Latest checkpoint's REDO WAL file:    0000000100000007000000F8
Latest checkpoint location:           8/0497B1F0
Latest checkpoint's REDO location:    7/FF054A78
Latest checkpoint's REDO WAL file:    0000000100000007000000FF
Latest checkpoint location:           8/0AA004C8
Latest checkpoint's REDO location:    8/05177990
Latest checkpoint's REDO WAL file:    000000010000000800000005
```

В примере нули помеченные красным цветом **pg_controldata** не выводит. *Нули* добавлены для наглядности.
Как визуально понять нужен ноль или не нужен? Число букв после слэша должно быть 8. Так же как и до слэша, но до слэша отсутствие нулей не создаёт неоднозначности. Пример:
`8/BC98978` должно быть `000000008/0BC98978`.

6) Посмотрите пример дампа трёх идущих жруг за другом журнальных записей:

```
pg_waldump -s 8/1164B298 -e 8/1164B3E8
 rmgr: Standby len (rec/tot):  76/76,  tx:       0, lsn: 8/1164B298, prev 8/1164B240,
desc: RUNNING_XACTS nextXid 8232887 latestCompletedXid 8232885 oldestRunningXid 8232886;
1 xacts: 8232886

 rmgr: XLOG    len (rec/tot): 148/148, tx:       0, lsn: 8/1164B2E8, prev 8/1164B298,
desc: CHECKPOINT_ONLINE redo 8/BC98978; tli 1; prev tli 1; fpw true; xid 8064948; oid
33402; multi 1; offset 0; oldest xid 723 in DB 1;
 oldest multi 1 in DB 5; oldest/newest commit timestamp xid: 0/0; oldest running xid
8064947; online

 rmgr: Heap    len (rec/tot):  86/86,  tx: 8232886, lsn: 8/1164B380, prev 8/1164B2E8,
desc: HOT_UPDATE old_xmax: 8232886, old_off: 113, old_infobits: [], flags: 0x20,
new_xmax: 0, new_off: 114, blkref #0: rel 1663/5/25198 blk 1
```

`prev` указывает на LSN начала предыдущей журнальной записи. Используя это поле можно двигаться по WAL в обратном направлении.

в примере до записи о конце контрольной точки присутствуют записи длинны 72-80 байт.

LSN следующей журнальной записи не хранится, потому что хранится длинна записи в `len (rec/tot)`. lsn+len + паддинг до 8 байт = LSN начала следующей записи.

```
select pg_wal_lsn_diff('8/1164B2E8','8/1164B298');
 pg_wal_lsn_diff
-----------------
              80
(1 row)
```

Длинна записи равна `80`, а не `len=76` из-за padding (дополнения до 8 байт в большую сторону).

```
select pg_wal_lsn_diff('8/1164B380','8/1164B2E8');
 pg_wal_lsn_diff
-----------------
             152
(1 row)
```

Длинна записи равна **152**, а не `len=148` из-за padding (дополнения до 8 байт в большую сторону).

Расширение `pg_walinspect`, функции которого имеют возможности, аналогичные утилите **pg_waldump** рассматривается в следующей главе.

Часть 4. Размер директории PGDATA/pg_wal

1) Посмотрите значения параметров, определюящих размер диектории журналов:

```
postgres@tantor:~$ psql -c "\dconfig *wal_size"
List of configuration parameters
  Parameter    | Value
---------------+-------
 max_wal_size | 1GB
 min_wal_size | 80MB
(2 rows)
```

2) Посмотрите сколько места занимает директория с журналами:

```
postgres@tantor:~$ du -h $PGDATA/pg_wal
4.0K    /var/lib/postgresql/tantor-se-16/data/pg_wal/archive_status
945M    /var/lib/postgresql/tantor-se-16/data/pg_wal/
```

Значения могут быть от 900МБ до 1.1ГБ.

3) Хочется уменьшить размер файлов в директории. Поменяйте параметр конфигурации из-за которого в директории занято много места:

```
postgres@tantor:~$
psql -c "alter system set max_wal_size='100MB';"
psql -c "select pg_reload_conf();"
 pg_reload_conf
----------------
 t
(1 row)
```

4) После изменения параметра место в директории не уменьшится. Оно не уменьшится после переключения на новый файл, (если только не выполнилась контрольная точка):

```
postgres@tantor:~$
psql -c "select pg_switch_wal();"
psql -c "select pg_switch_wal();"
psql -c "select pg_switch_wal();"
psql -c "select pg_switch_wal();"

 pg_switch_wal
---------------
 249/ABBE616A
(1 row)

 pg_switch_wal
---------------
 249/AC000000
(1 row)

 pg_switch_wal
---------------
 249/AC00008A
(1 row)

 pg_switch_wal
---------------
 249/AD00008A
(1 row)

postgres@tantor:~$ du -h $PGDATA/pg_wal
4.0K    /var/lib/postgresql/tantor-se-16/data/pg_wal/archive_status
897M    /var/lib/postgresql/tantor-se-16/data/pg_wal/
```

В примере выполнилась контрольная точка по времени.

5) Выполните контрольную точку:

```
postgres@tantor:~$ psql -c "checkpoint;"
du -h $PGDATA/pg_wal
CHECKPOINT
4.0K    /var/lib/postgresql/tantor-se-16/data/pg_wal/archive_status
881M    /var/lib/postgresql/tantor-se-16/data/pg_wal/
```

После контрольной точки место уменьшилось.

6) Выполните контрольную точку:

```
postgres@tantor:~$ psql -c "checkpoint;"
CHECKPOINT
postgres@tantor:~$ du -h $PGDATA/pg_wal/
4.0K    /var/lib/postgresql/tantor-se-16/data/pg_wal/archive_status
881M    /var/lib/postgresql/tantor-se-16/data/pg_wal/
```

Место не уменьшилось, так как не было активности.

7) Выполните переключение файла журнала для создания активности и контрольную точку:

```
postgres@tantor:~$ psql -c "select pg_switch_wal();"
 pg_switch_wal
---------------
 249/AF000292
(1 row)

postgres@tantor:~$ psql -c "checkpoint;"
CHECKPOINT
postgres@tantor:~$ du -h $PGDATA/pg_wal/
4.0K    /var/lib/postgresql/tantor-se-16/data/pg_wal/archive_status
```

```
865M    /var/lib/postgresql/tantor-se-16/data/pg_wal/
```

После контрольной точки занятое журналами место уменьшилось.

8) Посмотрите сообщения в журнале, чтобы посмотреть сколько файлов WAL было удалено (removed):

```
postgres@tantor:~$ cat $PGDATA/log/postgresql*

21:42:45.914 MSK [331791] LOG:  checkpoint starting: time
21:42:49.240 MSK [331791] LOG:  checkpoint complete: wrote 31 buffers (0.2%); 0 WAL file(s) added,
0 removed, 0 recycled; write=3.270 s, sync=0.019 s, total=3.326 s; sync files=3, longest=0.012 s,
average=0.006 s; distance=226 kB, estimate=397099 kB; lsn=249/ABBE6070, redo lsn=249/ABBE6028
11:20:33.612 MSK [331789] LOG:  received SIGHUP, reloading configuration files
11:20:33.613 MSK [331789] LOG:  parameter "max_wal_size" changed to "100MB"
11:21:36.243 MSK [331791] LOG:  checkpoint starting: wal
11:21:36.314 MSK [331791] LOG:  checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added, 3
removed, 0 recycled; write=0.001 s, sync=0.001 s, total=0.071 s; sync files=0, longest=0.000 s,
average=0.000 s; distance=36968 kB, estimate=361086 kB; lsn=249/AE000070, redo lsn=249/AE000028
11:23:45.824 MSK [331791] LOG:  checkpoint starting: immediate force wait
11:23:45.877 MSK [331791] LOG:  checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added, 1
removed, 0 recycled; write=0.001 s, sync=0.001 s, total=0.054 s; sync files=0, longest=0.000 s,
average=0.000 s; distance=16384 kB, estimate=326616 kB; lsn=249/AF0000B8, redo lsn=249/AF000070
11:23:55.592 MSK [331791] LOG:  checkpoint starting: immediate force wait
11:23:55.632 MSK [331791] LOG:  checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added, 0
removed, 0 recycled; write=0.001 s, sync=0.001 s, total=0.041 s; sync files=0, longest=0.000 s,
average=0.000 s; distance=0 kB, estimate=293954 kB; lsn=249/AF000198, redo lsn=249/AF000150
11:24:07.754 MSK [331791] LOG:  checkpoint starting: immediate force wait
11:24:07.801 MSK [331791] LOG:  checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added, 1
removed, 0 recycled; write=0.001 s, sync=0.001 s, total=0.047 s; sync files=0, longest=0.000 s,
average=0.000 s; distance=16383 kB, estimate=266197 kB; lsn=249/B0000070, redo lsn=249/B0000028
```

`distance=36968 kB` округляется в большую сторону до 16Мб (размер сегмента) и получается что checkpointer удалил три файла (`3 removed`). Это соответствует изменению места, занимаемого файлами в директории: `945M-897M=48Мб`

`distance=16384 kB` соответствует удалению одного файла (`1 removed`). Это соответствует изменению места, занимаемого файлами в директории: `897M-881M=16Мб` и `881M-865M=16Мб`

9) Использовать для удаления файлов из директории pg_wal устилиту `pg_archivecleanup` нельзя. Либо утилита ничего не удалит:

```
pg_controldata | grep "REDO WAL"
Latest checkpoint's REDO WAL file:    000000010000024A0000004B
pg_archivecleanup -n $PGDATA/pg_wal  000000010000024A0000004B
```

Либо удалит файл, который нужен для восстановления и без которого кластер будет разрушен, если указать произвольное имя файла в директории **pg_wal**:

```
pg_controldata | grep "REDO WAL"
Latest checkpoint's REDO WAL file:    000000010000024A0000004B
pg_archivecleanup -n $PGDATA/pg_wal  000000010000024A00000088 | grep 0004B
/var/lib/postgresql/tantor-se-16/data/pg_wal/000000010000024A0000004B
```

Процесс контрольной точки создаёт файлы WAL заранее, чтобы при быстром переключении WAL не было задержек. В директории файлы **000000010000024A0000004B ... 000000010000024A00000088** не являются историческими.
Удалять файлы в директории **pg_wal** на работающем экземпляре опасно, так как есть вероятность удалить файл, нужный для восстановления, реплики, подписки.

9) Верните значение в значение по умолчанию:

```
postgres@tantor:~$
psql -c "alter system reset max_wal_size;"
psql -c "select pg_reload_conf();"
 pg_reload_conf
----------------
 t
(1 row)
```

10) Выполнять этот пункт не нужно. Посмотрите следующий пример.

Останавливается экземпляр:

```
postgres@tantor:~$ pg_ctl stop
waiting for server to shut down.... done
server stopped
```

Удаляется директория кластера:

```
postgres@tantor:~$ rm -rf $PGDATA/*
```

Создаётся новый кластер:

```
postgres@tantor:~$ initdb -k
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "en_US.UTF-8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are enabled.

fixing permissions on existing directory /var/lib/postgresql/tantor-se-16/data ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting default time zone ... Europe/Moscow
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

initdb: warning: enabling "trust" authentication for local connections
initdb: hint: You can change this by editing pg_hba.conf or using the option -A, or --auth-local and --auth-
host, the next time you run initdb.

Success. You can now start the database server using:

    pg_ctl -D /var/lib/postgresql/tantor-se-16/data -l logfile start
```

Запускается экземпляр:

```
postgres@tantor:~$ sudo restart
```

В директории только один файл журнала:

```
postgres@tantor:~$ ls $PGDATA/pg_wal
000000010000000000000001  archive_status
```

После переключения журнала и контрольной точки:

```
postgres@tantor:~$ psql -c "select pg_switch_wal();"
 pg_switch_wal
---------------
 0/175C7E2
(1 row)
```

```
postgres@tantor:~$ psql -c "checkpoint;"
CHECKPOINT
```

добавляется ещё один файл журнала:

```
postgres@tantor:~$ ls $PGDATA/pg_wal
000000010000000000000002  000000010000000000000003  archive_status
```

После еще одного переключения и контрольной точки не добавляется:

```
postgres@tantor:~$ psql -c "select pg_switch_wal();"
 pg_switch_wal
---------------
 0/200016A
(1 row)

postgres@tantor:~$ psql -c "checkpoint;"
CHECKPOINT

postgres@tantor:~$ ls $PGDATA/pg_wal
000000010000000000000003  000000010000000000000004  archive_status
```

Для генерации большого объема изменений запускается создание таблиц pgbench с масштабом 64:

```
postgres@tantor:~$ pgbench -i -s 64
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)...
6400000 of 6400000 tuples (100%) done (elapsed 15.78 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 22.08 s (drop tables 0.00 s, create tables 0.03 s, client-side generate 15.92 s, vacuum 0.40 s, primary
keys 5.74 s).
```

Число журналов и увеличилось:

```
postgres@tantor:~$ ls $PGDATA/pg_wal

000000010000000000000025  000000010000000000000032  000000010000000000000003F  000000010000000000000004C
000000010000000000000026  000000010000000000000033  000000010000000000000040  000000010000000000000004D
000000010000000000000027  000000010000000000000034  000000010000000000000041  000000010000000000000004E
000000010000000000000028  000000010000000000000035  000000010000000000000042  000000010000000000000004F
000000010000000000000029  000000010000000000000036  000000010000000000000043  000000010000000000000050
00000001000000000000002A  000000010000000000000037  000000010000000000000044  000000010000000000000051
00000001000000000000002B  000000010000000000000038  000000010000000000000045  000000010000000000000052
00000001000000000000002C  000000010000000000000039  000000010000000000000046  000000010000000000000053
00000001000000000000002D  00000001000000000000003A  000000010000000000000047  000000010000000000000054
00000001000000000000002E  00000001000000000000003B  000000010000000000000048  000000010000000000000055
00000001000000000000002F  00000001000000000000003C  000000010000000000000049  000000010000000000000056
000000010000000000000030  00000001000000000000003D  00000001000000000000004A  archive_status
000000010000000000000031  00000001000000000000003E  00000001000000000000004B

postgres@tantor:~$ du -h $PGDATA/pg_wal
4.0K    /var/lib/postgresql/tantor-se-16/data/pg_wal/archive_status
801M    /var/lib/postgresql/tantor-se-16/data/pg_wal
```

В директории журналами занято **801M**.

Уменьшите значение параметра `max_wal_size`:

```
postgres@tantor:~$
psql -c "alter system set max_wal_size='100MB';"
psql -c "select pg_reload_conf();"
ALTER SYSTEM
 pg_reload_conf
----------------
 t
(1 row)
```

Выполните переключений журналов и контрольных точек по числу файлов журналов:

```
postgres@tantor:~$
for (( i=0; i <= 50; i+=1 ));
do
psql -c "select pg_switch_wal();" > /dev/null 2> /dev/null
psql -c "checkpoint;" > /dev/null
done
```

Цикл симулирует работу экземпляра и постепенное уменьшение числа WAL-сегментов. Число файлов уменьшилось. Суммарный размер файлов журналов уменьшился до значения параметра `min_wal_size`:

```
postgres@tantor:~$ ls $PGDATA/pg_wal

000000010000000000000068  000000010000000000000006A  000000010000000000000006C
000000010000000000000069  000000010000000000000006B  archive_status

postgres@tantor:~$ du -h $PGDATA/pg_wal
4.0K    /var/lib/postgresql/tantor-se-16/data/pg_wal/archive_status
81M     /var/lib/postgresql/tantor-se-16/data/pg_wal

postgres@tantor:~$ psql -c "show min_wal_size;"
 min_wal_size
--------------
 80MB
(1 row)
```

Если нужно быстро уменьшить число сегментов до минимума, игнорируя расчетные значения checkpointer, то можно выключить параметр `wal_recycle` и выполнить одну контрольную точку. Процесс checkpointer удалит все WAL-сегменты до минимума. Останутся только те сегмены, которые должны удерживаться. Сегментов, созданных (переименованных) заранее не будет. Параметр можно будет обрано включить. Изменение значения параметра `wal_recycle` не требует рестарта экземпляра, достаточно перечитать конфигурацию. Пример:

```
postgres@tantor:~$
psql -c "alter system set wal_recycle = off;"
psql -c "select pg_reload_conf();"
psql -c "checkpoint;"
psql -c "alter system set wal_recycle = on;"
psql -c "select pg_reload_conf();"
ALTER SYSTEM
 pg_reload_conf
----------------
 t
(1 row)

CHECKPOINT
ALTER SYSTEM
 pg_reload_conf
----------------
 t
(1 row)
```

Возврат **max_wal_size** к значению по умолчанию и установка параметров конфигурации в значения до удаления кластера:

```
postgres@tantor:~$
psql -c "alter system set recovery_init_sync_method = 'syncfs'"
psql -c "alter system set track_io_timing = 'on'"
psql -c "alter system set logging_collector = 'on'"
psql -c "alter system reset max_wal_size;"
psql -c "create extension if not exists pageinspect;"
psql -c "create extension if not exists pg_visibility;"
psql -c "create extension if not exists pg_columnar;"
psql -c "create extension if not exists bloom;"
psql -c "create extension if not exists pgstattuple;"
psql -c "create extension if not exists pg_buffercache;"
psql -c "alter system set shared_preload_libraries='pg_prewarm';"
psql -c "create extension if not exists pg_prewarm;"
sudo restart
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
CREATE EXTENSION
CREATE EXTENSION
CREATE EXTENSION
CREATE EXTENSION
CREATE EXTENSION
CREATE EXTENSION
ALTER SYSTEM
CREATE EXTENSION
```

Результат этого пункта: файлы журналов создаются процессом checkpointer в конце контрольной точки. Число файлов определяется активностью генерации журналов. Число файлов доходит до значения примерно равному **max_wal_size**. При уменьшении значения постепенно после выполнения контрольной точки число файлов уменьшается и доходит до **min_wal_size**. Число файлов не уменьшается, если были контрольные точки по объему.
Также журналы могут удерживаются слотами репликации.

Практика к главе 14

Часть 1. Статистика ввода-вывода в представлении pg_stat_io

Посмотрите названия столбцов представления `pg_stat_io`:

```
postgres=# select * from pg_stat_io where backend_type='checkpointer' limit 1
\gx
-[ RECORD 1 ]--+------------------------------
backend_type    | checkpointer
object          | relation
context         | normal
reads           |
read_time       |
writes          | 286
write_time      | 4.04
writebacks      | 286
writeback_time  | 6.932
extends         |
extend_time     |
op_bytes        | 8192
hits            |
evictions       |
reuses          |
fsyncs          | 135
fsync_time      | 313.348
stats_reset     | 2035-01-01 01:01:26.002532+03
```

В представлении много столбцов. Чтобы столбцы поместились на странице можно использовать псевдонимы для столбцов:

```
postgres=# select backend_type, left(object,4) obj, context, writes w,
round(write_time::numeric,2) wt, writebacks wb, round(writeback_time::numeric,2)
wbt, extends e, extend_time et, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric,2) fst from pg_stat_io;
      backend_type     | obj  |  context  | w  |  wt  | wb |  wbt  |  e  |  et   | ev | ru | fs |  fst
-----------------------+------+-----------+----+------+----+-------+-----+-------+----+----+----+--------
 autovacuum launcher   | rela | bulkread  |  0 | 0.00 |  0 |  0.00 |     |       |  0 |  0 |    |
 autovacuum launcher   | rela | normal    |  0 | 0.00 |  0 |  0.00 |     |       |  0 |    |  0 |   0.00
 autovacuum worker     | rela | bulkread  |  0 | 0.00 |  0 |  0.00 |     |       |  0 |  0 |    |
 autovacuum worker     | rela | normal    |  0 | 0.00 |  0 |  0.00 |   3 | 0.085 |  0 |    |  0 |   0.00
 autovacuum worker     | rela | vacuum    |  0 | 0.00 |  0 |  0.00 |   0 |     0 |  0 |  0 |    |
 client backend        | rela | bulkread  |  0 | 0.00 |  0 |  0.00 |     |       |  0 |  0 |    |
 client backend        | rela | bulkwrite |  0 | 0.00 |  0 |  0.00 |   6 | 0.136 |  0 |  0 |    |
 client backend        | rela | normal    |  0 | 0.00 |  0 |  0.00 | 165 | 1.967 |  0 |    |  0 |   0.00
 client backend        | rela | vacuum    | 24 | 0.24 |  0 |  0.00 |   0 |     0 |  0 | 24 |    |
 client backend        | temp | normal    | 50 | 0.39 |    |       | 151 | 1.567 | 51 |    |    |
 background worker     | rela | bulkread  |  0 | 0.00 |  0 |  0.00 |     |       |  0 |  0 |    |
 background worker     | rela | bulkwrite |  0 | 0.00 |  0 |  0.00 |   0 |     0 |  0 |  0 |    |
 background worker     | rela | normal    |  0 | 0.00 |  0 |  0.00 |   0 |     0 |  0 |    |  0 |   0.00
 background worker     | rela | vacuum    |  0 | 0.00 |  0 |  0.00 |   0 |     0 |  0 |  0 |    |
 background worker     | temp | normal    |  0 | 0.00 |    |       |   0 |     0 |  0 |    |    |
 background writer     | rela | normal    |  0 | 0.00 |  0 |  0.00 |     |       |    |    |  0 |   0.00
 checkpointer          | rela | normal    | 94 | 4.76 | 94 |  2.70 |     |       |    |    | 62 | 160.88
 standalone backend    | rela | bulkread  |  0 | 0.00 |  0 |  0.00 |     |       |  0 |  0 |    |
 standalone backend    | rela | bulkwrite |  0 | 0.00 |  0 |  0.00 |   0 |     0 |  0 |  0 |    |
 standalone backend    | rela | normal    |  0 | 0.00 |  0 |  0.00 |   0 |     0 |  0 |    |  0 |   0.00
 standalone backend    | rela | vacuum    |  0 | 0.00 |  0 |  0.00 |   0 |     0 |  0 |  0 |    |
 startup               | rela | bulkread  |  0 | 0.00 |  0 |  0.00 |     |       |  0 |  0 |    |
 startup               | rela | bulkwrite |  0 | 0.00 |  0 |  0.00 |   0 |     0 |  0 |  0 |    |
 startup               | rela | normal    |  0 | 0.00 |  0 |  0.00 |   0 |     0 |  0 |    |  0 |   0.00
 startup               | rela | vacuum    |  0 | 0.00 |  0 |  0.00 |   0 |     0 |  0 |  0 |    |
 walsender             | rela | bulkread  |  0 | 0.00 |  0 |  0.00 |     |       |  0 |  0 |    |
 walsender             | rela | bulkwrite |  0 | 0.00 |  0 |  0.00 |   0 |     0 |  0 |  0 |    |
 walsender             | rela | normal    |  0 | 0.00 |  0 |  0.00 |   0 |     0 |  0 |    |  0 |   0.00
 walsender             | rela | vacuum    |  0 | 0.00 |  0 |  0.00 |   0 |     0 |  0 |  0 |    |
 walsender             | temp | normal    |  0 | 0.00 |    |       |   0 |     0 |  0 |    |    |
(30 rows)
```

В тех полях где значения пустые - значений не может быть в принципе. В тех полях где нули могут быть ненулевые значения. Правила в каких полях значений не может быть перечислены ниже.

Характеристики представления **pg_stat_io**:

Столбцы writeback_time и fsync_time заполняются только, если параметр конфигурации **track_io_timing = on**.
Основной интерес представляют столбцы writebacks и fsyncs, так как именно они приводят к передаче системных вызовов в операционную систему.

writebacks выполняются:
a) серверными процессами при расширении файлов (extends) и поиске буфера на вытеснение (вызовом функции GetVictimBuffer(..))
b) процессами bgwriter и checkpointer в контексте normal. Используется очередь процесса checkpointer.

fsyncs:
a) передаются процессу checkpointer для выполнения в конце контрольной точки один раз по каждому файлу
b) если не могут быть переданы, то выполняются самим процессом
c) при работе с буферными кольцами не подсчитываются.

В столбце object могут быть значения: relation (объект постоянного хранения) и temp relation (временный объект). С временными объектами writebacks и fsyncs не выполняются, так как гарантия записи временным таблицам не нужна - при падении процесса или после запуска экземпляра файлы должны быть удалены.
В столбце context могут быть значения: normal, bulkread, bulkwrite, vacuum. Последние три это буферные кольца. reuses непустые только у колец, у контекста normal отсутствуют. У колец fsyncs не считаются. У bulkread отсутствуют extends, так как при чтении файлы не меняют размер.
Процессы autovacuum launcher и autovacuum worker не работают с кольцами типа bulkwrite. Процесс autovacuum launcher не работает с кольцом типа vacuum.
Процессы bgwriter и checkpointer с кольцами не работают.
У bgwriter и checkpointer в столбцах reads, hits, evictions всегда нули.
У bgwriter и autovacuum launcher в столбце extends нули, так как они не увеличивают размеры файлов.
По процессу loggingg collector статистика не собирается, так как процесс loggingg collector не имеет доступа к разделяемой памяти.
По процессу архиватора статистика не собирается, так как archiver запускает команды или программы, которые и выполняют операции ввода-вывода.
Процессы WAL Receiver и WAL Writer отсутствуют в pg_stat_io (не реализовано).

При использовании колец, серверный процессы могут добавить к кольцу буфер и заменить в добавляемом буфере блок на блок объекта, с которым идет работа в кольце. Такое бывает когда:
a) в буферном кольце нет или мало буферов, так как команда только начала работать
b) для замены буфера в кольце на другой буфер, так как прежний буфер закреплен или используется другим процессом и в буфере нельзя заменить блок. Такой буфер исключается из кольца.

evictions заполняется с любым контекстом - и кольцами и в контексте normal. evictions означает, что в буфере был заменён блок.
reused заполняется только для колец.
В чем отличия evictions от reused при работе с кольцами?
Замена блока в буфере учитывается в статистике evictions соответствующего типа (context) кольца. Если же блок находится в буфере относящимся к кольцу и идёт замена блока

на другой блок в буфере того же кольца, то замена блока учитывается в статистике `reused` этого типа (`context`) кольца. Но если блок таблицы закреплен (pinned) другим процессом, то при прицеплении буфера в кольцо увеличивается статистика `evictions`, а не `reused`.

Столбцы представления **pg_stat_io**:
`op_bytes` - значения всегда равны размеру блока данных: 8192.
`hits` - почти не представляет диагностической ценности.

2) Выполните команды:

```
postgres@tantor:~$
psql -c "alter system set track_io_timing = on;"
psql -c "alter system reset shared_buffers;"
psql -c "alter system set logging_collector = off;"
psql -c "select pg_stat_reset_shared('io');"
psql -c "select pg_stat_reset_shared('bgwriter');"
pg_ctl stop
pg_ctl start
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
 pg_stat_reset_shared
----------------------

(1 row)


 pg_stat_reset_shared
----------------------

(1 row)

LOG:  received fast shutdown request
LOG:  database system is shut down
...
 done
server stopped
 done
server started
```

Отключение `logging_collector` и перезапуск экземпляра утилитой `pg_ctl` позволит выводить диагностический лог в терминал. Это удобно для наблюдения за контрольными точками. Статистики в представлениях `pg_stat_io` и `pg_stat_bgwriter` связаны и выдают пересекающиеся значения, поэтому их стоит обнулять одновременно. Представление `pg_stat_io` появилось в 16 версии PostgreSQL. Представление `pg_stat_bgwriter` использовалось до 16 версии и системы мониторинга и формулы для настройки контрольных точек и параметров bgwriter связаны с этим представлением, поэтому pg_stat_bgwriter может использоваться для мониторинга и анализа статистик. Начиная с 17 версии из `pg_stat_bgwriter` убраны столбцы, относящиеся к серверным процессам и checkpointer и появилось представление pg_stat_checkpointer. В практике не приведены примеры запросов к pg_stst_bgwriter, представление актуально, если эксплуатируются версии PostgreSQL до 16 и если нужно мигрировать скрипты мониторинга на новые версии.

3) Откройте второй терминал linux и выполните:

```
postgres@tantor:~$ pgbench -i -s 64
pgbench -T 6000 -c 64 -P 10
```

Запускается тест в 64 сессиях.

4) В первом терминале запустите psql и выполните команду:

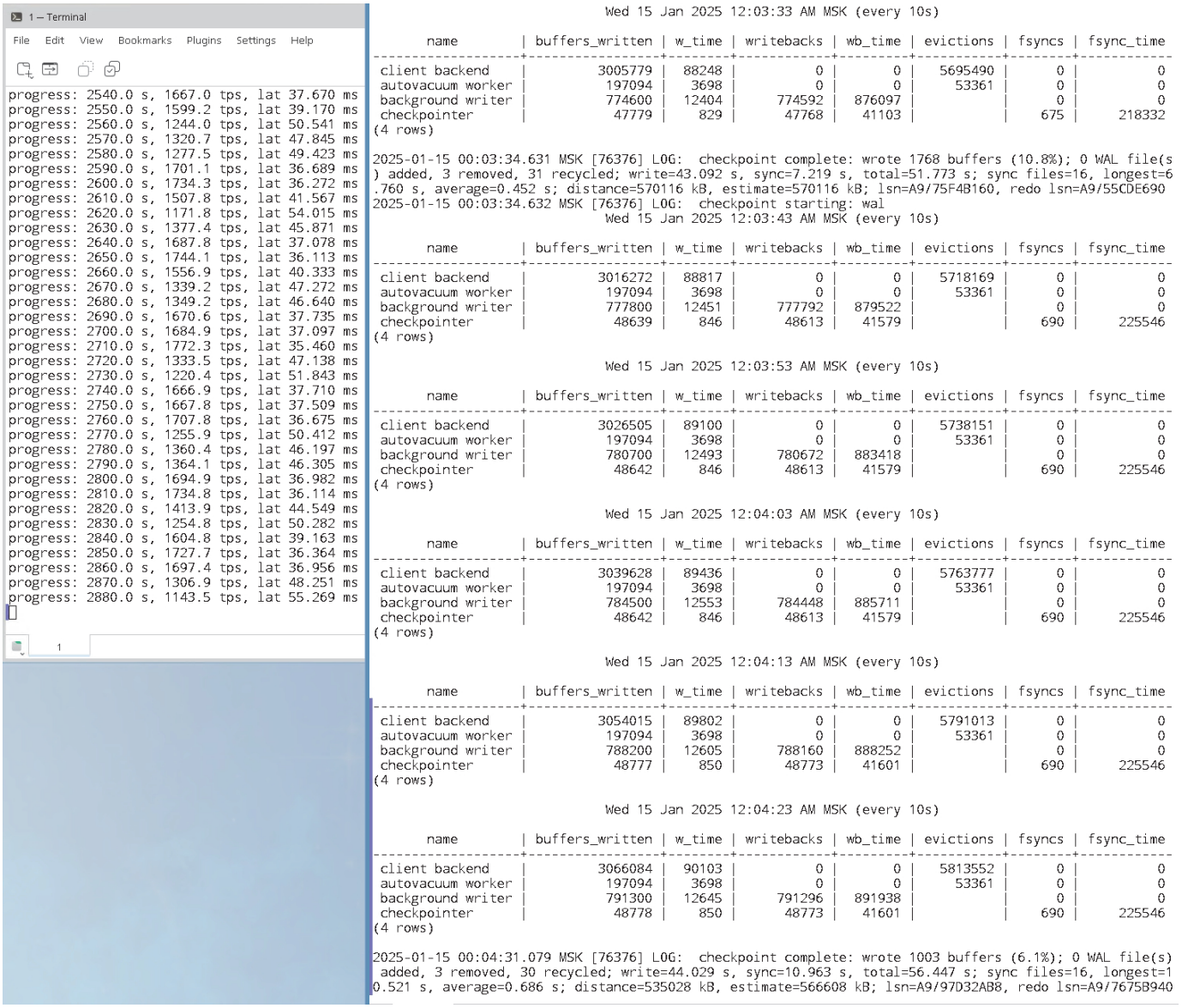
```
postgres@tantor:~$ psql
postgres=# select backend_type name, sum(writes) buffers_written,
round(sum(write_time)) w_time, sum(writebacks) writebacks, sum(evictions)
evictions, sum(fsyncs) fsyncs, round(sum(fsync_time)) fsync_time from pg_stat_io
group by backend_type having sum(writes)> 0 or sum(writebacks)> 0 or
sum(fsyncs)>0 or sum(evictions)>0;
      name      |buffers_written |w_time |writebacks |evictions |fsyncs |fsync_time
----------------+----------------+-------+-----------+----------+-------+----------
client backend  |         102848 |  1100 |         0 |        0 |     0 |         0
checkpointer    |            105 |     3 |        96 |          |     0 |         0
(2 rows)
```

5) В первом терминале выполните команду:

```
postgres=# \watch 10
```

Эта команда psql будет повторять последнюю команду с интервалом в 10 секунд. Прервать повторение можно комбинацией клавш <ctrl+c>.

6) Окна терминалов будут выглядеть так:



Можно наблюдать как меняются показатели ввода вывода по процессам.

fsyncs по файлам в табличных пространствах выполняет процесс checkpointer и поле обновляется только после контрольной точки любого типа. В терминал выводятся диагностические сообщении о выполнении контрольных точек. В пример видна контрольная точка "по требованию" - из-за генерации большого объема WAL. max_wal_size=1GB и по достижении **distance=570116 kB** начинается контрольная точка.

Синхронизировались блоки **sync files=16** файлов, что соответствует разнице в столбце fsyncs 690-675=15. Было записано **wrote 1768 buffers**. Выполнение системных вызовов sync_file_range(fd, offset, nbytes, SYNC_FILE_RANGE_WRITE).

заняло **sync=7.219 s**, что соответствует значению в столбце fsync_time=225546-218332=7,214секунд.

Блоки из кэша буферов посылались в очередь на запись (процессу контрольной точки):

серверными процессами buffers_written **3066084**-**3005779**=60305. Серверные процессы которые потратили на это w_time 90103-88248=1,855 секунд.

background writer столбец buffers_written **791300**-**774600**=16700 за 12645-12404=241 миллисекунду.

При этом процесс фоновой записи сам послал блоки в страничный кэш linux примерно в таком же количестве и затратил на это wb_time 891938-876097=15,841 секунд.

7) Довольно большое количество идентификаторов блоков, посылались в очередь на запись процессу chekpointer. Однако, процесс контрольной точки послал в страничный кэш операционной системы 48773-47768=1005 блоков.

Для расчета значений полей, которые менялись каждый раз при запросе, нужно брать результаты запросов, соответствующие интервалам между контрольными точками. Например запросы сразу после контрольной точки. Точные значения получить нельзя, но оценить можно.

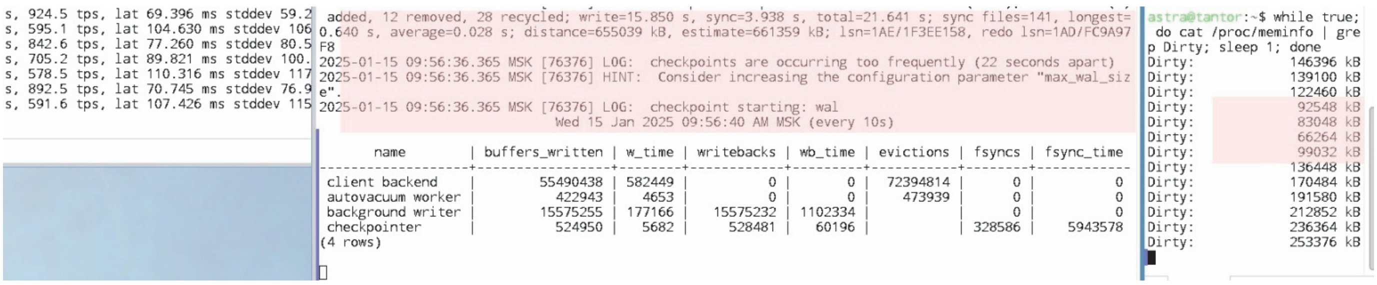
Один вызов sync_file_range, в среднем, выполнялся 21833/675=32,345 миллисекунд.

```
                            12:03:33 AM MSK (every 10s)

       name        | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
-------------------+-----------------+--------+------------+---------+-----------+--------+------------
 client backend    |         3005779 |  88248 |          0 |       0 |   5695490 |      0 |          0
 autovacuum worker |          197094 |   3698 |          0 |       0 |     53361 |      0 |          0
 background writer |          774600 |  12404 |     774592 |  876097 |           |      0 |          0
 checkpointer      |           47779 |    829 |      47768 |   41103 |           |    675 |     218332
(4 rows)

   00:03:34.631 MSK [76376] LOG:  checkpoint complete: wrote 1768 buffers (10.8%); 0 WAL file(s) added, 3
removed, 31 recycled; write=43.092 s, sync=7.219 s, total=51.773 s; sync files=16, longest=6.760 s,
average=0.452 s; distance=570116 kB, estimate=570116 kB; lsn=A9/75F4B160, redo lsn=A9/55CDE690
   00:03:34.632 MSK [76376] LOG:  checkpoint starting: wal
                            12:03:43 AM MSK (every 10s)

       name        | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
-------------------+-----------------+--------+------------+---------+-----------+--------+------------
 client backend    |         3016272 |  88817 |          0 |       0 |   5718169 |      0 |          0
 autovacuum worker |          197094 |   3698 |          0 |       0 |     53361 |      0 |          0
 background writer |          777800 |  12451 |     777792 |  879522 |           |      0 |          0
 checkpointer      |           48639 |    846 |      48613 |   41579 |           |    690 |     225546
(4 rows)

                            12:03:53 AM MSK (every 10s)

       name        | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
-------------------+-----------------+--------+------------+---------+-----------+--------+------------
 client backend    |         3026505 |  89100 |          0 |       0 |   5738151 |      0 |          0
 autovacuum worker |          197094 |   3698 |          0 |       0 |     53361 |      0 |          0
 background writer |          780700 |  12493 |     780672 |  883418 |           |      0 |          0
 checkpointer      |           48642 |    846 |      48613 |   41579 |           |    690 |     225546
(4 rows)

                            12:04:03 AM MSK (every 10s)

       name        | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
-------------------+-----------------+--------+------------+---------+-----------+--------+------------
 client backend    |         3039628 |  89436 |          0 |       0 |   5763777 |      0 |          0
 autovacuum worker |          197094 |   3698 |          0 |       0 |     53361 |      0 |          0
 background writer |          784500 |  12553 |     784448 |  885711 |           |      0 |          0
 checkpointer      |           48642 |    846 |      48613 |   41579 |           |    690 |     225546
(4 rows)
```

```
                    12:04:13 AM MSK (every 10s)

       name        | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
-------------------+-----------------+--------+------------+---------+-----------+--------+------------
 client backend    |         3054015 |  89802 |          0 |       0 |   5791013 |      0 |          0
 autovacuum worker |          197094 |   3698 |          0 |       0 |     53361 |      0 |          0
 background writer |          788200 |  12605 |     788160 |  888252 |           |      0 |          0
 checkpointer      |           48777 |    850 |      48773 |   41601 |           |    690 |     225546
(4 rows)

                    12:04:23 AM MSK (every 10s)

       name        | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
-------------------+-----------------+--------+------------+---------+-----------+--------+------------
 client backend    |         3066084 |  90103 |          0 |       0 |   5813552 |      0 |          0
 autovacuum worker |          197094 |   3698 |          0 |       0 |     53361 |      0 |          0
 background writer |          791300 |  12645 |     791296 |  891938 |           |      0 |          0
 checkpointer      |           48778 |    850 |      48773 |   41601 |           |    690 |     225546
(4 rows)

   00:04:31.079 MSK [76376] LOG:  checkpoint complete: wrote 1003 buffers (6.1%); 0 WAL file(s) added, 3 removed, 30 recycled; write=44.029 s, sync=10.963 s, total=56.447 s; sync files=16, longest=10.521 s, average=0.686 s; distance=535028 kB, estimate=566608 kB; lsn=A9/97D32AB8, redo lsn=A9/7675B940
   00:04:31.080 MSK [76376] LOG:  checkpoint starting: wal
                    12:04:33 AM MSK (every 10s)

       name        | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
-------------------+-----------------+--------+------------+---------+-----------+--------+------------
 client backend    |         3074683 |  90654 |          0 |       0 |   5830219 |      0 |          0
 autovacuum worker |          197590 |   3707 |          0 |       0 |     54303 |      0 |          0
 background writer |          793500 |  12674 |     793472 |  897126 |           |      0 |          0
 checkpointer      |           49146 |    854 |      49131 |   42288 |           |    704 |     236498
(4 rows)
```

8) Запустите третий терминал и в нём выполните команду:

astra@tantor:~$ **while true; do cat /proc/meminfo | grep Dirty; sleep 1; done**
Пример окна с командой справа:



Можно наблюдать в динамике, что за ~4 секунды (`sync=3.938 s` в последующем сообщении о контрольной точке) перед появлением сообщения о контрольной точке объем грязных страниц в страничном кэше linux уменьшается. После контрольной точки увеличивается, так как pgbench (левое окно на картинке) грязнит страницы.

Прервать цикл можно нажав в окне комбинацию клавиш <ctrl+c> и если окно терминала не нужно, то закрыть его.

9) Остановите тест pgbench комбинацией клавиш <ctrl+c>.

Часть 2. Выполнение fsyncs при остановленном checkpointer

1) Выполните команды:

```
postgres@tantor:~$
time psql -f initdrop.sql > /dev/null 2> /dev/null
psql -c "\dconfig *flush*"
sudo restart

real    0m5.547s
user    0m0.009s
```

```
sys     0m0.000s

List of configuration parameters
       Parameter        | Value
------------------------+-------
 backend_flush_after    | 0
 bgwriter_flush_after   | 512kB
 checkpoint_flush_after | 256kB
 wal_writer_flush_after | 1MB
(4 rows)
```

Команды удаляют тестовые таблицы, если они есть, выводят размеры буферов и перезапускают экземпляр.

2) Найдите номера процессов checkpointer и background writer экземпляра:

```
postgres@tantor:~$ ps -ef | egrep "backgr|checkp"
postgres  286008  286006  0 Jan18 ?        00:00:00 postgres: 15/main: checkpointer
postgres  286009  286006  0 Jan18 ?        00:00:12 postgres: 15/main: background writer
postgres  421548  421544  0 21:32 ?        00:00:03 postgres: checkpointer
postgres  421549  421544  0 21:32 ?        00:00:00 postgres: background writer
postgres  421692  421352  0 21:51 pts/0    00:00:00 grep -E backgr|checkp
```

3) Запустите во втором терминале psql и команды:

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)
```

Команды обнуляют статистику ввода-вывода в двух представлениях.

4) Запустите во втором терминале psql и команды:

```
postgres=#
select backend_type, context, writes w, round(write_time::numeric,2) wt,
writebacks wb, round(writeback_time::numeric,2) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
\watch 1
```

Команды выводят статистику из представления **pg_stat_io** с периодичностью раз в секунду:

```
  backend_type  | context | w |  wt   | wb | wbt   | ex | et | hits | ev | ru | fs | fst
----------------+---------+---+-------+----+-------+----+----+------+----+----+----+-----
 client backend | normal  | 0 | 0.00 |  0 | 0.00 |  0 | 0  |   16 |  0 |    |  0 |  0
(1 row)
```

5) Запустите третий терминал и в нём выполните команду:

```
astra@tantor:~$ while true; do cat /proc/meminfo | grep Dirty; sleep 5; done
```

6) Запустите скрипт теста в первом терминале:

```
postgres@tantor:~$
time psql -f init.sql > /dev/null 2> /dev/null
```

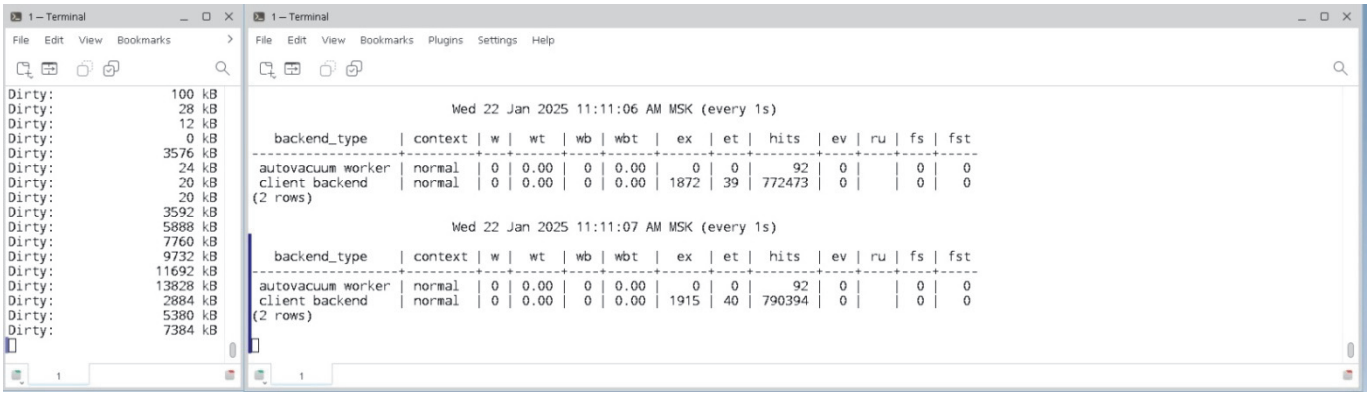
Второй скрипт закончит работать через 2m23:

```
real    2m23.289s
user    0m0.396s
sys     0m0.168s
```

7) Пока скрипт работает наблюдайте за статистикой ввода-вывода:

```
   backend_type     | context | w |  wt  | wb | wbt  |  ex  | et  |  hits   | ev | ru | fs | fst
--------------------+---------+---+------+----+------+------+-----+---------+----+----+----+-----
 autovacuum worker | normal  | 0 | 0.00 |  0 | 0.00 |    0 |   0 |   19814 |  0 |    |  0 |   0
 autovacuum worker | vacuum  | 0 | 0.00 |  0 | 0.00 |    0 |   0 |   16884 |  0 |  0 |    |
 client backend    | normal  | 0 | 0.00 |  0 | 0.00 | 5553 | 120 | 2310438 |  0 |    |  0 |   0
(3 rows)
```

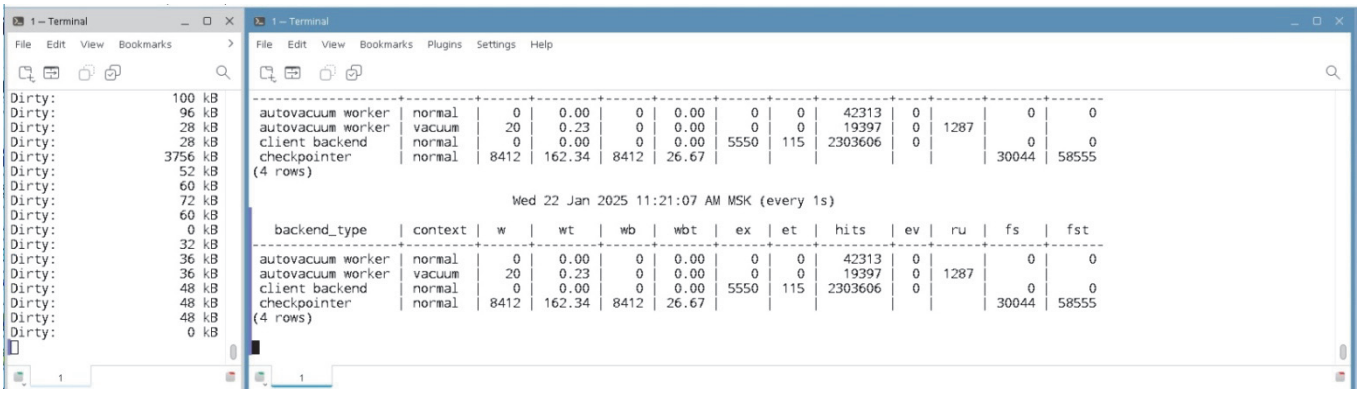
В пустых полях значений не может быть в принципе. Например, при работе с буферным кольцом vacuum не могут посылаться fsyncs.



В столбцах **fsyncs fs, fsync_time fst** у всех процессов кроме процесса checkpointer нули. У процесса checkpointer ненулевые значения обновятся и увеличатся только после окончания контрольной точки. Интервал между контрольными точками точки 5 минут. Если контрольная точка началась, то в течение ~4,5 минут в строке checkpointer будут увеличиваться цифры:

```
   backend_type     | context |   w    |   wt   |   wb   |   wbt   |  ex  | et  |  hits   | ev | ru | fs | fst
--------------------+---------+--------+--------+--------+---------+------+-----+---------+----+----+----+-----
 autovacuum worker | normal  |      0 |   0.00 |      0 |    0.00 |    0 |   0 |   67705 |  0 |    |  0 |   0
 autovacuum worker | vacuum  |      0 |   0.00 |      0 |    0.00 |    0 |   0 |   46055 |  0 |  0 |    |
 client backend    | normal  |      0 |   0.00 |      0 |    0.00 | 5553 | 120 | 4425701 |  0 |    |  0 |   0
 checkpointer      | normal  | 3376 | 63.99 | 3360 | 24.57 |      |     |         |    |    |  0 |   0
(4 rows)
```

Пример после окончания контрольной точки, fsyncs=30044 что равно числу файлов в табличных пространствах кластера, блоки которых обновлялись в течение контрольной точки. Это 30000 создаваемых скриптом таблиц и индексов и 44 relations системного каталога:

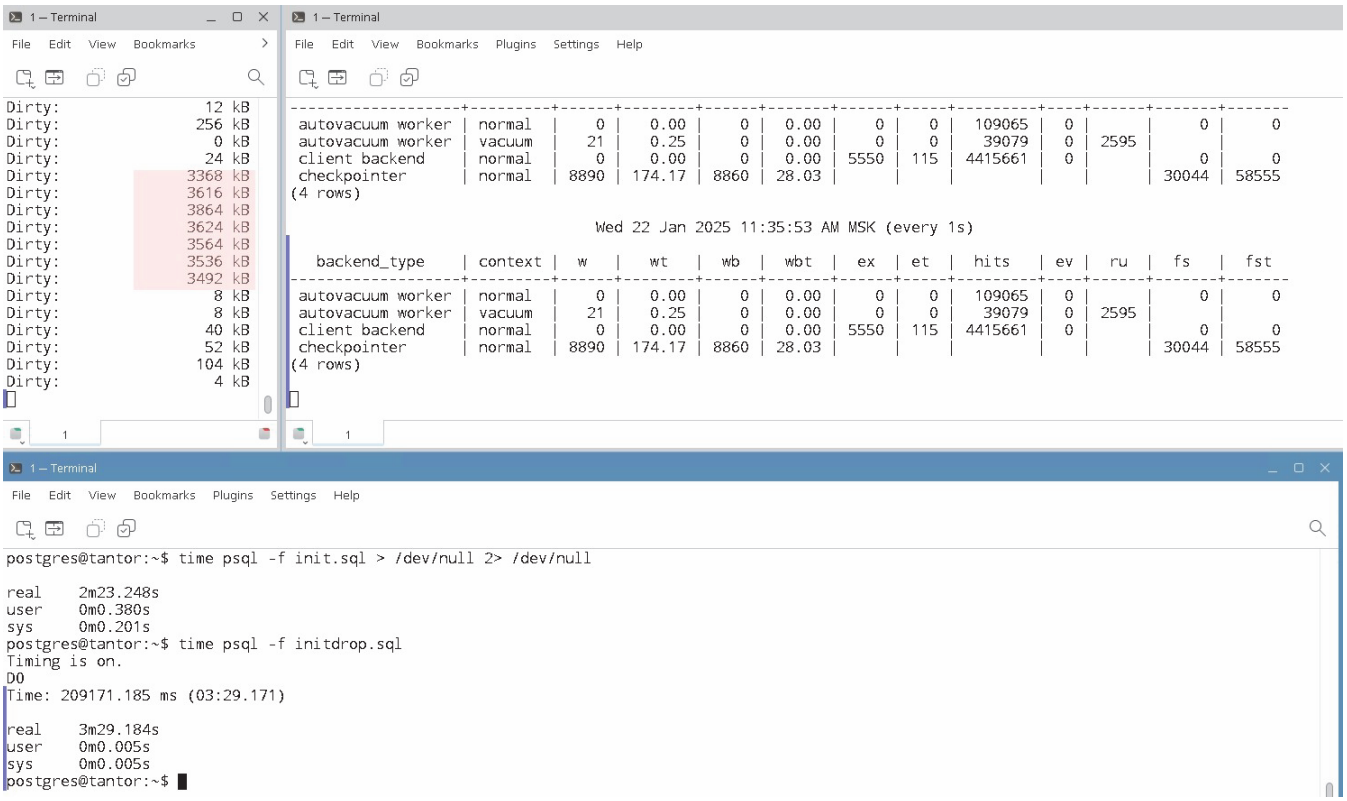


Какие бы тесты не запускались, в столбцах **fsyncs fs, fsync_time fst** у всех процессов, кроме процесса checkpointer, будут только нули. Нули означают нормальную работу экземпляра. Однако, под большой нагрузкой и неравномерным доступом к памяти (NUMA), есть вероятность что checkpointer не сможет обработать очередь блоков на синхронизацию (writeback) и в этом случае процессы, у которых в столбце **fsyncs fs** нули могут посылать системные вызовы writeback и fsync в операционную систему самостоятельно. Дальше посмотрим такие случаи и как это влияет на производительность.

8) Запустите скрипт удаления таблиц:

```
postgres@tantor:~$ time psql -f initdrop.sql > /dev/null 2> /dev/null
real    0m5.614s
user    0m0.009s
sys     0m0.000s
```

Через некоторое время после выполнения скрипта число грязных страниц в страничном кэше операционной системы увеличится:



Удержание страниц в кэше linux настраивается параметрами linux.

Если запустить скрипт удаления таблиц после остановки checkpointer, то команды удаления не смогут выполняться, при этом блокировки в pg_locks отражаться не будут. В примере на картинке скрипт initdrop.sql висел 3m29.184s до тех пор пока не была возобновлена работа процесса checkpointer. Для наблюдения за такими задержками можно использовать представление pg_stat_activity. В представлении будет событие ожидания **RegisterSyncRequest**:

```
postgres=# select * from pg_stat_activity where wait_event_type='Timeout'\gx
-[ RECORD 1 ]----+----------------------------------------------------------------
datid            | 5
datname          | postgres
pid              | 577742
leader_pid       |
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
```

```
client_port      | -1
backend_start    | 13:18:01.647753+03
xact_start       | 13:18:01.650781+03
query_start      | 13:18:01.650781+03
state_change     | 13:18:01.650785+03
wait_event_type  | Timeout
wait_event       | RegisterSyncRequest
state            | active
backend_xid      |
backend_xmin     |
query_id         |
query            | DO                                                              +
                 | $$                                                              +
                 | begin                                                           +
                 |  for i in 1..10000 by 100 loop                                  +
                 |   for j in 0..99 loop                                           +
                 |    execute concat('drop table if exists test.film_summary',i+j);+
                 |   end loop;                                                     +
                 |   commit;                                                       +
                 |  end loop;                                                      +
                 |  execute 'drop schema if exists test cascade';                  +
                 | end;                                                            +
                 | $$                                                              +
                 | LANGUAGE plpgsql;
backend_type     | client backend
```

Ожидание **RegisterSyncRequest** - это передача запросов синхронизации процессу контрольной точки из-за переполнения локальной очереди запросов. Событие не связано со способом удаления (не зависит от наличия кода plpgsql, execute, commit). Размер очереди не зависит от параметров конфигурации `*_flush_after`.

9) Остановите процесс checkpointer указав его PID:

```
postgres@tantor:~$ kill -STOP 421548
```

10) Проверьте, что процесс остановлен:

```
postgres@tantor:~$ ps -eo pid,s,comm | grep postgres | grep T
 421548 T postgres
```

11) Запустите скрипт создания 10000 таблиц:

```
postgres@tantor:~$ time psql -f init.sql > /dev/null 2> /dev/null
```

12) Через 1 минуту (послу увеличения extends на ~3000 блоков) в столбцах **fsyncs**, **fsync_time** у `client backend` (серверных процессов) появятся ненулевые значения и начнут расти:

```
   backend_type    | context | w |  wt  | wb | wbt  |  ex  | et |  hits   | ev |  ru  | fs | fst
-------------------+---------+---+------+----+------+------+----+---------+----+------+----+-----
 autovacuum worker | normal  | 0 | 0.00 |  0 | 0.00 |    0 |  0 |    8588 |  0 |      |  0 |   0
 autovacuum worker | vacuum  | 0 | 0.00 |  0 | 0.00 |    0 |  0 |    5679 |  0 | 2896 |    |
 client backend    | normal  | 0 | 0.00 |  0 | 0.00 | 3031 | 110 | 1260557 |  0 |      | 19 | 165
(3 rows)
```

Как только серверный процесс заполнил структуру `Checkpointer Data` идентификаторами блоков в разделяемой памяти, так что в ней не осталось места, серверному процессу пришлось писать в локальную память. Размер структуры 512Кб:

```
postgres=# SELECT name, allocated_size, pg_size_pretty(allocated_size) FROM
pg_shmem_allocations where name like '%Check%' ORDER BY size DESC;
          name          | allocated_size | pg_size_pretty
------------------------+----------------+----------------
 Checkpointer Data      |         524416 | 512 kB
 Checkpoint BufferIds   |         327680 | 320 kB
(2 rows)
```

В `Checkpointer Data` пишут все процессы. Из структуры читает процесс checkpointer и перемещает идентификаторы блоков в хэш-таблицу в своей локальной памяти, которая называется "`Pending Ops Table`" или "`pending sync hash`".

13) Через `5m32` скрипт доработает:

```
real    5m32.496s
user    0m0.387s
sys     0m0.180s
```

Длительность работы скрипта при fsyncs выдаваемых серверным процессом, а не процессом контрольной точки увеличилась с `2m23` до `5m32`, в 3 раза. Если процесс checkpointer не справляется и другие процессы не могут записать в память checkpointer указатели на свои блоки, то эффективность работы снижается.

За время работы число fsyncs инициированных серверным процессом `16138`:

```
   backend_type    | context |  w   |  wt   |  wb  |  wbt  |  ex   |  et   |  hits   | ev | ru |  fs   |  fst
-------------------+---------+------+-------+------+-------+-------+-------+---------+----+----+-------+--------
 autovacuum worker | normal  |    0 |  0.00 |    0 |  0.00 |     0 |     0 |  168119 |  0 |    |     0 |      0
 autovacuum worker | vacuum  |    0 |  0.00 |    0 |  0.00 |     0 |     0 |   77867 |  0 |  0 |       |
 client backend    | normal  |    0 |  0.00 |    0 |  0.00 | 11104 | 29495 | 6736834 |  0 |    | 16138 | 164095
 checkpointer      | normal  | 4286 | 81.14 | 4256 | 27.17 |       |       |         |    |    |     0 |      0
(4 rows)
```

11) Возобновите работу процесса checkpointer:

```
postgres@tantor:~$ kill -CONT 421548
postgres@tantor:~$ ps -eo pid,s,comm | grep postgres | grep T
```

12) Выполните контрольную точку:

```
postgres@tantor:~$ time psql -c "checkpoint;"
CHECKPOINT

real    0m1.043s
user    0m0.009s
sys     0m0.000s
```

```
   backend_type    | context |   w   |   wt   |  wb   |   wbt   |  ex   |  et   |  hits   | ev | ru |  fs   |  fst
-------------------+---------+-------+--------+-------+---------+-------+-------+---------+----+----+-------+--------
 autovacuum worker | normal  |     0 |   0.00 |     0 |    0.00 |     0 |     0 |  180324 |  0 |    |     0 |      0
 autovacuum worker | vacuum  |     0 |   0.00 |     0 |    0.00 |     0 |     0 |   77867 |  0 |  0 |       |
 client backend    | normal  |     0 |   0.00 |     0 |    0.00 | 11104 | 29495 | 6736892 |  0 |    | 16138 | 164095
 checkpointer      | normal  | 13339 | 169.75 | 13339 | 201.58 |       |       |         |    |    | 46456 |  90487
(4 rows)
```

13) Посмотрите как контрольные точки отражены в журнале:

```
postgres@tantor:~$ cat $PGDATA/log/postgresql-*
   22:26:46.992 MSK [427531] LOG:  starting Tantor Special Edition 16.2.0 e12e484f on x86_64-pc-linux-gnu,
compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
   22:26:46.992 MSK [427531] LOG:  listening on IPv4 address "127.0.0.1", port 5432
   22:26:46.993 MSK [427531] LOG:  listening on IPv6 address "::1", port 5432
   22:26:47.002 MSK [427531] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
   22:26:47.031 MSK [427536] LOG:  database system was shut down at 22:26:46 MSK
   22:26:47.052 MSK [427531] LOG:  database system is ready to accept connections
   22:33:20.721 MSK [427534] LOG:  checkpoint starting: time
   22:44:06.515 MSK [427534] LOG:  checkpoint complete: wrote 5692 buffers (34.7%); 0 WAL file(s) added, 0
removed, 14 recycled; write=529.876 s, sync=115.114 s, total=645.794 s; sync files=46417, longest=0.041 s,
average=0.003 s; distance=220844 kB, estimate=220844 kB; lsn=3/5B929D50, redo lsn=3/4E4F28C0
   22:48:01.468 MSK [427534] LOG:  checkpoint starting: immediate force wait
   22:48:02.770 MSK [427534] LOG:  checkpoint complete: wrote 3068 buffers (18.7%); 0 WAL file(s) added, 0
removed, 15 recycled; write=0.131 s, sync=0.112 s, total=1.302 s; sync files=41, longest=0.027 s, average=0.003
s; distance=253551 kB, estimate=253551 kB; lsn=3/5DC8E8A8, redo lsn=3/5DC8E860
   22:53:01.203 MSK [427534] LOG:  checkpoint starting: time
```

```
   23:28:28.751 MSK [427534] LOG:  checkpoint complete: wrote 5922 buffers (36.1%); 0 WAL file(s) added, 0
removed, 8 recycled; write=2035.711 s, sync=91.719 s, total=2127.548 s; sync files=46423, longest=0.028 s,
average=0.002 s; distance=123698 kB, estimate=240566 kB; lsn=3/72C6A338, redo lsn=3/6555B2F0
   23:28:48.533 MSK [427534] LOG:  checkpoint starting: immediate force wait
   23:28:49.561 MSK [427534] LOG:  checkpoint complete: wrote 7430 buffers (45.3%); 0 WAL file(s) added, 0
removed, 13 recycled; write=0.285 s, sync=0.087 s, total=1.028 s; sync files=35, longest=0.018 s, average=0.003
s; distance=220220 kB, estimate=238531 kB; lsn=3/72C6A460, redo lsn=3/72C6A418
```

Часть 3. Тестирование производительности ввода-вывода

В предыдущем тесте создавались таблицы, но грязнилось небольшое число блоков, которые все поместились в буферный кэш, поэтому активность автовакуума и bgwriter не наблюдалась.

Посмотрим какая нагрузка на ввод-вывод будет, если в буферном кэше все буфера будут заняты.

1) 3) Запустите во втором терминале psql и команды:

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)
```

2) Запустите скрипт вставки 300 строк в каждую из 10000 таблиц:

```
postgres@tantor:~$ cat init2.sql
select format('insert into test.film_summary%1$s select i, %2$s || i, i from
generate_series(0, 153) as i;', g.id, E'\'text number \'') from
generate_series(1, 10000) as g(id) \gexec
postgres@tantor:~$ time psql -f init2.sql > /dev/null 2> /dev/null

real    0m44.707s
user    0m0.351s
sys     0m0.149s
```

Вставка выполнилась за 44 секунды.

3) Остановите процесс checkpointer указав его PID:

```
postgres@tantor:~$ kill -STOP 421548
postgres@tantor:~$ ps -eo pid,s,comm | grep postgres | grep T
 421548 T postgres
```

4) Запустите скрипт вставки 300 строк в каждую из 10000 таблиц:

```
postgres@tantor:~$ time psql -f init2.sql > /dev/null 2> /dev/null

real    4m22.384s
user    0m0.400s
sys     0m0.220s
```

Вставка выполнится вместо 44 секунд за 4m22.

5) Пока скрипт работает наблюдайте в окне с циклом как меняется число грязных страниц в кэше linux:

```
Dirty:              4396 kB
Dirty:              4436 kB
Dirty:               280 kB
Dirty:                88 kB
Dirty:               220 kB
Dirty:               668 kB
Dirty:               672 kB
Dirty:               888 kB
Dirty:               788 kB
Dirty:               844 kB
Dirty:              1428 kB
Dirty:              1500 kB
Dirty:              1580 kB
Dirty:              1200 kB
```

```
   backend_type    | context |   w   |    wt     |  wb   |   wbt   |  ex   |   et   |  hits   |  ev   | ru |  fs   |  fst
-------------------+---------+-------+-----------+-------+---------+-------+--------+---------+-------+----+-------+--------
 autovacuum worker | normal  |     5 |    123.04 |     0 |    0.00 |     0 |      0 |   26649 |   433 |    |     5 |    113
 client backend    | normal  |  4723 |   1826.44 |     0 |    0.00 | 50000 | 126423 | 3347825 | 58748 |    | 10995 | 170635
 background writer | normal  | 33122 | 221750.88 | 33088 | 1959.54 |       |        |         |       |    | 13326 | 195512
 checkpointer      | normal  |  3535 |     66.54 |  3520 |    9.97 |       |        |         |       |    |     0 |      0
(4 rows)
```

6) Остановите процесс bgwriter указав его PID:

```
postgres@tantor:~$ kill -STOP 421549
postgres@tantor:~$ ps -eo pid,s,comm | grep postgres | grep T
 421548 T postgres
 421549 T postgres
```

7) Запустите во втором терминале psql и команды:

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)
```

8) Запустите во втором терминале psql и команды:

```
postgres=#
select backend_type, context, writes w, round(write_time::numeric,2) wt, writebacks wb, round(writeback_time::numeric,2) wbt, extends ex, round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs, round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or evictions>0 or reuses>0 or hits>0;
\watch 1
```

9) Запустите скрипт вставки 153 строки (один блок) в каждую из 10000 таблиц:

```
postgres@tantor:~$ cat init2.sql
select format('insert into test.film_summary%1$s select i, %2$s || i, i from generate_series(0, 153) as i;', g.id, E'\'text number \'') from generate_series(1, 10000) as g(id) \gexec
```

```
postgres@tantor:~$ time psql -f init2.sql > /dev/null 2> /dev/null
```

10) При остановленном bgwriter время выполнения скрипта увеличилось:

```
real    6m57.672s
user    0m0.426s
sys     0m0.165s
```

11) Возобновите работу процесса checkpointer и bgwriter:

```
postgres@tantor:~$ kill -CONT 421549
```

12) Посмотрите как будут меняться статистики:

```
   backend_type    | context |   w   |     wt      | wb  | wbt  |  ex   |   et   |  hits   |  ev   | ru |  fs   |  fst
-------------------+---------+-------+-------------+-----+------+-------+--------+---------+-------+----+-------+--------
 autovacuum worker | normal  |     0 |        0.00 |   0 | 0.00 |     0 |      0 |   29544 |     0 |    |     0 |      0
 client backend    | normal  | 25551 | 263639.10 |   0 | 0.00 | 10000 | 102812 | 1675770 | 44402 |    | 35551 | 320925
 background writer | normal  |   200 |     1875.13 | 192 | 0.33 |       |        |         |       |    |   200 |   1505
(3 rows)

                                Wed 22 Jan 2025 12:14:11 AM MSK (every 1s)

   backend_type    | context |   w   |     wt      | wb  | wbt  |  ex   |   et   |  hits   |  ev   | ru |  fs   |  fst
-------------------+---------+-------+-------------+-----+------+-------+--------+---------+-------+----+-------+--------
 autovacuum worker | normal  |     0 |        0.00 |   0 | 0.00 |     0 |      0 |   29544 |     0 |    |     0 |      0
 client backend    | normal  | 25551 | 263639.10 |   0 | 0.00 | 10000 | 102812 | 1675770 | 44402 |    | 35551 | 320925
 background writer | normal  |   300 |     2806.14 | 320 | 0.53 |       |        |         |       |    |   300 |   2252
(3 rows)

                                Wed 22 Jan 2025 12:14:12 AM MSK (every 1s)

   backend_type    | context |   w   |     wt      | wb  | wbt  |  ex   |   et   |  hits   |  ev   | ru |  fs   |  fst
-------------------+---------+-------+-------------+-----+------+-------+--------+---------+-------+----+-------+--------
 autovacuum worker | normal  |     0 |        0.00 |   0 | 0.00 |     0 |      0 |   29544 |     0 |    |     0 |      0
 client backend    | normal  | 25551 | 263639.10 |   0 | 0.00 | 10000 | 102812 | 1675770 | 44402 |    | 35551 | 320925
 background writer | normal  |   400 |     3723.27 | 384 | 0.63 |       |        |         |       |    |   400 |   2986
(3 rows)
```

Раз в секунду bgwriter записывает **bgwriter_lru_maxpages=100** блоков и так как в каждой таблице файл основного слоя состоит из одного блока, то даёт по каждому файлу один fsync. Если бы в таблицу вставлялись не 153 строки, а больше (так чтобы было больше блоков в каждом файле), то число fsyncs в секунду было бы меньше, чем writes.

13) Возобновите работу процесса checkpointer:

```
postgres@tantor:~$ kill -CONT 421548
postgres@tantor:~$ ps -eo pid,s,comm | grep postgres | grep T
```

14) Запустите скрипт, который выполняет vacuum full всех таблиц:

```
postgres@tantor:~$ cat init4.sql
select format('vacuum (freeze, analyze) test.film_summary%s;', g.id) from
generate_series(1, 10000) as g(id) \gexec
postgres@tantor:~$ time psql -f init4.sql > /dev/null 2> /dev/null
```

Пока работает вакуум число грязных страниц в кэше linux увеличено:

```
Dirty:             48688 kB
Dirty:             45060 kB
Dirty:             46588 kB
Dirty:             48048 kB
Dirty:             48860 kB
Dirty:             50196 kB
Dirty:             46844 kB
Dirty:             48644 kB
```

Появится новая строка с буферным кольцом типа **vacuum**:

```
   backend_type    | context |   w   |     wt      | wb  |  wbt  |  ex   |   et   |  hits   |  ev   | ru |  fs   |  fst
```

```
------------------+---------+-------+-----------+-------+----------+-------+--------+----------+--------+-----+-------+--------
 autovacuum worker | normal  |    11 |      0.26 |     0 |     0.00 |     0 |      0 |    48157 |     34 |     |     0 |      0
 autovacuum worker | vacuum  |    24 |      0.40 |     0 |     0.00 |     0 |      0 |     1366 |     27 |   0 |       |
 client backend    | normal  | 31060 | 263736.52 |     0 |     0.00 | 20864 | 103049 | 8178640  | 87519  |     | 35551 | 320925
 client backend    | vacuum  |  2670 |     45.12 |     0 |     0.00 |     0 |      0 |    21612 |  21615 |   0 |       |
 background writer | normal  | 44522 |  25762.81 | 44544 |  5029.43 |       |        |          |        |     |  2685 |  19948
 checkpointer      | normal  |  1534 |     16.76 |  1541 |     9.91 |       |        |          |        |     | 36425 |  83780
(6 rows)
```

Число fsync нигде не увеличивается, так как работает checkpointer. Он выполнит fsyncs в конце своего цикла и число увеличится с 36425 до **65799** по числу файлов.

```
   backend_type    | context |   w   |    wt     |  wb   |   wbt   | ex  |  et   |   hits   |   ev   | ru |  fs   |  fst
-------------------+---------+-------+-----------+-------+---------+-----+-------+----------+--------+----+-------+--------
 autovacuum worker | normal  |    18 |      0.38 |     0 |    0.00 |   0 |     0 |    74617 |    125 |    |     0 |      0
 autovacuum worker | vacuum  |   600 |      5.79 |     0 |    0.00 |   0 |     0 |     4433 |     86 | 561 |       |
 client backend    | normal  | 33246 | 263777.41 |     0 |    0.00 | 25081 | 103136 | 10702396 | 104216 |    | 35551 | 320925
 client backend    | vacuum  |  3743 |     62.30 |     0 |    0.00 |   0 |     0 |    30000 |  30000 |  0 |       |
 background writer | normal  | 66713 |  26172.87 | 66688 | 7645.14 |     |       |          |        |    |  2685 |  19948
 checkpointer      | normal  |  1534 |     16.76 |  1552 |    9.92 |     |       |          |        |    | 65799 | 161190
(6 rows)
```

Если число не увеличилось, то чтобы не ждать до 5 минут контрольной точки по времени можно выполнить контрольную точку вручную:

```
postgres@tantor:~$ time psql -c "checkpoint;"
CHECKPOINT
```

15) Этими тестами можно оценивать как влияют размеры буферов на время выполнения тестов:

```
alter system set backend_flush_after = 1;
alter system set bgwriter_flush_after = 1;
alter system set checkpoint_flush_after = 1;
select pg_reload_conf();
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');

postgres@tantor:~$ time psql -f init2.sql > /dev/null 2> /dev/null

real    1m2.733s
user    0m0.403s
sys     0m0.163s
```

Время увеличилось с 0m44 до 1m2.

```
postgres@tantor:~$ time psql -f initdrop.sql > /dev/null 2> /dev/null

real    0m9.544s
user    0m0.010s
sys     0m0.000s
```

Время удаления таблиц тоже увеличилось с 0m5.547s до 0m9.544s.

16) Если меняли значения параметров, то верните значения параметров к значениям по умолчанию:

```
postgres=# alter system reset backend_flush_after;
alter system reset bgwriter_flush_after;
alter system reset checkpoint_flush_after;
select pg_reload_conf();
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
 pg_reload_conf
```

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

 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)
```

Часть 4. Выбор размера temp_buffers при работе с временными таблицами с помощью pg_stat_io

1) В первом терминале запустите:

```
astra@tantor:~$ while true; do cat /proc/meminfo | grep Dirty; sleep 3; done
```

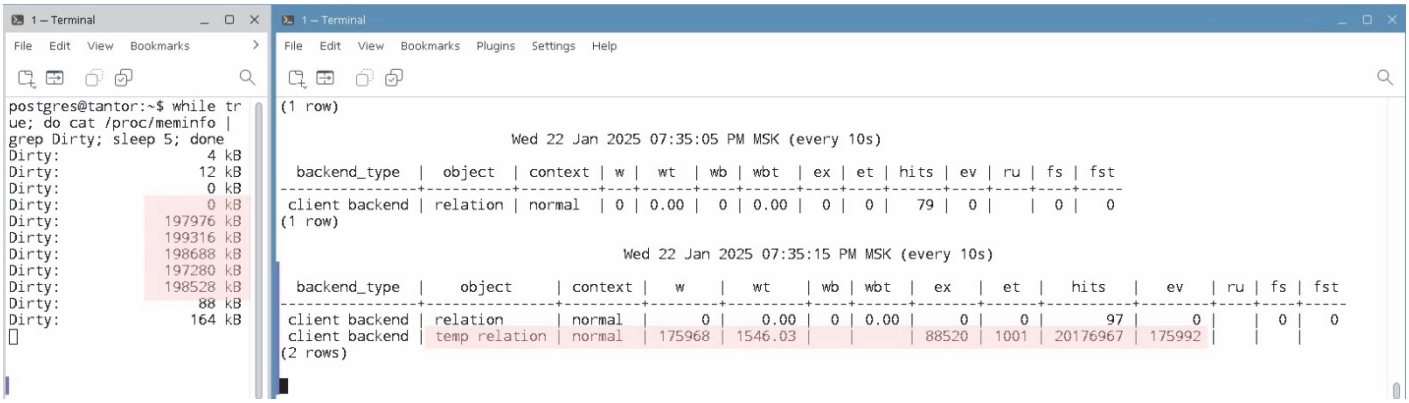
2) Во втором терминале в psql запустите команды:

```
postgres=#
select backend_type, context, writes w, round(write_time::numeric,2) wt,
writebacks wb, round(writeback_time::numeric,2) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
\watch 10
```

3) В третьем терминале в psql запустите команды:

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
create temp table temp1 (id integer);
\timing on \\
insert into temp1 select * from generate_series(1, 20000000);
explain (analyze, timing off, buffers) select * from temp1;
drop table temp1;
\timing off \\

 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)

CREATE TABLE
Timing is on.
INSERT 0 20000000
Time: 19265.840 ms (00:19.266)
                                          QUERY PLAN
-----------------------------------------------------------------------------------------------
 Seq Scan on temp1  (cost=0.00..314160.80 rows=22566480 width=4) (actual rows=20000000 loops=1)
   Buffers: local read=88496 dirtied=88496 written=88494
   I/O Timings: local read=663.960 write=794.682
 Planning:
   Buffers: shared hit=4
 Planning Time: 0.048 ms
 Execution Time: 5923.242 ms
(7 rows)
```

```
Time: 5923.524 ms (00:05.924)
Timing is off.
DROP TABLE
Time: 191.131 ms
```



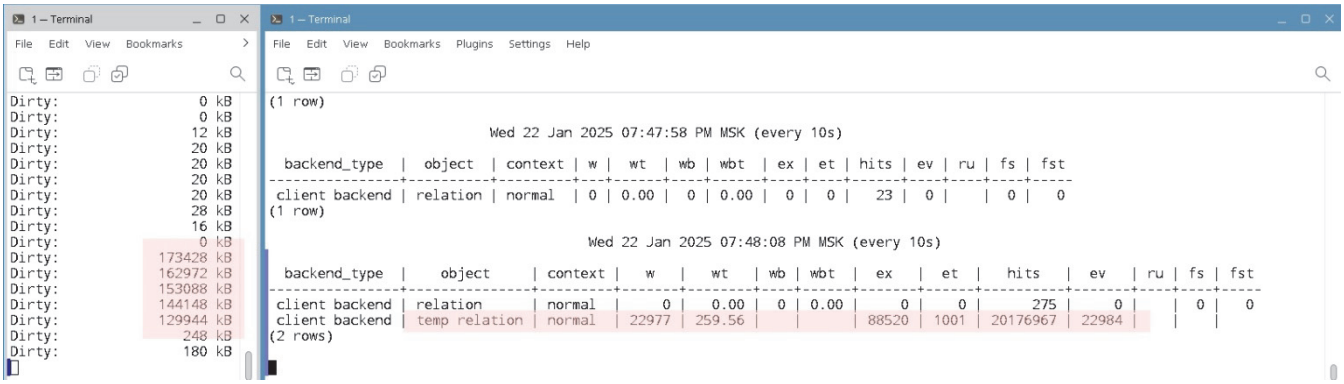
4) В третьем терминале перезапустите psql и поменяйте размер локального кэша:

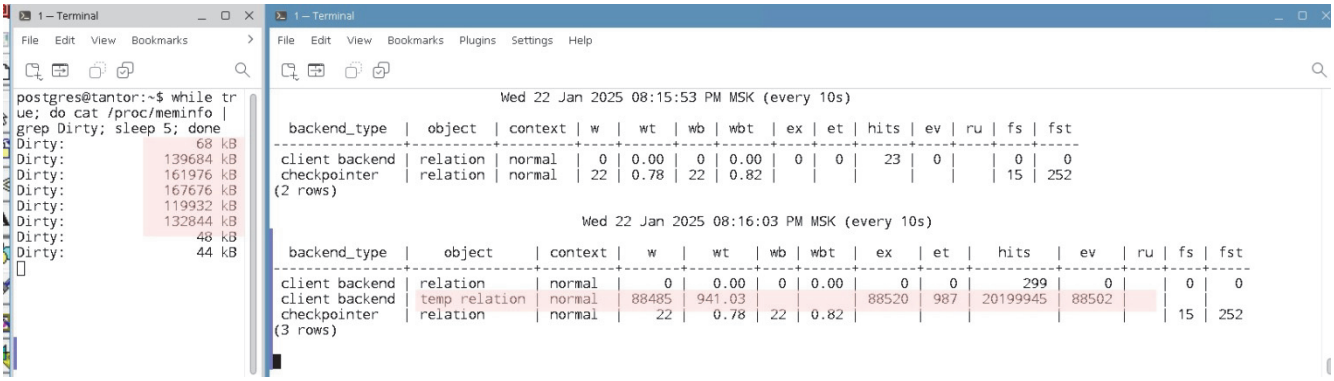
```
postgres=# \q
postgres@tantor:~$ psql
Type "help" for help.

postgres=#
show temp_buffers;
set temp_buffers='512MB';
 temp_buffers
--------------
 8MB
(1 row)
SET
```

5) В третьем терминале в psql повторите команды теста:

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
create temp table temp1 (id integer);
\timing on \\
insert into temp1 select * from generate_series(1, 20000000);
explain (analyze, timing off, buffers) select * from temp1;
select pg_size_pretty(pg_table_size('temp1'));
drop table temp1;
\timing off \\
 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)

CREATE TABLE
Timing is on.
INSERT 0 20000000
Time: 19858.309 ms (00:19.858)
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on temp1  (cost=0.00..314160.80 rows=22566480 width=4) (actual rows=20000000 loops=1)
   Buffers: local hit=22978 read=65518 dirtied=65518 written=65508
   I/O Timings: local read=581.173 write=693.463
 Planning:
   Buffers: shared hit=22
 Planning Time: 0.126 ms
```

```
 Execution Time: 5523.708 ms
(7 rows)

Time: 5524.133 ms (00:05.524)
  pg_size_pretty
----------------
         692 MB
(1 row)
DROP TABLE
Time: 90.952 ms
Timing is off.
```





Временные объекты тспользуют локальный кэш серверного процесса.

Размер таблицы **692 MB**. При размере локального пула буферов **8MB** было записано **writes=175968** блоков = 1374Мб. Это равно двойному размеру таблицы минус **8MB**.

Буфера являются отражением содержимого файла и файл создаётся сразу при создании временной таблицы. **Значение в поле `extends = 88520` = 691Мб соответствует размеру таблицы и одинаково в обоих тестах.**

При размере локального пула буферов **512MB** после команды **`insert`** было **`writes`** (передано в кэш linux и позже записано на диск, что можно было наблюдать в первом терминале **`cat /proc/meminfo | grep Dirty`** - число грязных страниц уменьшилось, что означает физическую запись на диск) 22977=179Мб. **512MB**+179Мб=691Мб, что соответствует размеру таблицы. После **`explain analyze`** суммарно было **`writes`** 88485 блоков = 691Мб.

Время выполнения команд изменилось не сильно. Вставка строк с размером буфера 8Мб была немного быстрее, чем с размером буфера 512Мб. В тесте массово вставлялись строки и один раз массово читались. В таком режиме размер кэша не играет роли, это режим работы со строками, аналогичный буферному кольцу и он эффективен. Если к строкам временной таблицы будет многократный произвольный доступ, размер кэша будет иметь значение. Поскольку с временной таблицей работает только одна сессия, то "многократность" меньше в сравнении с обычными таблицами, доступным сотням сессий. Типичны режим работы с временными таблицами: массово вставляются данные, массово обновляются и результат читается. В таком режиме даже индексы не нужны. Для такого режима размер кэша не имеет значения. Однако, приложения могут использовать временные таблицы и в других режимах, поэтому при выборе размера **`temp_buffers`** стоит использовать команды, приближенные к тем, которые использует приложение.

По файлам временных объектов **fsyncs** и **writebacks** не выполняются, на что указывают пустые значения в строке с **object='temp relation'**.

6) Посмотрите какие будут показатели работы с нежуралируемой таблицей:

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
create unlogged table temp1 (id integer);
\timing on \\
insert into temp1 select * from generate_series(1, 20000000);
explain (analyze, timing off, buffers) select * from temp1;
select pg_size_pretty(pg_table_size('temp1'));
drop table temp1;
\timing off \\
 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)
CREATE TABLE
Timing is on.
INSERT 0 20000000
Time: 20868.504 ms (00:20.869)
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on temp1  (cost=0.00..314160.80 rows=22566480 width=4) (actual rows=20000000 loops=1)
   Buffers: shared hit=16344 read=72152 dirtied=74086 written=72120
   I/O Timings: shared read=1011.583 write=633.551
 Planning:
   Buffers: shared hit=1 read=3
   I/O Timings: shared read=208.287
 Planning Time: 208.366 ms
 Execution Time: 5935.770 ms
(8 rows)

Time: 6144.520 ms (00:06.145)
DROP TABLE
Time: 260.237 ms
Timing is off.
```

Время выполнения команд схоже с времени выполнения при работе с временной таблицей и буфером 8MB. При выборке из таблицы использовалось буферное кольцо:

```
 backend_type     | object |context |  w   |  wt   |  wb  |   wbt    | ex  | et |  hits    | ev    | ru   |fs |fst
------------------+--------+--------+------+-------+------+----------+-----+----+----------+-------+------+---+---
autovacuum worker|relation|normal  |    0|  0.00|    0 |     0.00 |    0|   0|   13862 |     0 |      | 0 |  0
client backend   |relation|bulkread|72120|633.55|    0 |     0.00 |     |    |   16344 |    32 |72120|   |
client backend   |relation|normal  |66840|684.00|    0 |     0.00 |88520|1396|20177108 |73036 |      | 0 |  0
background writer|relation|normal  | 7291|118.17| 7296 | 4993.65 |     |    |         |      |      | 0 |  0
(4 rows)
```

7) Почему не использовалось буферное кольцо на запись? Перепишите команды, чтобы использовалось кольцо массовой записи:

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
\timing on \\
create unlogged table temp1 as select id from generate_series(1, 20000000)
x(id);
explain (analyze, timing off, buffers) select * from temp1;
select pg_size_pretty(pg_table_size('temp1'));
```

```
drop table temp1;
\timing off \\
 ----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)

Timing is on.
SELECT 20000000
Time: 16304.857 ms (00:16.305)
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Seq Scan on temp1  (cost=0.00..314217.60 rows=22570560 width=4) (actual rows=20000000 loops=1)
   Buffers: shared hit=2048 read=86464 dirtied=86464 written=86432
   I/O Timings: shared read=635.782 write=752.176
 Planning:
   Buffers: shared hit=3 read=1
   I/O Timings: shared read=0.010
 Planning Time: 0.097 ms
 Execution Time: 5793.216 ms
(8 rows)

Time: 5793.607 ms (00:05.794)
 pg_size_pretty
----------------
 692 MB
(1 row)

Time: 14.803 ms
DROP TABLE
Time: 127.085 ms
Timing is off.
```

Команды стали выполняться **быстрее**.

```
  backend_type   | object | context |  w   |  wt   |wb|wbt | ex    |et  |hits  |ev | ru    |fs |fst
-----------------+--------+---------+------+-------+--+----+-------+----+------+---+-------+---+---
autovacuum worker|relation|normal   |    0|  0.00| 0|0.00|     0 |  0 | 2447 | 0 |       | 0 |  0
client backend   |relation|bulkread |86432|752.18| 0|0.00|       |    | 2048 | 0 |86432 |   |
client backend   |relation|bulkwrite|86464|885.37| 0|0.00|88512 |508 |87107 | 0 |86464 |   |
client backend   |relation|normal   |    0|  0.00| 0|0.00|     0 |  0 |  100 | 0 |       | 0 |  0
(4 rows)
```

Часть 5. Пример анализа статистик работы команды vacuum с кольцом

1) Создайте таблицу для теста:

```
postgres=# drop table if exists test;
create table test(a int, b int) with (autovacuum_enabled = 'false');
insert into test select i, i from generate_series(1, 4500) i;
SELECT n.nspname, c.relname, count(*) AS buffers, count(b.isdirty) FILTER
(WHERE b.isdirty = true) as buffer_dirty from pg_buffercache b JOIN pg_class c ON
b.relfilenode = pg_relation_filenode(c.oid) AND b.reldatabase IN (0, (SELECT oid
FROM pg_database WHERE datname = current_database())) JOIN pg_namespace n ON
n.oid = c.relnamespace GROUP BY n.nspname, c.relname having relname='test';
NOTICE:  table "test" does not exist, skipping
DROP TABLE
CREATE TABLE
INSERT 0 4500
 nspname | relname | buffers | buffer_dirty
---------+---------+---------+--------------
 public  | test    |      23 |           20
(1 row)
```

В кэше буферов по файлам таблицы 23 блока, 20 из них грязные.

2) Чтобы убрать блоки файлов таблицы из кэша буферов выполните команду:

```
postgres=# vacuum full test;
SELECT n.nspname, c.relname, count(*) AS buffers, count(b.isdirty) FILTER
(WHERE b.isdirty = true) as buffer_dirty from pg_buffercache b JOIN pg_class c ON
b.relfilenode = pg_relation_filenode(c.oid) AND b.reldatabase IN (0, (SELECT oid
FROM pg_database WHERE datname = current_database())) JOIN pg_namespace n ON
n.oid = c.relnamespace GROUP BY n.nspname, c.relname having relname='test';
VACUUM
 nspname | relname | buffers | buffer_dirty
---------+---------+---------+--------------
(0 rows)
```

3) Таблица занимает 20 блоков:

```
postgres=# select pg_table_size('test')/8192 blocks;
select pg_relation_size('test')/8192 blocks;
 blocks
--------
     20
(1 row)
 blocks
--------
     20
(1 row)
```

В таблице один файл основного слоя, файлов vm и fsm нет.

4) Обнулите статистику, вакуумируйте таблицу, посмотрите статистику:

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
vacuum (BUFFER_USAGE_LIMIT 128) test;
select backend_type, context, writes w, round(write_time::numeric,2) wt,
writebacks wb, round(writeback_time::numeric,2) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
SELECT n.nspname, c.relname, count(*) AS buffers, count(b.isdirty) FILTER
(WHERE b.isdirty = true) as buffer_dirty from pg_buffercache b JOIN pg_class c ON
b.relfilenode = pg_relation_filenode(c.oid) AND b.reldatabase IN (0, (SELECT oid
FROM pg_database WHERE datname = current_database())) JOIN pg_namespace n ON
n.oid = c.relnamespace GROUP BY n.nspname, c.relname having relname='test';
 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)

VACUUM
 backend_type  | context | w |  wt  | wb | wbt  | ex | et | hits | ev | ru | fs | fst
```

```
---------------+---------+---+------+----+------+----+----+------+----+----+----+----
client backend | normal  | 0 | 0.00 |  0 | 0.00 |  4 |  0 |  828 |  0 |    |  0 |    0
client backend | vacuum  | 4 | 0.03 |  0 | 0.00 |  0 |  0 |    0 |  0 |  4 |    |
(2 rows)
 nspname | relname | buffers | buffer_dirty
---------+---------+---------+--------------
 public  | test    |      20 |           20
(1 row)
```

При использовании параметра `BUFFER_USAGE_LIMIT` **буферное кольцо используется даже с маленькими таблицами (меньше 1/4 размера кэша буферов)**. Без параметра `BUFFER_USAGE_LIMIT` буферное кольцобы не использовалось, так как таблица маленькая. Если `BUFFER_USAGE_LIMIT=0`, то буферное кольцо не будет использоваться с таблицами любого размера.

`extends=4` означает, что к каким-то файлам командой **`vacuum`** было добавлено 4 блока, а возможно даже файлов не было и они были созданы.

`context=vacuum` означает, что команда **`vacuum`** использовала буферное кольцо.

`reuses=4` означает, что к кольцу было добавлено 4 блока.

`writes=4` означает, что 4 блока были посланы на запись.

`hits=828` означает, что команда **`vacuum`** обратилась 828 к буферам в буферном кэше. Это блоки объектов системного каталога. Возможно, к части буферов были неоднократные обращения.

В буферный кэш было считано 20 блоков таблицы и они были изменены. Подгрузка блоков в буферный кэш ("hits" есть, "miss" отсутствует), а также грязнение блоков ("reads" есть, "dirtiness" отсутствует) в представлении **`pg_stat_io`** не отражается.

При тестировании и анализе результатов тестов это стоит принимать во внимание.

5) Почему 4 блока, ведь в таблице 20 блоков? Посмотрите сколько блоков во всех файлах таблицы:

```
postgres=#
select pg_table_size('test')/8192 blocks;
select pg_relation_size('test')/8192 blocks;
select pg_relation_size('test','main')/8192 blocks;
select pg_relation_size('test','vm')/8192 blocks;
select pg_relation_size('test','fsm')/8192 blocks;
 blocks
--------
     24
(1 row)

 blocks
--------
     20
(1 row)

 blocks
--------
     20
(1 row)

 blocks
--------
      1
(1 row)

 blocks
--------
```

```
       3
(1 row)
```

После команды **vacuum** размер файлов таблицы стал 24 блока. 4 блока относятся к файлам слоёв vm и fsm. Команда **vacuum** создала эти файлы.

В чем отличия evictions от reused при работе с кольцами?

Если же блок находится в буфере, относящимся к кольцу и идёт замена блока на другой блок в буфере того же кольца, то замена блока учитывается в статистике reused этого типа (context) кольца. В данном случае в кольцо были добавлены незанятые блоки.

Если бы блок таблицы был закреплен (pinned) другим серверным процессом, то при прицеплении буфера в кольцо увеличилась бы статистика evictions, а не reused. Также замена блока в буфере кольца на другой блок той же таблицы учитывалась бы как evictions в строке, относящемся к кольцу (context=**vacuum**).

6) Удалите таблицу:

```
postgres=# drop table test;
DROP TABLE
```

7) Выполните команды:

```
postgres=# checkpoint;
select backend_type, context, writes w, round(write_time::numeric,2) wt,
writebacks wb, round(writeback_time::numeric,2) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
select * from pg_stat_bgwriter\gx
```

```
CHECKPOINT
   backend_type    | context | w  |  wt  | wb | wbt  | ex | et | hits | ev | ru | fs | fst
-------------------+---------+----+------+----+------+----+----+------+----+----+----+-----
 autovacuum worker | normal  |  0 | 0.00 |  0 | 0.00 |  0 |  0 | 4906 |  0 |    |  0 |   0
 client backend    | normal  |  0 | 0.00 |  0 | 0.00 |  4 |  0 |  919 |  0 |    |  0 |   0
 client backend    | vacuum  |  4 | 0.05 |  0 | 0.00 |  0 |  0 |    0 |  0 |  4 |    |
 checkpointer      | normal  | 39 | 0.44 | 39 | 1.12 |    |    |      |    |    | 17 |  41
(4 rows)

-[ RECORD 1 ]---------+---
checkpoints_timed     | 0
checkpoints_req       | 1
checkpoint_write_time | 9
checkpoint_sync_time  | 45
buffers_checkpoint    | 39
buffers_clean         | 0
maxwritten_clean      | 0
buffers_backend       | 40
buffers_backend_fsync | 0
buffers_alloc         | 20
```

Значения статистик в **pg_stat_bgwriter** отличаются от значений в **pg_stat_io**.

Часть 6. Работа bgwriter и сопоставление статистик в представлениях `pg_stat_bgwriter` и `pg_stat_io`

1) Создайте файлы для более сложного теста, приближенного к реальным нагрузкам:

```
postgres@tantor:~$ mcedit initcre.sql

DO
$$
begin
 for i in 1..10000 by 100 loop
  for j in 0..99 loop
   execute concat('drop table if exists test.film_summary',i+j);
  end loop;
  commit;
 end loop;
end;
$$
LANGUAGE plpgsql;
drop schema if exists test cascade;
create schema test;
select format('create table test.film_summary%s (film_id int, title int,
release_year int) with (autovacuum_enabled=off);', g.id) from generate_series(1,
10) as g(id)
\gexec
```

скрипт удаляет 100000 таблиц от предыдущих тестов, использовавшихся в практиках, если они есть и создает 10 таблиц.

2) Создайте файл скрипта **initdeleven.sql**:

```
postgres@tantor:~$ mcedit initdeleven.sql

select format('delete from test.film_summary%s where
int8mod((ctid::text::point)[0]::bigint,2) = 1', g.id) from generate_series(1, 10)
as g(id);
\gexec
```

Скрипт удаляет все строки в нечётных блоках. 5млн.строк в 10 таблицах ~3 минуты.

3) Создайте файл скрипта **initdelodd.sql**:

```
postgres@tantor:~$ mcedit initdelodd.sql

select format('delete from test.film_summary%s where
int8mod((ctid::text::point)[0]::bigint,2) = 0;', g.id) from generate_series(1,
10) as g(id);
\gexec
```

Скрипт удаляет все строки в чётных блоках.

4) Создайте файл скрипта **initins.sql**:

```
postgres@tantor:~$ mcedit initins.sql

select format('insert into test.film_summary%s select i, i , i from
generate_series(0, 10000000-1) as i;', g.id) from generate_series(1, 10) as g(id)
\gexec
```

Скрипт вставляет строки в таблицы.

5) Создайте файл скрипта **initsel.sql**:

```
postgres@tantor:~$ mcedit initsel.sql
```

```
select format('select * from test.film_summary%s where film_id = 0;', g.id) from
generate_series(1, 10) as g(id)
\gexec
```

6) Запустите в первом терминале psql и команды:

```
postgres=#
select backend_type, context, writes w, round(write_time::numeric) wt,
writebacks wb, round(writeback_time::numeric) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
\watch 1
```

Команды выводят статистику из представления **pg_stat_io** с периодичностью раз в секунду:

```
  backend_type   | context | w |  wt| wb | wbt| ex | et | hits | ev | ru | fs | fst
-----------------+---------+---+----+----+----+----+----+------+----+----+----+----
 client backend | normal  | 0 |  0 |  0 |  0 |  0 | 0  |   16 |  0 |    |  0 |  0
(1 row)
```

7) Выполните команды:

```
postgres@tantor:~$
psql -c "select pg_stat_reset_shared('io');"
psql -c "select pg_stat_reset_shared('bgwriter');"
psql -f initcre.sql 2> /dev/null > /dev/null
time psql -f initins.sql > /dev/null
psql -c "select count(*) from test.film_summary1"
time psql -f initsel.sql > /dev/null
```

```
 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)

real    3m58.679s
user    0m0.010s
sys     0m0.000s
 count
----------
10000000
(1 row)

real    1m27.860s
user    0m0.011s
sys     0m0.000s
```

Пока работает скрипт, наблюдайте в первом терминале работу background writer:

```
  backend_type    | context |   w    |   wt    |   wb   |  wbt   |   ex   |  et  |    hits    |   ev   | ru | fs |  fst
------------------+---------+--------+---------+--------+--------+--------+------+------------+--------+----+----+------
autovacuum worker | normal  |   3278 |   32.71 |      0 |      0 |      0 |    0 |       6524 |   3300 |    |  0 |    0
client backend    | normal  | 405512 | 4009.88 |      0 |      0 | 540710 | 6994 | 101113340 | 525526 |    |  0 |    0
```

```
 background writer | normal  | 112490 | 1289.38 | 112512 | 17299 |        |      |           |        |     | 0  |     0
 checkpointer      | normal  |   4387 |   38.64 |   4392 |   143 |        |      |           |        |     | 92 | 17413
 (4 rows)
```

Работа bgwriter довольно активна. Большая часть writes и writebacks при работе скриптов initcre.sql и `initins.sql` выполняется этим процессом.

8) Запустите во втором терминале команду:

```
postgres@tantor:~$
psql -c "select pg_stat_reset_shared('io');"
psql -c "select pg_stat_reset_shared('bgwriter');"
time psql -f initdeleven.sql > /dev/null

 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)

real    3m17.525s
user    0m0.010s
sys     0m0.000s
```

Скрипт удаляет все строки в чётных блоках файлов таблиц.

9) Пока выполняется скрипт удаления чётных строк, в первом терминале будут выдаваться результаты:

```
   backend_type    | context  |   w   |  wt |  wb   |  wbt | ex | et |   hits   |  ev   |  ru   | fs | fst
-------------------+----------+-------+-----+-------+------+----+----+----------+-------+-------+----+------
 autovacuum worker | normal   |   151 |   1 |     0 |    0 |  0 |  0 |     4113 |   799 |       |  0 |    0
 client backend    | bulkread | 32438 | 380 |     0 |    0 |    |    |        0 | 53955 | 54155 |    |
 client backend    | normal   |     0 |   0 |     0 |    0 |  0 |  0 | 10000278 |    81 |       |  0 |    0
 background writer | normal   | 22679 | 230 | 22656 | 3490 |    |    |          |       |       |  0 |    0
 checkpointer      | normal   |   600 |   5 |   576 |   29 |    |    |          |       |       |  5 | 2458
(5 rows)
```

Число fsyncs по файлам небольшое.

10) Процесс контрольной точки записал мало буферов. Размер таблицы больше 1/4 буферного кэша:

```
postgres@tantor:~$ psql -c "select pg_size_pretty(pg_relation_size('test.film_summary9'));"
 pg_size_pretty
----------------
 422 MB
(1 row)
```

Буферное кольцо используется для выборки. Если строки меняются, кольцо BULKREAD не используется, а кольцо BULKWRITE используется с командами массовой вставки.

Из документации к столбцу `pg_stat_io.fsyncs`: "Число вызовов fsync. Вызовы отслеживаются только в контексте normal." То есть **Операции fsyncs при работе с буферными кольцами считаются и учитываются в контексте normal** представления `pg_stat_io`. Причина:

"the number of backend fsyncs doesn't say anything about the efficiency of the BufferAccessStrategy. And counting both fsyncs done in IOCONTEXT_NORMAL and IOCONTEXT_[BULKREAD, BULKWRITE, VACUUM] under IOCONTEXT_NORMAL is likely clearer when investigating the number of backend fsyncs."

11) Выполните запросы к представлениям:

```
postgres@tantor:~$ psql -c "select backend_type, context, writes w,
round(write_time::numeric) wt, writebacks wb, round(writeback_time::numeric) wbt,
extends ex, round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs
fs, round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;"
  psql -c "select checkpoint_write_time ckpt_write_t, checkpoint_sync_time
ckpt_sync_t, buffers_checkpoint ckpt_buffers,buffers_clean bgwr_buffers,
maxwritten_clean, buffers_backend, buffers_alloc from pg_stat_bgwriter;"
```

```
  backend_type   |context |  w    | wt |  wb   |  wbt  |ex|et|   hits   |   ev   |   ru   | fs |  fst
-----------------+--------+-------+----+-------+-------+--+--+----------+--------+--------+----+-------
autovacuum worker|normal  |  2416|  24|     0 |     0 | 0| 0|    40928 |   3280 |        |  0 |     0
client backend   |bulkread|177573|2226|     0 |     0 |  |  |        0 | 269768 | 270783 |    |
client backend   |normal  |    48|   1|     0 |     0 | 0| 0| 50001666 |    227 |        |  0 |     0
background writer|normal  | 94562|1053| 94528 | 29283 |  |  |          |        |        |  0 |     0
checkpointer     |normal  |  5056|  75|  5163 |   725 |  |  |          |        |        | 29 | 16335
(5 rows)

ckpt_write_t |ckpt_sync_t |ckpt_buffers |bgwr_buffers |maxwritten_clean|buffers_backend |buffers_alloc
-------------+------------+-------------+-------------+----------------+----------------+-------------
      495800 |      16408 |        5163 |       94562 |             851|         180037 |       273312
(1 row)
```

Цветом отмечены статистики, присутствующие в обеих представлениях.

`maxwritten_clean` - Сколько раз `background writer` приостанавливал сброс грязных страниц на диск из-за того, что записал слишком много буферов. Такой статистики нет в **`pg_stat_io`**.

Если свободных буферов в кэше буферов нет (это норма), то `buffers_alloc` соответствуют `evictions`. Если свободные буфера есть, то они берутся из списка свободных статистика увеличивается, а evictions не увеличиваются

Этот пункт практики полезен, если вы использовали представление **`pg_stat_bgwriter`** для мониторинга и хотите понять как наблюдать за статистиками начиная с 17 версии.

7) Запустите тест, который читает все блоки таблиц, обновляет бит commited у строк и тем самым грязнит блоки:

```
postgres@tantor:~$ time psql -f initsel.sql > /dev/null

real    1m3.605s
user    0m0.011s
sys     0m0.000s
```

```
  backend_type   |context |   w    |  wt  |   wb   |  wbt  |ex|et|  hits    |   ev   |   ru   | fs |  fst
-----------------+--------+--------+------+--------+-------+--+--+----------+--------+--------+----+------
autovacuum worker|normal  |   3961 |   38 |      0 |     0 | 0| 0|    19684 |   4876 |        |  0 |     0
client backend   |bulkread| 256022 | 3211 |      0 |     0 |  |  |        0 | 360295 | 362282 |    |
client backend   |normal  |     44 |    1 |      0 |     0 | 0| 0|50001250  |    319 |        |  0 |     0
background worker|bulkread| 157499 | 2195 |      0 |     0 |  |  |        0 | 178032 | 180486 |    |
background worker|normal  |    174 |    3 |      0 |     0 | 0| 0|     3112 |    203 |        |  0 |     0
background writer|normal  | 121776 | 1439 | 121792 | 52859 |  |  |          |        |        |  0 |     0
checkpointer     |normal  |   4621 |   50 |   4645 |   458 |  |  |          |        |        | 40 | 23091
(7 rows)
```

8) Попробуйте самостоятельно ответить на вопросы:
Чем занимались процессы **`background worker`** в `context=bulkread`?
Сколько процессов **`background worker`** работало в `context=bulkread`?
Вы можете ответить самостоятельно. Усилия, затраченные на поиск ответа, наиболее эффективно помогут вам научиться диагностировать проблемы. Время, затраченное на поиск ответа, не важно. Результативность работы не зависит от быстроты нахождения ответа.

Часть 7. Работа bgwriter на буферном кэше 128Мб и 1Гб

1) Проверьте, как повлияет на показатели **pg_stat_io** увеличение размера буферного кэша до 1Гб. Дальше пул буферов будет увеличен и тест повторён. Выполните команды:

```
postgres@tantor:~$
psql -c "alter system set shared_buffers='1GB'"
sudo restart
```

2) В первом терминале соединение будет закрыто. Перезапустите psql и выполните команды:

```
FATAL:  terminating connection due to administrator command
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
The connection to the server was lost. Attempting reset: Failed.
The connection to the server was lost. Attempting reset: Failed.
```

```
!?> \q
postgres@tantor:~$ psql
```

```
postgres=# select backend_type, context, writes w, round(write_time::numeric)
wt, writebacks wb, round(writeback_time::numeric) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
\watch 1
```

3) Во втором терминале выполните команды:

```
postgres@tantor:~$
psql -c "select pg_stat_reset_shared('io');"
psql -c "select pg_stat_reset_shared('bgwriter');"
psql -f initcre.sql 2> /dev/null > /dev/null
time psql -f initins.sql > /dev/null
psql -c "select count(*) from test.film_summary1"
time psql -f initsel.sql > /dev/null
```

```
 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)

real    3m31.435s
user    0m0.010s
sys     0m0.000s

  count
----------
 10000000
(1 row)

real    1m21.983s
user    0m0.010s
sys     0m0.000s
```

Время уменьшилось незначительно. При одиночных последовательных вставках и чтении размер кэша не играет роли.

4) Пока работает скрипт, наблюдайте в первом терминале какие процессы выполняют ввод-вывод:

```
    backend_type    | context |   w    |  wt  |   wb   |  wbt  |   ex   |  et  |   hits    |   ev   | ru | fs | fst
--------------------+---------+--------+------+--------+-------+--------+------+-----------+--------+----+----+------
 autovacuum worker | normal  |      0 |    0 |      0 |     0 |      0 |    0 |      6566 |   2468 |    |  0 |    0
 client backend    | normal  |      0 |    0 |      0 |     0 | 540710 | 8056 | 101082280 | 410689 |    |  0 |    0
 checkpointer      | normal  | 535651 | 5273 | 535641 | 45090 |        |      |           |        |    | 82 | 8434
(3 rows)
```

В отличие от предыдущей части практики, где выполнялись те же скрипты на кэше буферов размером 128Мб, сейчас активности bgwriter нет. Все writes и writebacks при работе скриптов **`initcre.sql`** и **`initins.sql`** выполняется процессом `checkpointer`.

5) Выполните команды:

```
postgres@tantor:~$
psql -c "select pg_stat_reset_shared('io');"
psql -c "select pg_stat_reset_shared('bgwriter');"
time psql -f initdeleven.sql > /dev/null
```

```
 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)

real    3m19.089s
user    0m0.011s
sys     0m0.000s
```

Разницы во времени выполнения по сравнению с предыдущей частью (в которой размер кэша буферов был 128Мб) нет.

6) Пока выполняется скрипт удаления чётных строк, в первом терминале будут выдаваться результаты:

```
  backend_type    |context |   w    |  wt  |   wb   |  wbt  |ex|et|  hits    |   ev   |   ru   | fs | fst
------------------+--------+--------+------+--------+-------+--+--+----------+--------+--------+----+----
autovacuum worker|normal  |      0 |    0 |      0 |     0 | 0| 0|    12280 |      0 |        |  0 |   0
client backend   |bulkread|      0 |    0 |      0 |     0 |  |  |        0 | 270380 | 270170 |    |
client backend   |normal  |      0 |    0 |      0 |     0 | 0| 0|50000504 |     56 |        |  0 |   0
checkpointer     |normal  |304518 | 3291 | 304523 | 40747 |  |  |          |        |        | 25 | 400
(4 rows)
```

Число fsyncs по файлам также как и в предыдущей части небольшое.
Активности процесса bgwriter нет, всю запись выполнил checkpointer.

7) Запустите тест, который читает все блоки таблиц, обновляет бит commited и тем самым грязнит буфера:

```
postgres@tantor:~$
psql -c "select pg_stat_reset_shared('io');"
psql -c "select pg_stat_reset_shared('bgwriter');"
time psql -f initsel.sql > /dev/null
```

```
real    1m8.394s
user    0m0.010s
sys     0m0.002s
```

```
  backend_type    | context |   w    |  wt  |   wb   |  wbt  |ex |et | hits |   ev   |   ru   | fs | fst
------------------+---------+--------+------+--------+-------+---+---+------+--------+--------+----+----
```

```
autovacuum worker | normal   |      0 |    0 |       0 |      0 | 0 | 0 | 2456 |       0 |         |  0 |   0
client backend    | bulkread |    118 |    2 |       0 |      0 |   |   |    0 |   91549 |   91565 |    |
client backend    | normal   |      0 |    0 |       0 |      0 | 0 | 0 |  441 |      18 |         |  0 |   0
background worker | bulkread |    179 |    5 |       0 |      0 |   |   |    0 |  178752 |  178684 |    |
background worker | normal   |      0 |    0 |       0 |      0 | 0 | 0 | 3302 |       0 |         |  0 |   0
background writer | normal   |    100 |    1 |      64 |     41 |   |   |      |         |         |  0 |   0
checkpointer      | normal   | 230549 | 3002 | 230531 | 36715 |   |   |      |         |         | 14 | 369
(7 rows)
```

Ни в одном из тестов активности bgwriter не было, в отличие от предыдущей части практики.

8) Вы знакомы с архитектурой работы экземпляра. Попробуйте самостоятельно ответить на вопросы:

a) почему bgwriter не проявлял никакой активности?

b) какому параметру конфигурации поменять значение, чтобы bgwriter работал так же активно, как и на кэше буферов размером 128Мб?

Приложение усилий к поиску ответа на эти вопросы поможет находить простые решения, не создающие побочных эффектов.

9) Выполните команды для возврата параметров конфигурации:

```
postgres@tantor:~$
psql -c "alter system reset shared_buffers";
sudo restart
```

10) Посмотрите статистику по таблицам:

```
postgres=# select schemaname||'.'||relname name, heap_blks_read tabread,
heap_blks_hit tabhit, idx_blks_read idxread, idx_blks_hit idxhit, toast_blks_read
toastread, toast_blks_hit toasthit from pg_statio_all_tables order by 2 desc
limit 15;
            name            | tabread |  tabhit   | idxread |  idxhit  | toastread | toasthit
----------------------------+---------+-----------+---------+----------+-----------+----------
 test.film_summary1         |  162165 | 15162143 |         |          |           |
 test.film_summary2         |  132926 | 15108088 |         |          |           |
 test.film_summary8         |  108110 | 15108088 |         |          |           |
 test.film_summary9         |  108110 | 15108088 |         |          |           |
 test.film_summary6         |  108110 | 15108088 |         |          |           |
 test.film_summary5         |  108110 | 15108088 |         |          |           |
 test.film_summary3         |  108110 | 15108088 |         |          |           |
 test.film_summary7         |  108110 | 15108088 |         |          |           |
 test.film_summary4         |  108110 | 15108088 |         |          |           |
 test.film_summary10        |   89867 | 15126331 |         |          |           |
 pg_catalog.pg_attribute    |   50048 |  7792567 |   30459 | 15974323 |           |
 pg_catalog.pg_class        |   38505 | 25246260 |   12697 | 12929351 |           |
 pg_catalog.pg_statistic    |    6777 |   136160 |     684 |  1710024 |        52 |      346
 pg_catalog.pg_type         |    4324 |  1035423 |    6194 |  3487848 |         0 |        0
 pg_catalog.pg_depend       |    3654 |  3894142 |   13502 | 12385733 |           |
(15 rows)
```

11) Запрос для просмотра статистики по индексам:

```
postgres=# select schemaname||'.'||relname table, indexrelname index,
idx_blks_read idxread, idx_blks_hit idxhit from pg_statio_all_indexes order by 4
desc limit 5;
           table           |                index                | idxread |  idxhit
---------------------------+-------------------------------------+---------+---------
 pg_catalog.pg_attribute   | pg_attribute_relid_attnum_index     |   14157 | 12490534
 pg_catalog.pg_class       | pg_class_oid_index                  |    3220 |  7968689
 pg_catalog.pg_depend      | pg_depend_depender_index            |    7466 |  7349793
 pg_catalog.pg_depend      | pg_depend_reference_index           |    6036 |  5035940
 pg_catalog.pg_class       | pg_class_relname_nsp_index          |    7815 |  3985553
(5 rows)
```

"**idx_blks_hit**" блок находился в буферном кэше.

"**idx_blks_read**" блок отсутствовал в буферном кэше. Процессу приходилось очищать буфер (если список свободных блоков пуст), загружать в очищенный буфер блок и только потом читать его содержимое.

Статистика по таблицам и индексам полезна тем, что позволяет определить "горячие" объекты - те, к которым чаще всего идёт доступ.

12) Удалите схему test с тестовыми таблицами, чтобы освободить место на диске:

```
postgres=# drop schema test cascade;
NOTICE:  drop cascades to 10 other objects
DETAIL:  drop cascades to table test.film_summary1
drop cascades to table test.film_summary2
drop cascades to table test.film_summary3
drop cascades to table test.film_summary4
drop cascades to table test.film_summary5
drop cascades to table test.film_summary6
drop cascades to table test.film_summary7
drop cascades to table test.film_summary8
drop cascades to table test.film_summary9
drop cascades to table test.film_summary10
DROP SCHEMA
```

16) Посмотрите примеры запросов к статистике по таблицам:

```
postgres=# select schemaname||'.'||relname name, seq_scan, idx_scan,
idx_tup_fetch, autovacuum_count, autoanalyze_count from pg_stat_all_tables where
idx_scan is not null order by 3 desc limit 5;
          name           | seq_scan | idx_scan | idx_tup_fetch | autovacuum_count | autoanalyze_count
-------------------------+----------+----------+---------------+------------------+-------------------
 pg_catalog.pg_class     |    31253 |  4330223 |       3274290 |               83 |               106
 pg_catalog.pg_attribute |     1565 |  3036625 |       9054711 |               90 |               105
 pg_catalog.pg_depend    |        3 |  2389005 |       2722183 |               83 |               103
 pg_catalog.pg_type      |        3 |  1093705 |        326604 |               80 |               104
 pg_catalog.pg_statistic |        2 |   823316 |         43138 |                9 |                 0
(5 rows)
```

```
postgres=#  select relname name, n_tup_ins ins, n_tup_upd upd, n_tup_del del,
n_tup_hot_upd hot_upd, n_tup_newpage_upd newblock, n_live_tup live, n_dead_tup
dead, n_ins_since_vacuum sv, n_mod_since_analyze sa from pg_stat_all_tables where
idx_scan is not null order by 3 desc limit 5;
      name      |   ins   |  upd   |   del   | hot_upd | newblock | live | dead | sv  |  sa
----------------+---------+--------+---------+---------+----------+------+------+-----+-------
 pg_class       |  459654 | 156877 |  476171 |  144372 |     9275 |  447 |   40 |  30 |    70
 pg_statistic   |   30034 |  12473 |   30032 |   12160 |      313 |  424 |   59 |   0 | 72539
 pg_shdepend    |       0 |      0 |       0 |       0 |        0 |    0 |    0 |   0 |     0
 pg_attribute   | 3065349 |      0 | 3175464 |       0 |        0 | 3340 |  360 | 270 |    90
 pg_type        |  306888 |      0 |  317898 |       0 |        0 |  643 |   80 |  60 |    20
(5 rows)
```

```
postgres=# select schemaname||'.'||relname name, indexrelname, idx_scan,
round(extract('epoch' from clock_timestamp() -
   last_idx_scan)) scan_sec, idx_tup_read, idx_tup_fetch from pg_stat_all_indexes
order by 3 desc limit 5;
          name           |           indexrelname           | idx_scan | scan_sec | idx_tup_read | idx_tup_fetch
-------------------------+----------------------------------+----------+----------+--------------+---------------
 pg_catalog.pg_class     | pg_class_oid_index               |  3493066 |        3 |      3051762 |       3032453
 pg_catalog.pg_attribute | pg_attribute_relid_attnum_index  |  3036645 |        3 |      9054760 |       9054759
 pg_catalog.pg_depend    | pg_depend_depender_index         |  1591566 |      598 |      1909239 |       1909239
 pg_catalog.pg_class     | pg_class_relname_nsp_index       |   837469 |        3 |       244094 |        241270
 pg_catalog.pg_statistic | pg_statistic_relid_att_inh_index |   823325 |      130 |        43329 |         43138
(5 rows)
```

idx_tup_read число индексных записей, возвращённых при сканированиях этого индекса. Значение увеличивается всякий раз, когда считывается индексная запись.

idx_tup_fetch число актуальных строк таблицы, выбранных с помощью индексной записи при простом сканировании этого индекса (Index Scan).

13) Поиск таблиц, к которым давно не было обращений:

```
postgres=# select schemaname schema, relname table,
pg_size_pretty(pg_total_relation_size(relid)) size, seq_scan, idx_scan,
last_seq_scan::time seq_scan_t, last_idx_scan::time idx_scan_t from
pg_stat_all_tables where schemaname not in
('pg_catalog','information_schema','pg_toast') order by 4, 5;
  schema  |      table       |    size    | seq_scan | idx_scan |   seq_scan_t    | idx_scan_t
----------+------------------+------------+----------+----------+-----------------+-----------
 public   | pgbench_history  | 0 bytes    |        0 |          |                 |
 public   | pgbench_branches | 56 kB      |        1 |        0 | 13:57:07.951584 |
 public   | pgbench_tellers  | 56 kB      |        1 |        0 | 13:57:07.951584 |
 public   | shmem_reference  | 16 kB      |        1 |          | 13:02:03.21597  |
 public   | test             | 192 kB     |        1 |          | 13:42:12.078525 |
 columnar | options          | 8192 bytes |        2 |        0 | 21:32:08.85896  |
 columnar | chunk_group      | 8192 bytes |        2 |        0 | 21:32:08.85896  |
 columnar | chunk            | 16 kB      |        2 |        0 | 21:32:08.85896  |
 public   | pgbench_accounts | 15 MB      |        2 |        0 | 13:57:07.951584 |
 columnar | stripe           | 16 kB      |        3 |        0 | 21:32:08.85896  |
 columnar | row_mask         | 32 kB      |        4 |        0 | 21:32:08.85896  |
 test     | film_summary3    | 422 MB     |        7 |          | 20:12:27.735492 |
 test     | film_summary4    | 422 MB     |        7 |          | 20:12:34.973333 |
 test     | film_summary10   | 422 MB     |        7 |          | 20:13:19.345392 |
 test     | film_summary6    | 422 MB     |        7 |          | 20:12:50.135028 |
 test     | film_summary7    | 422 MB     |        7 |          | 20:12:58.123483 |
 test     | film_summary8    | 422 MB     |        7 |          | 20:13:04.938543 |
 test     | film_summary9    | 422 MB     |        7 |          | 20:13:12.069751 |
 test     | film_summary5    | 422 MB     |        7 |          | 20:12:42.720495 |
 test     | film_summary2    | 422 MB     |        7 |          | 20:12:21.332072 |
 test     | film_summary1    | 422 MB     |       10 |          | 20:12:15.237606 |
(21 rows)
```

14) Поиск давно не использовавшихся индексов:

```
postgres=# select s.schemaname schema, s.relname table, indexrelname index,
pg_size_pretty(pg_relation_size(s.indexrelid)) size, s.idx_scan scans,
s.last_idx_scan::time last_scan from pg_stat_all_indexes s join pg_index i on
s.indexrelid = i.indexrelid where s.schemaname not in
('information_schema','pg_toast') and not i.indisunique order by 5, 6;
   schema   |       table        |                index                |    size    |  scans  |    last_scan
------------+--------------------+-------------------------------------+------------+---------+----------------
 pg_catalog | pg_publication_rel | pg_publication_rel_prpubid_index    | 8192 bytes |       0 |
 pg_catalog | pg_constraint      | pg_constraint_conparentid_index     | 16 kB      |       0 |
 pg_catalog | pg_inherits        | pg_inherits_parent_index            | 8192 bytes |       0 |
 pg_catalog | pg_trigger         | pg_trigger_tgconstraint_index       | 8192 bytes |       0 |
 pg_catalog | pg_auth_members    | pg_auth_members_grantor_index       | 16 kB      |       0 |
 pg_catalog | pg_shdepend        | pg_shdepend_reference_index         | 8192 bytes |       0 |
 pg_catalog | pg_class           | pg_class_tblspc_relfilenode_index   | 1008 kB    |       0 |
 pg_catalog | pg_constraint      | pg_constraint_contypid_index        | 16 kB      |       0 |
 pg_catalog | pg_constraint      | pg_constraint_conname_nsp_index     | 16 kB      |       6 | 13:57:07.951584
 pg_catalog | pg_index           | pg_index_indrelid_index             | 424 kB     |   65381 | 00:42:46.840573
 pg_catalog | pg_statistic_ext   | pg_statistic_ext_relid_index        | 8192 bytes |  186119 | 00:42:31.871778
 pg_catalog | pg_shdepend        | pg_shdepend_depender_index          | 8192 bytes |  794253 | 23:50:14.82469
 pg_catalog | pg_depend          | pg_depend_reference_index           | 3928 kB    |  797566 | 23:50:14.846317
 pg_catalog | pg_depend          | pg_depend_depender_index            | 4664 kB    | 1591819 | 23:50:14.82469
(14 rows)
```

15) Для проверки правильности ответа: в названии параметра конфигурации из 8 пункта 12 знаков.

Часть 8. Использование расширения pg_walinspect

Расширение появилось в 15 версии. Функции расширения выдают те же данные, что и утилита командной строки pg_waldump. В этой части практики вы посмотрите удобно ли работать с функциями расширения и какие у них возможности.

1) Установите расширение и посмотрите содержимое расширения:

```
postgres=# create extension pg_walinspect;
\dx+ pg_walinspect
CREATE EXTENSION
         Objects in extension "pg_walinspect"
                  Object description
-------------------------------------------------------
 function pg_get_wal_block_info(pg_lsn,pg_lsn,boolean)
 function pg_get_wal_record_info(pg_lsn)
 function pg_get_wal_records_info(pg_lsn,pg_lsn)
 function pg_get_wal_stats(pg_lsn,pg_lsn,boolean)
(4 rows)
```

Расширение состоит из четырёх функций.

2) Выполните запрос:

```
postgres=# select "resource_manager/record_type" type, count,
round(count_percentage) "count%", record_size, round(record_size_percentage)
"size%", fpi_size, round(fpi_size_percentage(tantor)) "fpi%", combined_size,
round(combined_size_percentage) "total%" from pg_get_wal_stats((select
'0/0'::pg_lsn + (pg_split_walfile_name(name(oleg))).segment_number * size from
pg_ls_waldir() oleg order by name limit 1), 'FFFFFFFF/FFFFFFFF', false) tantor
where count<>0 order by 8 desc;
    type     | count | count% | record_size | size% | fpi_size | fpi% | combined_size | total%
-------------+-------+--------+-------------+-------+----------+------+---------------+--------
 XLOG        |  1814 |     48 |       92786 |    10 | 14798492 |   99 |      14891278 |     94
 Heap2       |  1821 |     49 |      780143 |    88 |     7300 |    0 |        787443 |      5
 Heap        |    45 |      1 |        7101 |     1 |    55904 |    0 |         63005 |      0
 Btree       |    55 |      1 |        4264 |     0 |    36976 |    0 |         41240 |      0
 Transaction |     1 |      0 |         887 |     0 |        0 |    0 |           887 |      0
 Standby     |     7 |      0 |         476 |     0 |        0 |    0 |           476 |      0
(6 rows)
```

Функция pg_get_wal_stats(start_lsn,end_lsn, per_record boolean) показывает статистику по журнальным записям между двумя LSN.
end_lsn можно указать максимальным 'FFFFFFFF/FFFFFFFF'.

Полные образы страниц fpi занимают существенное место.

3) Если третий параметр true, то выдаётся статистика ещё и по типу журнальной записи record_type, а не суммарная по каждому resource_manager. Повторите запрос указав per_record=true:

```
postgres=# select "resource_manager/record_type" type, count,
round(count_percentage) "count%", record_size, round(record_size_percentage)
"size%", fpi_size, round(fpi_size_percentage(tantor)) "fpi%", combined_size,
round(combined_size_percentage) "total%" from pg_get_wal_stats((select
'0/0'::pg_lsn + (pg_split_walfile_name(name(oleg))).segment_number * size from
pg_ls_waldir() oleg order by name limit 1), 'FFFFFFFF/FFFFFFFF', true) tantor
where count<>0 order by 8 desc;
          type          | count | count% | record_size | size% | fpi_size | fpi% | combined_size | total%
------------------------+-------+--------+-------------+-------+----------+------+---------------+--------
 XLOG/FPI_FOR_HINT      |  1810 |     48 |       92310 |    10 | 14798492 |   99 |      14890802 |     94
 Heap2/PRUNE            |  1808 |     48 |      779040 |    88 |        0 |    0 |        779040 |      5
```

```
 Btree/INSERT_LEAF         |    55 |       1 |         4264 |       0 |     36976 |     0 |          41240 |       0
 Heap/INSERT               |    13 |       0 |         3285 |       0 |     36752 |     0 |          40037 |       0
 Heap/UPDATE               |     3 |       0 |          261 |       0 |     19152 |     0 |          19413 |       0
 Heap2/MULTI_INSERT        |    13 |       0 |         1103 |       0 |      7300 |     0 |           8403 |       0
 Heap/HOT_UPDATE           |    15 |       0 |         1861 |       0 |         0 |     0 |           1861 |       0
 Transaction/COMMIT        |     1 |       0 |          887 |       0 |         0 |     0 |            887 |       0
 Heap/UPDATE+INIT          |     1 |       0 |          851 |       0 |         0 |     0 |            851 |       0
 Heap/DELETE               |     8 |       0 |          480 |       0 |         0 |     0 |            480 |       0
 Standby/RUNNING_XACTS     |     7 |       0 |          476 |       0 |         0 |     0 |            476 |       0
 XLOG/CHECKPOINT_ONLINE    |     2 |       0 |          296 |       0 |         0 |     0 |            296 |       0
 Heap/LOCK                 |     4 |       0 |          240 |       0 |         0 |     0 |            240 |       0
 XLOG/CHECKPOINT_SHUTDOWN  |     1 |       0 |          148 |       0 |         0 |     0 |            148 |       0
 Heap/INSERT+INIT          |     1 |       0 |          123 |       0 |         0 |     0 |            123 |       0
 XLOG/NEXTOID              |     1 |       0 |           32 |       0 |         0 |     0 |             32 |       0
(16 rows)
```

Строк вывелось больше и в результат добавился **record_type**.

4) Посмотрите полный список менеджеров ресурсов:

```
postgres=# select * from pg_get_wal_resource_managers();
 rm_id |       rm_name       | rm_builtin
-------+---------------------+------------
     0 | XLOG                | t
     1 | Transaction         | t
     2 | Storage             | t
     3 | CLOG                | t
     4 | Database            | t
     5 | Tablespace          | t
     6 | MultiXact           | t
     7 | RelMap              | t
     8 | Standby             | t
     9 | Heap2               | t
    10 | Heap                | t
    11 | Btree               | t
    12 | Hash                | t
    13 | Gin                 | t
    14 | Gist                | t
    15 | Sequence            | t
    16 | SPGist              | t
    17 | BRIN                | t
    18 | CommitTs            | t
    19 | ReplicationOrigin   | t
    20 | Generic             | t
    21 | LogicalMessage      | t
    22 | Heap3               | t
(23 rows)
```

Библиотеки расширений могут регистрировать свои менеджеры ресурсов.

5) Есть представление со статистикой по журналам. Представление не входит в расширение, оно стандартное. Посмотрите, что выдаётся в представлении:

```
postgres=# select * from pg_stat_wal\gx
-[ RECORD 1 ]----+------------------------------
wal_records      | 1304529873
wal_fpi          | 9623757
wal_bytes        | 166102694837
wal_buffers_full | 8083465
wal_write        | 8387587
wal_sync         | 293575
wal_write_time   | 0
wal_sync_time    | 0
stats_reset      | 2025-01-01 01:01:33.035882+03
```

6) Статистика накапливается с момента создания кластера или сброса статистики функцией **pg_stat_reset_shared('wal')**. Выполните команды:

```
postgres=#
select pg_stat_reset_shared('wal');
select * from pg_stat_wal\gx \\
 pg_stat_reset_shared
----------------------

(1 row)

-[ RECORD 1 ]----+----------------------
wal_records      | 0
wal_fpi          | 0
wal_bytes        | 0
wal_buffers_full | 0
wal_write        | 0
wal_sync         | 0
wal_write_time   | 0
wal_sync_time    | 0
stats_reset      | 2035-01-01 01:01:55.994752+03
```

7) Уменьшите число WAL-файлов:

```
postgres@tantor:~$
psql -c "alter system set wal_recycle = off;"
psql -c "select pg_reload_conf();"
psql -c "checkpoint;"
psql -c "alter system set wal_recycle = on;"
psql -c "select pg_reload_conf();"
ALTER SYSTEM
 pg_reload_conf
----------------
 t
(1 row)

CHECKPOINT
ALTER SYSTEM
 pg_reload_conf
----------------
 t
(1 row)
```

8) посмотрите сколько журнальных файлов осталось:

```
postgres@tantor:~$ ls $PGDATA/pg_wal
000000010000000250000000D8  000000010000000250000000D9  000000010000000250000000DA
000000010000000250000000DB  000000010000000250000000DC  archive_status
```

9) Выведите список файлов журналов и LSN начала каждого WAL-файла:

```
postgres=# select name, '0/0'::pg_lsn +
(pg_split_walfile_name(name(oleg))).segment_number * size min from pg_ls_waldir()
oleg order by 1;
             name             |        min
------------------------------+-------------------
 000000010000000250000000DC | 00000025/DC000000
 000000010000000250000000DB | 00000025/DB000000
 000000010000000250000000DA | 00000025/DA000000
 000000010000000250000000D9 | 00000025/D9000000
 000000010000000250000000D8 | 00000025/D8000000
(5 rows)
```

10) Выполните запрос с функцией `pg_get_wal_stats()`, который вы ранее уже выполняли:

```
postgres=#
select "resource_manager/record_type" type, count, round(count_percentage)
"count%", record_size, round(record_size_percentage) "size%", fpi_size,
round(fpi_size_percentage(tantor)) "fpi%", combined_size,
round(combined_size_percentage) "total%" from pg_get_wal_stats((select
'0/0'::pg_lsn + (pg_split_walfile_name(name(oleg))).segment_number * size from
pg_ls_waldir() oleg order by name limit 1), 'FFFFFFFF/FFFFFFFF', true) tantor
where count<>0 order by 8 desc;
           type          | count | count% | record_size | size% | fpi_size | fpi% | combined_size | total%
-------------------------+-------+--------+-------------+-------+----------+------+---------------+--------
 XLOG/CHECKPOINT_ONLINE  |     1 |     33 |         148 |    52 |        0 |    0 |           148 |     52
 Standby/RUNNING_XACTS   |     2 |     67 |         136 |    48 |        0 |    0 |           136 |     48
(2 rows)
```

Файлов WAL пять, четыре из них полностью пустые. Сохранены записи только за последнюю контрольную точку.

11) Хочется посмтреть что в самих журнальных записях. Посмотрите текст функции **pg_get_wal_records_info()**:

```
postgres=# \sf pg_get_wal_records_info
CREATE OR REPLACE FUNCTION public.pg_get_wal_records_info(start_lsn pg_lsn,
end_lsn pg_lsn, OUT start_lsn pg_lsn, OUT end_lsn pg_lsn, OUT prev_lsn pg_lsn,
OUT xid xid, OUT resource_manager text, OUT record_type text, OUT record_length
integer, OUT main_data_length integer, OUT fpi_length integer, OUT description
text, OUT block_ref text)
 RETURNS SETOF record
 LANGUAGE c
 PARALLEL SAFE STRICT
AS '$libdir/pg_walinspect', $function$pg_get_wal_records_info$function$
```

У функции два входных параметра и много выходных.

12) Воспользуйтесь функцией **pg_get_wal_records_info**, чтобы найти журнальные записи об окончании контрольной точки:

```
postgres=# select resource_manager res, record_type, start_lsn, end_lsn,
prev_lsn, record_length len, left(description,20) from
pg_get_wal_records_info((select '0/0'::pg_lsn +
(pg_split_walfile_name(name(oleg))).segment_number * size from pg_ls_waldir()
oleg order by name limit 1), 'FFFFFFFF/FFFFFFFF') tantor where record_type like
'CHECKPOINT%' order by 3 limit 20;
 res  |    record_type    |  start_lsn  |   end_lsn   |  prev_lsn   | len |         left
------+-------------------+-------------+-------------+-------------+-----+----------------------
 XLOG | CHECKPOINT_ONLINE | 25/D8000070 | 25/D8000108 | 25/D8000028 | 148 | redo 25/D8000028; tl
 XLOG | CHECKPOINT_ONLINE | 25/D8001FE8 | 25/D8002098 | 25/D8001FA0 | 148 | redo 25/D8001FA0; tl
 XLOG | CHECKPOINT_ONLINE | 25/D8007B90 | 25/D8007C28 | 25/D8007B48 | 148 | redo 25/D8007B48; tl
(3 rows)
```

Записи о начале контрольной точки не существует. Запись об окончании контрольной точки имеет идентификатор с буквосочетанием **CHECKPOINT**.

13) Посмотрите всё, что может выдать функция по какой-нибудь журнальной записи:

```
postgres=# \pset xheader_width 40
Expanded header width is 40.
postgres=# select * from pg_get_wal_records_info((select '0/0'::pg_lsn +
(pg_split_walfile_name(name(oleg))).segment_number * size from pg_ls_waldir()
oleg order by name limit 1), 'FFFFFFFF/FFFFFFFF') tantor where record_type like
'CHECKP%' order by 3 limit 1\gx
```

```
-[ RECORD 1 ]----+----------------------
start_lsn        | 25/D8000070
end_lsn          | 25/D8000108
prev_lsn         | 25/D8000028
xid              | 0
resource_manager | XLOG
record_type      | CHECKPOINT_ONLINE
record_length    | 148
main_data_length | 120
fpi_length       | 0
description      | redo 25/D8000028; tli 1; prev tli 1; fpw true; xid 361549; oid 942308;
multi 1; offset 0; oldest xid 723 in DB 1; oldest multi 1 in DB 5; oldest/newest commit
timestamp xid: 0/0; oldest running xid 361549; online
block_ref        |
```

14) Посмотрите на выданную запись утилитой `pg_waldump`:

```
postgres=# \! pg_waldump -s 25/D8000070 -e 25/D8000108
rmgr: XLOG        len (rec/tot):    148/   148, tx: 0, lsn: 25/D8000070, prev
25/D8000028, desc: CHECKPOINT_ONLINE redo 25/D8000028; tli 1; prev tli 1; fpw
true; xid 361549; oid 942308; multi 1; offset 0; oldest xid 723 in DB 1; oldest
multi 1 in DB 5; oldest/newest commit timestamp xid: 0/0; oldest running xid
361549; online
```

Функция из расширения выдаёт то же самое, что и утилита **pg_waldump** и дополнительно выдаёт расчётные данные.

15) Выведите журнальную запись функцией `pg_get_wal_record_info`, которая выводит одну журнальную запись, а не диапазон и поэтому имеет один входной параметр:

```
postgres=# select * from pg_get_wal_record_info('25/D8000070')\gx
-[ RECORD 1 ]----+----------------------
start_lsn        | 25/D8000070
end_lsn          | 25/D8000108
prev_lsn         | 25/D8000028
xid              | 0
resource_manager | XLOG
record_type      | CHECKPOINT_ONLINE
record_length    | 148
main_data_length | 120
fpi_length       | 0
description      | redo 25/D8000028; tli 1; prev tli 1; fpw true; xid 361549; oid 942308;
multi 1; offset 0; oldest xid 723 in DB 1; oldest multi 1 in DB 5; oldest/newest commit
timestamp xid: 0/0; oldest running xid 361549; online
block_ref        |
```

Функция **pg_get_wal_record_info(in_lsn)** выдаёт те же самые данные, что и функция **pg_get_wal_records_info(start_lsn, end_lsn)**, но только по одной журнальной записи.

Для просмотра содержимого журнала можно использовать функции стандартного расширения `pg_walinspect` и утилиту командной строки `pg_waldump` в зависимости от того, что привычнее в использовании. Для фильтрации записей журнала с расширением можно использовать навыки фильтрации строк выражениями SQL. С `pg_waldump` для фильтрации записей приходится использовать утилиты grep, egrep, sed.

Часть 9. Наблюдение за блокировками

Будут использоваться три сессии. Откройте три терминала и запустите в них psql.

1) В первой сессии выполните команду блокировки какой-нибудь таблицы:

```
postgres=# begin;
lock table pgbench_accounts;
BEGIN
LOCK TABLE
postgres=*#
```

Команда блокировки работает в транзакции, поэтому была открыта транзакция.

2) Во второй сессии выполните запросы:

```
postgres=# select pid, left(backend_type, 14) type, left(state,6) state,
wait_event_type wait_type, wait_event, age(backend_xid) age1, age(backend_xmin)
age2, round(extract('epoch' from clock_timestamp() - xact_start)) sec,
left(query,28) query from pg_stat_activity order by 1;
  select pl.pid, pl.mode, pl.granted g, pl.relation::regclass, a.wait_event_type
wait, a.wait_event event, round(extract('epoch' from clock_timestamp() -
a.query_start)) sec, left(a.query,28) query from pg_locks pl left join
pg_stat_activity a ON pl.pid = a.pid where a.datname = current_database() order
by 1;
```

```
  pid   |      type       | state  | wait_type |     wait_event      | age1 | age2 | sec |            query
--------+-----------------+--------+-----------+---------------------+------+------+-----+-----------------------------
 699375 | checkpointer    |        | Activity  | CheckpointerMain    |      |      |     |
 699376 | background wri  |        | Activity  | BgWriterHibernate   |      |      |     |
 699378 | walwriter       |        | Activity  | WalWriterMain       |      |      |     |
 699379 | autovacuum lau  |        | Activity  | AutoVacuumMain      |      |      |     |
 699380 | logical replic  |        | Activity  | LogicalLauncherMain |      |      |     |
 762650 | client backend  | idle i | Client    | ClientRead          |    1 |      | 903 | lock table pgbench_accounts;
 762653 | client backend  | active |           |                     |      |    1 |   0 | select pid, left(backend_typ
(7 rows)

  pid   |        mode         | g |         relation          |  wait   |   event    | sec |            query
--------+---------------------+---+---------------------------+---------+------------+-----+------------------------------
 762650 | ExclusiveLock       | t |                           | Client  | ClientRead | 903 | lock table pgbench_accounts;
 762650 | ExclusiveLock       | t |                           | Client  | ClientRead | 903 | lock table pgbench_accounts;
 762650 | AccessExclusiveLock | t | pgbench_accounts          | Client  | ClientRead | 903 | lock table pgbench_accounts;
 762653 | AccessShareLock     | t | pg_authid_oid_index       |         |            |   0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock     | t | pg_database_oid_index     |         |            |   0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock     | t | pg_authid_rolname_index   |         |            |   0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock     | t | pg_database_datname_index |         |            |   0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock     | t | pg_authid                 |         |            |   0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock     | t | pg_stat_activity          |         |            |   0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock     | t | pg_database               |         |            |   0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock     | t | pg_locks                  |         |            |   0 | select pl.pid, pl.mode, pl.g
 762653 | ExclusiveLock       | t |                           |         |            |   0 | select pl.pid, pl.mode, pl.g
(12 rows)
```

В представлениях много столбцов и они не помещаются на странице, поэтому используется два запроса. Второй запрос выбирает значения, соединяя два представления.

Для мониторинга блокировок используется `pg_locks`. Для того чтобы детально узнать, какую команду выполняет процесс используется представление `pg_stat_activity`.

В столбцах sec выдаётся число секунд от момента на который выполнялся запрос. Если запросы выполнить в разное время, то значения будут разные.

3) Создадим ситуацию, когда третья сессия ждёт снятия блокировки, установленной первой сессией. В третьей сессии выполните запрос к заблокированной таблице:

```
postgres=# select count(*) from pgbench_accounts;
```

Сессия заблокировалась и не возвращает промпт.

4) Во второй сессии повторите два запроса:

```
postgres=# select pid, left(backend_type, 14) type, left(state,6) state,
wait_event_type wait_type, wait_event, age(backend_xid) age1, age(backend_xmin)
age2, round(extract('epoch' from clock_timestamp() - xact_start)) sec,
left(query,28) query from pg_stat_activity order by 1;
  select pl.pid, pl.mode, pl.granted g, pl.relation::regclass, a.wait_event_type
wait, a.wait_event event, round(extract('epoch' from clock_timestamp() -
a.query_start)) sec, left(a.query,28) query from pg_locks pl left join
pg_stat_activity a ON pl.pid = a.pid where a.datname = current_database() order
by 1;
```

```
  pid   |      type       | state  | wait_type |     wait_event      | age1 | age2 | sec  |            query
--------+-----------------+--------+-----------+---------------------+------+------+------+-----------------------------
 699375 | checkpointer    |        | Activity  | CheckpointerMain    |      |      |      |
 699376 | background wri  |        | Activity  | BgWriterHibernate   |      |      |      |
 699378 | walwriter       |        | Activity  | WalWriterMain       |      |      |      |
 699379 | autovacuum lau  |        | Activity  | AutoVacuumMain      |      |      |      |
 699380 | logical replic  |        | Activity  | LogicalLauncherMain |      |      |      |
 762650 | client backend  | idle i | Client    | ClientRead          |    1 |      | 1451 | lock table pgbench_accounts;
 762653 | client backend  | active |           |                     |      |    1 |    0 | select pid, left(backend typ
 763886 | client backend  | active | Lock      | relation            |      |    1 |  129 | select count(*) from pgbench
(8 rows)

  pid   |        mode         | g |         relation          |  wait   |   event    | sec  |            query
--------+---------------------+---+---------------------------+---------+------------+------+-----------------------------
 762650 | ExclusiveLock       | t |                           | Client  | ClientRead | 1451 | lock table pgbench_accounts;
 762650 | AccessExclusiveLock | t | pgbench_accounts          | Client  | ClientRead | 1451 | lock table pgbench_accounts;
 762650 | ExclusiveLock       | t |                           | Client  | ClientRead | 1451 | lock table pgbench_accounts;
 762653 | ExclusiveLock       | t |                           |         |            |    0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock     | t | pg_authid_oid_index       |         |            |    0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock     | t | pg_database_oid_index     |         |            |    0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock     | t | pg_stat_activity          |         |            |    0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock     | t | pg_authid_rolname_index   |         |            |    0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock     | t | pg_database_datname_index |         |            |    0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock     | t | pg_authid                 |         |            |    0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock     | t | pg_database               |         |            |    0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock     | t | pg_locks                  |         |            |    0 | select pl.pid, pl.mode, pl.g
 763886 | AccessShareLock     | f | pgbench_accounts          | Lock    | relation   |  129 | select count(*) from pgbench
 763886 | ExclusiveLock       | t |                           | Lock    | relation   |  129 | select count(*) from pgbench
(14 rows)
```

В результатах добавились строки. На то, что сессия заблокирована во втором запросе указывает `pg_locks.granted=false`. В первом запросе на блокировку указывает `wait_type='Lock'`. Длительность выполнения команды, включая ожидания выдаются в столбцах `pg_stat_activity` `xact_start` и `query_start`. Значения в столбцах:

`xact_start` - начало открытой транзакции или null, если транзакция не открыта. Совпадает с началом первой команды в транзакции. **Непустое значение в `xact_start` означает, что удерживаеттся горизонт базы данных.**

`query_start` - начало выполняющегося в данный момент запроса или длительность последнего запроса (в этом случае значение столбца `state<>'active'`). Пока работет запрос, горизонт базы данных удерживается.

5) Для мониторинга горизонта баз данных можно использовать запросы:

```
postgres=# select datname database, trunc(extract(epoch from
max(clock_timestamp()-xact_start))) sec, greatest(max(age(backend_xmin)),
max(age(backend_xid))) age1 from pg_stat_activity where backend_xmin is not null
or backend_xid is not null group by datname order by datname;
  select pid, trunc(extract(epoch from (clock_timestamp()-xact_start))) sec,
age(backend_xid) age1, age(backend_xmin) age2, datname database, usename, state,
wait_event, left(query,20) query from pg_stat_activity where backend_xmin is not
null or backend_xid is not null order by greatest(age(backend_xmin),
age(backend_xid)) desc;
```

```
 database | sec  | age1
----------+------+------
 postgres | 1458 |    1
(1 row)

  pid   | sec  | age1 | age2 | database | usename  |        state        | wait_event |        query
--------+------+------+------+----------+----------+---------------------+------------+---------------------
 762650 | 1458 |    1 |      | postgres | postgres | idle in transaction | ClientRead | lock table pgbench_a
 763886 |  871 |      |    1 | postgres | postgres | active              | relation   | select count(*) from
 765653 |    0 |      |    1 | postgres | postgres | active              |            | select pid, trunc(ex
(3 rows)
```

Первый запрос выдаёт длительность удержания горизонта по базам данных.
Второй запрос выдаёт все процессы, и длительность их транзакций.
Долгое время удержания горизонта существенно снижает производительность.

6) Все предыдущие запросы не удобны для поиска заблокированных или блокирующих процессов.
Более удобные запросы для вывода заблокированных и блокирующих процессов:

```
postgres=# select pl.pid blocked, pl.mode, a.wait_event_type wait, a.wait_event
event, age(a.backend_xmin) age_xmin, round(extract('epoch' from clock_timestamp()
- a.query_start)) waitsec, left(a.query,40) blocked_query from pg_locks pl left
join pg_stat_activity a ON pl.pid = a.pid where pl.granted='f' and a.datname =
current_database();
  select a.pid blocked, bl.pid blocker, a.wait_event_type wait, a.wait_event
event, age(a.backend_xmin) age, round(extract('epoch' from clock_timestamp() -
a.query_start)) waitsec, bl.wait_event_type bl_type, bl.wait_event bl_wait,
round(extract('epoch' from clock_timestamp() - bl.query_start)) bl_sec,
left(a.query,10) blocked_q, left(bl.query,10) blocker_q from pg_stat_activity a
join pg_stat_activity bl on bl.pid = ANY(pg_blocking_pids(a.pid)) where
a.wait_event_type='Lock' and cardinality(pg_blocking_pids(a.pid))>0;
```

```
blocked |     mode       | wait |  event   | age_xmin | waitsec |             blocked_query
--------+----------------+------+----------+----------+---------+---------------------------------------
 763886 |AccessShareLock | Lock | relation |        1 |    1436 | select count(*) from pgbench_accounts;
(1 row)

blocked |blocker | wait |  event   | age | waitsec | bl_type |  bl_wait   | bl_sec | blocked_q  | blocker_q
--------+--------+------+----------+-----+---------+---------+------------+--------+------------+----------
 763886 | 762650 | Lock | relation |   1 |    1436 | Client  | ClientRead |   2758 | select cou | lock table
(1 row)
```

Первый запрос показывает заблокированный процесс и команду, которую он выполняет.
Второй запрос показывает номер блокирующего процесса. Номер выдаётся функцией , которая дорогостоящая в выполнении, так как устанавливает блокировки. Частые вызовы этой функции снижают производительность. Использование этой функции в периодически выполняющихся скриптах мониторинга нежелательно.
Для поиска нет ли процессов, которые заблокированы лучше сначала использовать первый запрос: обращение к `pg_locks`. Если запрос выдаст процессы, у которых `granted='f'`., то тогда можно использовать функцию `pg_blocking_pids(pid)` для поиска процессов-блокировщиков.

7) Выполните команду:

```
postgres=# select pg_cancel_backend(762650);
 pg_cancel_backend
-------------------
 t
(1 row)
```

После выполнения функции ничего не изменилось. Блокирующая сессия простаивает, а эта функция прерывает выполнение команд. Если бы выполнялась команда, которая установила блокировку, то функция бы прервала выполнение команды и, возможно, блокировка была бы снята. Если блокировка не снимется или сессия простаивает, то можно использовать функцию, которая прерывает сессию:

8) Выполните команду:

```
postgres=# select pg_terminate_backend(762650);
 pg_cancel_backend
-------------------
 t
(1 row)
```

Во второй сессии команда получила блокировку и успешно выполнилась:

```
 count
--------
 100000
(1 row)
```

В первой сессии, которая простаивает, изменений нет.

9) Во второй сессии повторите запросы:

Во второй сессии выполните запросы:

```
postgres=# select pid, left(backend_type, 14) type, left(state,6) state,
wait_event_type wait_type, wait_event, age(backend_xid) age1, age(backend_xmin)
age2, round(extract('epoch' from clock_timestamp() - xact_start)) sec,
left(query,28) query from pg_stat_activity order by 1;
  select pl.pid, pl.mode, pl.granted g, pl.relation::regclass, a.wait_event_type
wait, a.wait_event event, round(extract('epoch' from clock_timestamp() -
a.query_start)) sec, left(a.query,28) query from pg_locks pl left join
pg_stat_activity a ON pl.pid = a.pid where a.datname = current_database() order
by 1;
```

```
  pid   |      type       | state  | wait_type |     wait_event      | age1 | age2 | sec |            query
--------+-----------------+--------+-----------+---------------------+------+------+-----+-----------------------------
 699375 | checkpointer    |        | Activity  | CheckpointerMain    |      |      |     |
 699376 | background wri  |        | Activity  | BgWriterHibernate   |      |      |     |
 699378 | walwriter       |        | Activity  | WalWriterMain       |      |      |     |
 699379 | autovacuum lau  |        | Activity  | AutoVacuumMain      |      |      |     |
 699380 | logical replic  |        | Activity  | LogicalLauncherMain |      |      |     |
 762653 | client backend  | active |           |                     |      |    0 |   0 | select pid, left(backend_typ
 763886 | client backend  | idle   | Client    | ClientRead          |      |      |     | select count(*) from pgbench
(7 rows)

  pid   |      mode       | g |         relation          | wait | event | sec |            query
--------+-----------------+---+---------------------------+------+-------+-----+-----------------------------
 762653 | AccessShareLock | t | pg_stat_activity          |      |       |   0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock | t | pg_locks                  |      |       |   0 | select pl.pid, pl.mode, pl.g
 762653 | ExclusiveLock   | t |                           |      |       |   0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock | t | pg_authid_oid_index       |      |       |   0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock | t | pg_database_oid_index     |      |       |   0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock | t | pg_authid_rolname_index   |      |       |   0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock | t | pg_database_datname_index |      |       |   0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock | t | pg_authid                 |      |       |   0 | select pl.pid, pl.mode, pl.g
 762653 | AccessShareLock | t | pg_database               |      |       |   0 | select pl.pid, pl.mode, pl.g
(9 rows)
```

Серверный процесс **762650** завершился и его нет в списке процессов в операционной системе и в результатах запросов.

10) В первой сессии можно выполнить любой запрос и убедиться, что сессия завершена:

```
postgres=*# select 1;
FATAL:  terminating connection due to administrator command
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
The connection to the server was lost. Attempting reset: Succeeded.
postgres=#
```

Сессия подсоединилась к новому серверному процессу.

11) Можно закрыть ненужные сессии и терминалы.

Практика к главе 15

Часть 1. Установка расширения pg_stat_kcache

Статистики, собираемые экземпляром в стандартных представлениях pg_stat_io не позволяют различить находился бли блок в страничном кэше linux или читался с диска. Расширение pg_stat_kcache позволяет это различать. Расширение собирает статистику linux, выполняя системный вызов getrusage после выполнении каждой команды.

В отличие от утилит операционной системы, расширение собирает статистики с детальностью до команды sql.

Если расширение отсутствует, его нужно установить. Расширение написано на языке C и включает в себя разделяемую библиотеку. Расширение не требует изменения ядра PostgreSQL.

Расширение `pg_stat_kcache` стандартно поставляется с СУБД Tantor SE начиная с версии 16.6.

1) Переключитесь в терминал пользователя операционной системы root:

```
postgres@tantor:~$ su -
Password: root
```

2) Установите компилятор языка с версии 13:

```
root@tantor:~# apt install clang-13
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
clang-13 is already the newest version (1:13.0.1-11+b1).
The following packages were automatically installed and are no longer required:
  libintl-perl libintl-xs-perl libmodule-find-perl libmodule-scandeps-perl libproc-processtable-
perl libsort-naturally-perl libterm-readkey-perl
Use 'apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 607 not upgraded.
```

Компилятор был установлен.

3) Скачайте исходный код расширения:

```
root@tantor:~# wget https://github.com/powa-
team/pg_stat_kcache/archive/REL2_3_0.tar.gz
HTTP request sent, awaiting response... 302 Found
Location: https://codeload.github.com/powa-team/pg_stat_kcache/tar.gz/refs/tags/REL2_3_0
[following]
Length: unspecified [application/x-gzip]
Saving to: 'REL2_3_0.tar.gz.1'
'REL2_3_0.tar.gz.1' saved [24477]
```

4) Разархивируйте файл REL2_3_0.tar.gz:

```
root@tantor:~# tar xzf ./REL2_3_0.tar.gz
```

5) Перейдите в созданную при распаковке архива директорию:

```
root@tantor:~# cd pg_stat_kcache-REL2_3_0
```

6) Соберите расширение:

```
root@tantor:~/pg_stat_kcache-REL2_3_0# make
gcc -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -Werror=vla -Wendif-
labels -Wmissing-format-attribute -Wimplicit-fallthrough=3 -Wcast-function-type -
Wshadow=compatible-local -Wformat-security -fno-strict-aliasing -fwrapv -fexcess-precision=standard
-Wno-format-truncation -Wno-stringop-truncation -O2 -pipe -Wno-missing-braces -DTANTOR_SE -fPIC -
fvisibility=hidden -I. -I./ -I/opt/tantor/db/16/include/postgresql/server -
I/opt/tantor/db/16/include/postgresql/internal  -O2 -pipe -Wno-missing-braces -DTANTOR_SE -
```

```
D_GNU_SOURCE -I/usr/include/libxml2 -I/usr/local/include/zstd   -c -o pg_stat_kcache.o
pg_stat_kcache.c
   gcc -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -Werror=vla -Wendif-
labels -Wmissing-format-attribute -Wimplicit-fallthrough=3 -Wcast-function-type -
Wshadow=compatible-local -Wformat-security -fno-strict-aliasing -fwrapv -fexcess-precision=standard
-Wno-format-truncation -Wno-stringop-truncation -O2 -pipe -Wno-missing-braces -DTANTOR_SE -fPIC -
fvisibility=hidden -shared -o pg_stat_kcache.so pg_stat_kcache.o -L/opt/tantor/db/16/lib   -
L/usr/lib/llvm-13/lib -L/usr/local/lib/zstd  -Wl,--as-needed -Wl,-rpath,'/opt/tantor/db/16/lib',--
enable-new-dtags -fvisibility=hidden
   /usr/bin/clang-13 -Wno-ignored-attributes -fno-strict-aliasing -fwrapv -Wno-unused-command-line-
argument -Wno-compound-token-split-by-macro -O2  -I. -I./ -
I/opt/tantor/db/16/include/postgresql/server -I/opt/tantor/db/16/include/postgresql/internal  -O2 -
pipe -Wno-missing-braces -DTANTOR_SE -D_GNU_SOURCE -I/usr/include/libxml2 -I/usr/local/include/zstd
-flto=thin -emit-llvm -c -o pg_stat_kcache.bc pg_stat_kcache.c
```

7) **На версии СУБД Tantor SE 16.6 и более новых этот пункт выполнять не нужно, так как библиотека расширения уже установлена.**

Установите файлы расширения:

```
   root@tantor:~/pg_stat_kcache-REL2_3_0# make install
   /usr/bin/mkdir -p '/opt/tantor/db/16/lib/postgresql'
   /usr/bin/mkdir -p '/opt/tantor/db/16/share/postgresql/extension'
   /usr/bin/mkdir -p '/opt/tantor/db/16/share/postgresql/extension'
   /usr/bin/install -c -m 755  pg_stat_kcache.so
'/opt/tantor/db/16/lib/postgresql/pg_stat_kcache.so'
   /usr/bin/install -c -m 644 .//pg_stat_kcache.control
'/opt/tantor/db/16/share/postgresql/extension/'
   /usr/bin/install -c -m 644 .//pg_stat_kcache--2.1.0--2.1.1.sql .//pg_stat_kcache--2.1.0.sql
.//pg_stat_kcache--2.1.1--2.1.2.sql .//pg_stat_kcache--2.1.1.sql .//pg_stat_kcache--2.1.2--
2.1.3.sql .//pg_stat_kcache--2.1.2.sql .//pg_stat_kcache--2.1.3--2.2.0.sql .//pg_stat_kcache--
2.1.3.sql .//pg_stat_kcache--2.2.0--2.2.1.sql .//pg_stat_kcache--2.2.0.sql .//pg_stat_kcache--
2.2.1--2.2.2.sql .//pg_stat_kcache--2.2.1.sql .//pg_stat_kcache--2.2.2--2.2.3.sql
.//pg_stat_kcache--2.2.2.sql .//pg_stat_kcache--2.2.3--2.3.0.sql .//pg_stat_kcache--2.2.3.sql
.//pg_stat_kcache--2.3.0.sql  '/opt/tantor/db/16/share/postgresql/extension/'
   /usr/bin/mkdir -p '/opt/tantor/db/16/lib/postgresql/bitcode/pg_stat_kcache'
   /usr/bin/mkdir -p '/opt/tantor/db/16/lib/postgresql/bitcode'/pg_stat_kcache/
   /usr/bin/install -c -m 644 pg_stat_kcache.bc
'/opt/tantor/db/16/lib/postgresql/bitcode'/pg_stat_kcache/./
   cd '/opt/tantor/db/16/lib/postgresql/bitcode' && /usr/lib/llvm-13/bin/llvm-lto -thinlto -thinlto-
action=thinlink -o pg_stat_kcache.index.bc pg_stat_kcache/pg_stat_kcache.bc
```

8) Вернитесь или откройте терминал пользователя операционной системы postgres:

```
   root@tantor:~/pg_stat_kcache-REL2_3_0#  <ctrl+d>
   logout
```

9) Подключите библиотеки расширений, которые будут использоваться дальше в практиках:

```
   postgres@tantor:~$ psql -c "alter system set shared_preload_libraries =
pg_stat_statements, pg_wait_sampling, pg_stat_kcache;"
   ALTER SYSTEM
```

10) Перезапустите экземпляр:

```
   postgres@tantor:~$ sudo restart
```

11) Запустите psql и установите расширение:

```
   postgres@tantor:~$ psql

   postgres=# create extension pg_stat_kcache;
   ERROR:  required extension "pg_stat_statements" is not installed
   HINT:  Use CREATE EXTENSION ... CASCADE to install required extensions too.
   postgres=# create extension pg_stat_kcache cascade;
   NOTICE:  installing required extension "pg_stat_statements"
   CREATE EXTENSION
```

Расширение kcache использует расширение pg_stat_statements. Его можно было установить отдельной командой, но использовалась опция cascade для установки.

12) Посмотрите, какие объекты включены в расширения:

```
postgres=# \dx+ pg_stat_statements
         Objects in extension "pg_stat_statements"
                    Object description
-------------------------------------------------------------
 function pg_stat_statements(boolean)
 function pg_stat_statements_info()
 function pg_stat_statements_reset(oid,oid,bigint,boolean)
 view pg_stat_statements
 view pg_stat_statements_info
(5 rows)

postgres=# \dx+ pg_stat_kcache
Objects in extension "pg_stat_kcache"
       Object description
--------------------------------
 function pg_stat_kcache()
 function pg_stat_kcache_reset()
 view pg_stat_kcache
 view pg_stat_kcache_detail
(4 rows)
```

В расширениях есть представления и функции.

Представление `pg_stat_kcache_detail` имеет столбцы: `query, top, rolname` и выдает данные с точностью до команды. Статистики выдаются из 14 столбцов для планирования и 14 столбцов для выполнения команд.

Представление pg_stat_kcache содержит суммарные статистики из pg_stat_kcache_detail, сгруппированные по базам данных:

```
CREATE VIEW pg_stat_kcache AS SELECT datname, SUM(столбцы) FROM
pg_stat_kcache_detail WHERE top IS TRUE GROUP BY datname;
```

Статистики в обоих представлениях:

```
exec_reads       reads, in bytes
exec_writes      writes, in bytes
exec_reads_blks  reads, in 8K-blocks
exec_writes_blks writes, in 8K-blocks
exec_user_time   user CPU time used
exec_system_time system CPU time used
exec_minflts     page reclaims (soft page faults)
exec_majflts     page faults (hard page faults)
exec_nswaps      swaps
exec_msgsnds     IPC messages sent
exec_msgrcvs     IPC messages received
exec_nsignals    signals received
exec_nvcsws      voluntary context switches
exec_nivcsws     involuntary context switches
```

Данные соответствуют структуре rusage, её содержимое можно посмотреть командой `man getrusage`.

Если `exec_majflts` незначительно по сравнению с `exec_minflts`, это означает, что оперативной памяти достаточно. Кроме этого утверждения статистика `exec_majflts` бессмысленна.

Пропорция exec_user_time и exec_system_time позволяет определить нет ли перекоса в сторону кода ядра или кода PostgreSQL и функций расширений и приложения.

Наличие exec_nivcsws (принудительных переключений контекста выполнения планировщиком операционной системы) указывает на то, что запрос активно нагружал процессор и процессор был узким местом.

13) Посмотрите какие у расширения есть параметры:

```
postgres=# \dconfig pg_stat_k*
    List of configuration parameters
           Parameter           | Value
-------------------------------+--------
 pg_stat_kcache.linux_hz       | 250000
 pg_stat_kcache.track          | top
 pg_stat_kcache.track_planning | off
(3 rows)
```

pg_stat_kcache.linux_hz (по умолчанию -1) устанавливается автоматически в значение параметра linux CONFIG_HZ и используется для компенсации ошибок сэмплирования. Менять не нужно.

pg_stat_kcache.track (по умолчанию top) три знчаения:

a) top - отслеживать команды, посланные на выполнение клиентом

b) all - отслеживать вложенные запросы. Например, запросы в вызываемых функциях

c) none - отключает сбор статистик pg_stat_kcache

pg_stat_kcache.track_planning (по умолчанию off) нужно ли собирать статистику для этапа планирования

14) Расширения могут выделять структуры в разделяемой памяти для накопления данных. Посмотрите, какие структуры памяти (буфера) были выделены расширениями:

```
postgres=# select * from (select *,lead(off) over(order by off)-off as true
from pg_shmem_allocations) as a where name like 'pg_%' order by 1;
            name             |    off    | size  | allocated_size |  true
-----------------------------+-----------+-------+----------------+---------
 pg_qualqueryexamples_hash   | 152021248 |  2896 |           2944 | 1067136
 pg_qualstatements_hash      | 151746048 |  2896 |           2944 |  275200
 pg_qualstats                | 151741696 |   147 |            256 |    4352
 pg_stat_kcache              | 150352128 |   992 |           1024 |    1024
 pg_stat_kcache hash         | 150353152 |  2896 |           2944 | 1388544
 pg_stat_statements          | 148163456 |    64 |            128 |     128
 pg_stat_statements hash     | 148163584 |  2896 |           2944 | 2188544
 pg_store_plans              | 153088384 |    72 |            128 |     128
 pg_store_plans hash         | 153088512 |  2896 |           2944 |  251136
 pg_wait_sampling            | 148145920 | 17536 |          17536 |   17536
(10 rows)
```

Расширение pg_stat_kcache использует два буфера в разделяемой памяти.

Число команд, по которым собирается статистика и размер структур разделяемой памяти определяется параметром pg_stat_statements.max (по умолчанию 5000), так как расширение pg_stat_kcache зависит от расширения pg_stat_statements.

15) Закройте psql и выполните команды:

```
postgres=# \q
postgres@tantor:~$
psql -c "alter system set pg_stat_statements.max=50000;"
sudo restart
```

```
  psql -c "select * from (select *,lead(off) over(order by off)-off as true from
pg_shmem_allocations) as a where name like 'pg_%' order by 1;"
  psql -c "alter system reset pg_stat_statements.max;"
  sudo restart
  psql

  ALTER SYSTEM
            name              |    off     | size  | allocated_size |   true
  ----------------------------+------------+-------+----------------+----------
   pg_qualqueryexamples_hash | 183898624 |  2896 |           2944 |  1067136
   pg_qualstatements_hash    | 183623424 |  2896 |           2944 |   275200
   pg_qualstats              | 183619072 |   147 |            256 |     4352
   pg_stat_kcache            | 169890816 |   992 |           1024 |     1024
   pg_stat_kcache hash       | 169891840 |  2896 |           2944 | 13727232
   pg_stat_statements        | 148163456 |    64 |            128 |      128
   pg_stat_statements hash   | 148163584 |  2896 |           2944 | 21727232
   pg_store_plans            | 184965760 |    72 |            128 |      128
   pg_store_plans hash       | 184965888 |  2896 |           2944 |   251136
   pg_wait_sampling          | 148145920 | 17536 |          17536 |    17536
  (10 rows)

  ALTER SYSTEM

  postgres=#
```

При увеличении числа команд, по которым собирается статистика в 10 раз с 5000 до 50000 параметром **pg_stat_statements.max**, размер буферов `pg_stat_kcache hash` и `pg_stat_statements hash` увеличился в ~10 раз.

Часть 2. Использование расширения pg_stat_kcache

1) Пример запроса к статистикам, собираемым расширением:

```
  postgres=# select d.datname database, round(s.total_exec_time::numeric, 0)
time, s.calls, pg_size_pretty(exec_minflts*4096) recl,
pg_size_pretty(exec_majflts*4096) faults, pg_size_pretty(k.exec_reads) reads,
pg_size_pretty(k.exec_writes) writes, round(k.exec_user_time::numeric, 2) user,
round(k.exec_system_time::numeric, 2) sys, k.exec_nvcsws vsw, k.exec_nivcsws isw,
left(s.query, 18) query from pg_stat_statements s join pg_stat_kcache() k using
(userid, dbid, queryid) join pg_database d on s.dbid = d.oid order by
total_exec_time desc limit 5;

 database | time | calls |   recl |faults|  reads   | writes   | user | sys  | vsw  | isw |      query
----------+------+-------+--------+------+----------+----------+------+------+------+-----+-----------------
 postgres | 1020 | 19988 |   4323 |    0 | 15 MB    | 0 bytes | 0.48 | 0.62 | 1281 |  25 | UPDATE pgbench_
 postgres |  488 | 19988 |      7 |    0 | 0 bytes | 0 bytes | 0.38 | 0.47 |    0 |  21 | UPDATE pgbench_
 postgres |  459 | 19988 |      9 |    0 | 0 bytes | 0 bytes | 0.37 | 0.47 |    0 |  19 | UPDATE pgbench_
 postgres |  369 | 19988 |      0 |    0 | 0 bytes | 0 bytes | 0.34 | 0.41 |    0 |  17 | SELECT abalance
 postgres |  306 | 19988 |    272 |    0 | 0 bytes | 1056 kB | 0.29 | 0.38 |    1 |  13 | INSERT INTO pgb
(5 rows)
```

2) Сбросьте статистику:

```
  postgres=#
  select pg_stat_statements_reset();
  select pg_stat_kcache_reset();
     pg_stat_statements_reset
  -------------------------------
   2035-01-01 01:01:01.101508+03
  (1 row)

   pg_stat_kcache_reset
  ----------------------
```

```
(1 row)
```

Так как запрос соединял данные двух расширений, то нужно сбрасывать статистику обоих расширений.

3) Повторите запрос:

```
postgres=# select d.datname database, round(s.total_exec_time::numeric, 0)
time, s.calls, pg_size_pretty(exec_minflts*4096) recl,
pg_size_pretty(exec_majflts*4096) faults, pg_size_pretty(k.exec_reads) reads,
pg_size_pretty(k.exec_writes) writes, round(k.exec_user_time::numeric, 2) user,
round(k.exec_system_time::numeric, 2) sys, k.exec_nvcsws vsw, k.exec_nivcsws isw,
left(s.query, 18) query from pg_stat_statements s join pg_stat_kcache() k using
(userid, dbid, queryid) join pg_database d on s.dbid = d.oid order by
total_exec_time desc limit 5;
 database | time | calls | faults | recl |  reads  | writes  | user | sys  | vsw | isw |       query
----------+------+-------+--------+------+---------+---------+------+------+-----+-----+-----------------
 postgres |    0 |     1 |      1 |    0 | 0 bytes | 0 bytes | 0.00 | 0.00 |   0 |   0 | select pg_stat_
(1 row)
```

Статистика была сброшена.

4) Выполните тест `initins.sql`, который включает в себя создание 10 таблиц размером по 422Мб каждая:

```
psql -c "select pg_stat_reset_shared('io');"
psql -c "select pg_stat_reset_shared('bgwriter');"
psql -f initcre.sql 2> /dev/null > /dev/null
time psql -f initins.sql > /dev/null
 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)

real    4m0.245s
user    0m0.010s
sys     0m0.000s
```

Добавление библиотек параметром **shared_preload_libraries** на время выполнения скрипта не повлияло: в пункте 7 части 6 практики 14 было `3m58`. Небольшое увеличение времени выполнения из-за того, что параметр `track_io_timing=on`.

5) Пока команды работают во втором окне выполните команды:

```
postgres=# select d.datname database, round(s.total_exec_time::numeric, 0) time,
s.calls, pg_size_pretty(exec_minflts*4096) recl,
pg_size_pretty(exec_majflts*4096) faults, pg_size_pretty(k.exec_reads) reads,
pg_size_pretty(k.exec_writes) writes, round(k.exec_user_time::numeric, 2) user,
round(k.exec_system_time::numeric, 2) sys, k.exec_nvcsws vsw, k.exec_nivcsws isw,
left(s.query, 18) query from pg_stat_statements s join pg_stat_kcache() k using
(userid, dbid, queryid) join pg_database d on s.dbid = d.oid order by
total_exec_time desc;
\watch 3
```

```
 database | time  | calls |   recl |faults|  reads  | writes  | user  | sys  | vsw | isw |       query
----------+-------+-------+--------+------+---------+---------+-------+------+-----+-----+--------------------
 postgres | 24959 |     1 |    938 |    0 | 0 bytes | 1402 MB | 11.50 | 3.66 | 276 | 265 | insert into test.f
 postgres | 24891 |     1 |    936 |    0 | 0 bytes | 1397 MB | 11.96 | 3.28 | 244 | 254 | insert into test.f
 postgres | 24271 |     1 |    945 |    0 | 0 bytes | 1295 MB | 12.04 | 3.25 | 234 | 277 | insert into test.f
 postgres | 24265 |     1 |    956 |    0 | 0 bytes | 1260 MB | 12.01 | 3.24 | 257 | 239 | insert into test.f
 postgres | 23717 |     1 |    934 |    0 | 0 bytes | 1305 MB | 11.94 | 3.35 | 231 | 241 | insert into test.f
```

```
 postgres | 22869 |     1 |   1205 |    0 | 0 bytes | 1317 MB | 12.46 | 2.70 | 337 | 269 | insert into test.f
 postgres | 19388 |     1 |  37420 |   79 | 3544 kB | 1253 MB | 12.34 | 2.82 | 276 | 191 | insert into test.f
 postgres |     5 |    16 |    222 |    0 | 0 bytes | 32 kB   |  0.01 | 0.00 |   0 |   0 | select d.datname d
 postgres |     0 |     1 |     75 |    0 | 0 bytes | 0 bytes |  0.00 | 0.00 |   0 |   0 | select d.datname d
 postgres |     0 |     2 |     12 |    0 | 0 bytes | 0 bytes |  0.00 | 0.00 |   0 |   0 | select pg_stat_res
 postgres |     0 |     2 |      8 |    0 | 0 bytes | 0 bytes |  0.00 | 0.00 |   0 |   0 | select format($1,
 postgres |     0 |     1 |      1 |    0 | 0 bytes | 0 bytes |  0.00 | 0.00 |   0 |   0 | select pg_stat_kca
(12 rows)
```

Строки с командами `insert` будут появляться по окончании этих команд.

6) После окончания выполнения скрипта посмотрите общую статистику запросом:

```
postgres=# select datname database, pg_size_pretty(exec_minflts*4096) recl,
pg_size_pretty(exec_majflts*4096) faults, pg_size_pretty(exec_reads) reads,
pg_size_pretty(exec_writes) writes, round(exec_system_time::numeric,0) sys,
round(exec_user_time::numeric,0) usr,  exec_nvcsws vsw, exec_nivcsws isw from
pg_stat_kcache;
 database |   recl |faults|  reads   | writes | sys | usr | vsw  | isw
----------+--------+------+---------+--------+-----+-----+------+-----
 postgres |  46586 |   79 | 3544 kB | 13 GB  |  33 | 119 | 2624 | 2737
(1 row)
```

7) Выполните команду:

```
postgres@tantor:~$ time psql -f initsel.sql > /dev/null

real    1m13.586s
user    0m0.012s
sys     0m0.000s
```

Тест выполнялся долго, потому что первый select по таблицам грязнил блоки.

8) во втором остановите `\watch 1` комбинацией клавиш `<ctrl+c>` и выполните команды:

```
postgres=# select d.datname database, round(s.total_exec_time::numeric, 0) time,
s.calls, pg_size_pretty(exec_minflts*4096) recl,
pg_size_pretty(exec_majflts*4096) faults, pg_size_pretty(k.exec_reads) reads,
pg_size_pretty(k.exec_writes) writes, round(k.exec_user_time::numeric, 2) user,
round(k.exec_system_time::numeric, 2) sys, k.exec_nvcsws vsw, k.exec_nivcsws isw,
left(s.query, 18) query from pg_stat_statements s join pg_stat_kcache() k using
(userid, dbid, queryid) join pg_database d on s.dbid = d.oid where query like
'select%' order by 12 desc limit 11;
\watch 3
```

```
 database | time  | calls |   recl |faults|  reads   | writes | user | sys  |  vsw  | isw |       query
----------+-------+-------+--------+------+---------+--------+------+------+-------+-----+--------------------
 postgres | 10366 |     1 |   7935 |    0 | 422 MB  | 731 MB | 2.70 | 2.05 | 11494 | 688 | select * from test
 postgres | 10122 |     1 |   7428 |    0 | 422 MB  | 743 MB | 2.83 | 1.90 | 11251 | 412 | select * from test
 postgres |  9087 |     1 |   7218 |    0 | 424 MB  | 744 MB | 2.84 | 1.88 | 11312 | 227 | select * from test
 postgres |  7779 |     1 |   7645 |    0 | 422 MB  | 778 MB | 2.97 | 2.05 | 11814 | 539 | select * from test
 postgres |  7758 |     1 |   7383 |    0 | 422 MB  | 752 MB | 2.93 | 1.73 | 12135 | 157 | select * from test
 postgres |  7121 |     1 |   7387 |    0 | 425 MB  | 772 MB | 2.93 | 1.99 | 12238 | 185 | select * from test
 postgres |  7112 |     1 |   7000 |    0 | 422 MB  | 770 MB | 2.84 | 2.02 | 11049 | 465 | select * from test
 postgres |  6954 |     1 |   8293 |    0 | 422 MB  | 749 MB | 2.96 | 1.96 | 10747 | 352 | select * from test
 postgres |  5870 |     1 |   7640 |    0 | 426 MB  | 761 MB | 2.96 | 1.98 | 11434 | 404 | select * from test
 postgres |  4765 |     1 |  38121 |    1 | 422 MB  | 675 MB | 2.75 | 1.95 | 12040 | 131 | select * from test
 postgres |     7 |    20 |    260 |    0 | 0 bytes | 32 kB  | 0.01 | 0.00 |     0 |   0 | select d.datname d
(11 rows)
```

9) Несколько раз повторите команды:

```
postgres@tantor:~$
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null
```

```
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null

real    0m17.399s
user    0m0.010s
sys     0m0.000s

real    0m18.726s
user    0m0.005s
sys     0m0.005s

real    0m13.751s
user    0m0.013s
sys     0m0.000s

real    0m15.422s
user    0m0.005s
sys     0m0.005s

real    0m15.717s
user    0m0.012s
sys     0m0.000s
```

Команды select в тесте не грязнили блоки.
Первые команды выполнялись ~18 секунд - заполнялся кэш linux, последующие ~15 секунд - кэш linux был заполнен.

Посмотрите какие значения увеличиваются при выполнении теста **initsel.sql**:

```
 database | time  | calls |  recl  |faults|  reads   | writes   | user | sys  |  vsw  | isw  |       query
----------+-------+-------+--------+------+----------+----------+------+------+-------+------+--------------------
 postgres | 14786 |     6 |  14402 |    0 | 1957 MB  | 749 MB   | 8.69 | 8.99 | 36143 | 1209 | select * from test
 postgres | 18002 |     6 |  13533 |    0 | 2075 MB  | 743 MB   | 8.49 | 8.81 | 38478 | 1176 | select * from test
 postgres | 18184 |     6 |  13969 |    0 | 2062 MB  | 731 MB   | 8.38 | 8.95 | 38795 | 1139 | select * from test
 postgres | 15365 |     6 |  13685 |    0 | 1931 MB  | 778 MB   | 8.38 | 8.97 | 36917 | 1120 | select * from test
 postgres | 13465 |     6 |  13667 |    0 | 1935 MB  | 761 MB   | 8.33 | 9.11 | 36669 | 1014 | select * from test
 postgres | 14994 |     6 |  13085 |    0 | 2078 MB  | 770 MB   | 8.31 | 9.03 | 38189 |  881 | select * from test
 postgres | 16188 |     6 |  13316 |    0 | 1744 MB  | 744 MB   | 8.38 | 8.74 | 33297 |  857 | select * from test
 postgres | 14761 |     6 |  13462 |    0 | 1962 MB  | 772 MB   | 8.67 | 8.71 | 37739 |  732 | select * from test
 postgres | 14318 |     6 |  34851 |    0 | 1672 MB  | 752 MB   | 8.99 | 6.99 | 32540 |  719 | select * from test
 postgres | 11515 |     6 |  59070 |    1 | 1592 MB  | 675 MB   | 8.42 | 8.56 | 31446 |  657 | select * from test
 postgres |     0 |     8 |     49 |    0 | 0 bytes  | 0 bytes  | 0.00 | 0.00 |     0 |    0 | select format($1,
 postgres |     7 |    20 |    260 |    0 | 0 bytes  | 32 kB    | 0.01 | 0.00 |     0 |    0 | select d.datname d
(12 rows)
```

Увеличиваются значения во всех числовых столбцах, кроме `faults` и `writes`.
В столбце `calls` увеличилось число выполнений запросов.

10) Сбросьте статистику и повторно выполните один тест:

```
postgres=#
psql -c "select pg_stat_statements_reset();"
psql -c "select pg_stat_kcache_reset();"
psql -c "select pg_stat_reset_shared('io');"
psql -c "select pg_stat_reset_shared('bgwriter');"
time psql -f initsel.sql > /dev/null
...
real    0m18.596s
user    0m0.010s
sys     0m0.002s
```

11) Посмотрите статистику по однократным запросам:

```
 database | time | calls |  recl  |faults|  reads   | writes   | user | sys  | vsw  | isw |       query
----------+------+-------+--------+------+----------+----------+------+------+------+-----+--------------------
 postgres | 1732 |     1 |   1453 |    0 | 366 MB   | 0 bytes  | 1.15 | 1.41 | 5966 | 291 | select * from test
 postgres | 1690 |     1 |   1473 |    0 | 376 MB   | 0 bytes  | 1.18 | 1.37 | 6138 | 125 | select * from test
 postgres |  792 |     1 |   2473 |    0 | 21 MB    | 0 bytes  | 1.10 | 1.07 |  372 | 122 | select * from test
 postgres | 1754 |     1 |   1426 |    0 | 391 MB   | 0 bytes  | 1.13 | 1.50 | 6404 |  91 | select * from test
 postgres | 1704 |     1 |   4431 |    0 | 375 MB   | 0 bytes  | 1.25 | 1.36 | 6111 |  84 | select * from test
 postgres | 1175 |     1 |   1432 |    0 | 171 MB   | 0 bytes  | 1.11 | 1.21 | 2892 |  62 | select * from test
 postgres | 1761 |     1 |   1442 |    0 | 410 MB   | 0 bytes  | 1.03 | 1.46 | 6782 |  62 | select * from test
 postgres | 1744 |     1 |   1449 |    0 | 377 MB   | 0 bytes  | 1.16 | 1.48 | 6181 |  51 | select * from test
```

```
 postgres | 1718 |      1 |    1429 |     0 | 395 MB  | 0 bytes | 1.15 | 1.41 | 6430 |   50 | select * from test
 postgres | 1012 |      1 |    1451 |     0 | 115 MB  | 0 bytes | 1.14 | 1.10 | 2050 |   35 | select * from test
 postgres |    1 |      4 |       0 |     0 | 0 bytes | 0 bytes | 0.00 | 0.00 |    0 |    1 | select d.datname d
 postgres |    0 |      1 |      13 |     0 | 0 bytes | 0 bytes | 0.00 | 0.00 |    0 |    0 | select pg_stat_kca
(12 rows)
```

Все чтения выполнялись из кэша linux.

При чтении одной таблицы из кэша linux читалось ~1400 блоков. У двух таблиц в несколько раз больше.

Размер каждой таблицы 422Мб:

```
postgres=# \dt+ test.*
                                    List of relations
 Schema |      Name       | Type  |  Owner   | Persistence | Access method |  Size  | Description
--------+-----------------+-------+----------+-------------+---------------+--------+-------------
 test   | film_summary1   | table | postgres | permanent   | heap          | 422 MB |
 test   | film_summary10  | table | postgres | permanent   | heap          | 422 MB |
 test   | film_summary2   | table | postgres | permanent   | heap          | 422 MB |
 test   | film_summary3   | table | postgres | permanent   | heap          | 422 MB |
 test   | film_summary4   | table | postgres | permanent   | heap          | 422 MB |
 test   | film_summary5   | table | postgres | permanent   | heap          | 422 MB |
 test   | film_summary6   | table | postgres | permanent   | heap          | 422 MB |
 test   | film_summary7   | table | postgres | permanent   | heap          | 422 MB |
 test   | film_summary8   | table | postgres | permanent   | heap          | 422 MB |
 test   | film_summary9   | table | postgres | permanent   | heap          | 422 MB |
(10 rows)
```

12) Размер буферного кэша 128Мб:

```
postgres=# show shared_buffers;
 shared_buffers
----------------
 128MB
(1 row)
```

13) При выполнении select использовались буферные кольца и буфера в кольцах повторно использовались под блоки таблиц:

```
postgres=# select backend_type, context, writes w, round(write_time::numeric,2)
wt, writebacks wb, round(writeback_time::numeric,2) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
     backend_type    | context  | w |  wt  | wb |  wbt  | ex | et | hits | ev |   ru   | fs | fst
---------------------+----------+---+------+----+-------+----+----+------+----+--------+----+-----
 autovacuum worker   | normal   | 0 | 0.00 |  0 |  0.00 |  0 |  0 | 2453 |  0 |        |  0 |   0
 client backend      | bulkread | 0 | 0.00 |  0 |  0.00 |    |    | 4817 | 320 | 179947 |    |
 client backend      | normal   | 0 | 0.00 |  0 |  0.00 |  0 |  0 |  463 |  0 |        |  0 |   0
 background worker   | bulkread | 0 | 0.00 |  0 |  0.00 |    |    | 9680 | 640 | 345146 |    |
 background worker   | normal   | 0 | 0.00 |  0 |  0.00 |  0 |  0 | 3240 |  0 |        |  0 |   0
(5 rows)
```

14) Проверьте, как повлияет на показатели увеличение размера буферного кэша до 2Гб. Дальше пул буферов будет увеличен и тест повторён. Выполните команды:

```
postgres@tantor:~$
psql -c "alter system set shared_buffers='2GB'"
psql -c "select pg_stat_statements_reset();"
psql -c "select pg_stat_kcache_reset();"
psql -c "select pg_stat_reset_shared('io');"
psql -c "select pg_stat_reset_shared('bgwriter');"
sudo restart
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null
```

```
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null
...
real    0m18.908s
user    0m0.010s
sys     0m0.000s

real    0m19.400s
user    0m0.009s
sys     0m0.001s

real    0m19.122s
user    0m0.009s
sys     0m0.004s

real    0m19.556s
user    0m0.002s
sys     0m0.012s

real    0m19.197s
user    0m0.014s
sys     0m0.000s
```

15) Статистики после выполнения команд:

```
postgres=# select d.datname database, round(s.total_exec_time::numeric, 0) time,
s.calls, pg_size_pretty(exec_minflts*4096) recl,
pg_size_pretty(exec_majflts*4096) faults, pg_size_pretty(k.exec_reads) reads,
pg_size_pretty(k.exec_writes) writes, round(k.exec_user_time::numeric, 2) user,
round(k.exec_system_time::numeric, 2) sys, k.exec_nvcsws vsw, k.exec_nivcsws isw,
left(s.query, 18) query from pg_stat_statements s join pg_stat_kcache() k using
(userid, dbid, queryid) join pg_database d on s.dbid = d.oid where query like
'select%' order by 12 desc limit 11;

select backend_type, context, writes w, round(write_time::numeric,2) wt,
writebacks wb, round(writeback_time::numeric,2) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
 database | time | calls |   recl |faults|  reads   | writes   | user | sys  |  vsw  | isw |       query
----------+------+-------+--------+------+----------+----------+------+------+-------+-----+--------------------
 postgres |    0 |     2 |     10 |    0 | 0 bytes  | 0 bytes  | 0.00 | 0.00 |     0 |   0 | select pg_stat_res
 postgres |    0 |     1 |     13 |    0 | 0 bytes  | 0 bytes  | 0.00 | 0.00 |     0 |   0 | select pg_stat_kca
 postgres | 9140 |     5 | 486570 |    0 | 2097 MB  | 0 bytes  | 6.27 | 7.53 | 34311 | 249 | select * from test
 postgres | 8991 |     5 | 550942 |    0 | 1949 MB  | 0 bytes  | 6.38 | 7.49 | 32074 | 325 | select * from test
 postgres | 9219 |     5 | 541808 |    0 | 2095 MB  | 0 bytes  | 6.48 | 7.38 | 34087 | 238 | select * from test
 postgres | 9206 |     5 | 551916 |    0 | 2022 MB  | 0 bytes  | 6.45 | 7.21 | 33507 | 206 | select * from test
 postgres | 9175 |     5 | 474780 |    0 | 2116 MB  | 0 bytes  | 6.20 | 7.38 | 35075 | 266 | select * from test
 postgres | 9178 |     5 | 487968 |    0 | 2095 MB  | 0 bytes  | 6.35 | 7.34 | 34234 | 211 | select * from test
 postgres | 9241 |     5 | 485323 |    0 | 2111 MB  | 0 bytes  | 6.18 | 7.48 | 34724 | 251 | select * from test
 postgres | 8598 |     5 | 550936 |    0 | 1828 MB  | 0 bytes  | 6.41 | 7.26 | 30014 | 320 | select * from test
 postgres | 8395 |     5 | 566218 |    0 | 1708 MB  | 0 bytes  | 6.36 | 7.33 | 28196 | 499 | select * from test
(11 rows)

    backend_type     | context | w |  wt  | wb | wbt  | ex | et | hits  |   ev    | ru | fs | fst
---------------------+---------+---+------+----+------+----+----+-------+---------+----+----+-----
 autovacuum launcher | normal  | 2 | 0.07 |  0 | 0.00 |    |    |    61 |       5 |    |  0 |   0
 autovacuum worker   | normal  | 0 | 0.00 |  0 | 0.00 |  0 |  0 |  5775 |    1584 |    |  0 |   0
 client backend      | normal  | 0 | 0.00 |  0 | 0.00 |  0 |  0 |  3913 |  832895 |    |  0 |   0
 background worker   | normal  | 0 | 0.00 |  0 | 0.00 |  0 |  0 | 16191 | 1608110 |    |  0 |   0
(4 rows)
```

Буферные кольца не использовались. Суммарный размер таблиц 4.4Гб, что превышает размер буферного кэша. Увеличивалось вытеснение занятых блоков и их замена на другие блоки (**evictions**). Из-за увеличения памяти экземпляра на страничный кэш linux осталось мало памяти и его эффективность уменьшилась. Время выполнения теста увеличилось до ~19 секунд. Если сбросить кэш linux результат будет тем же:

```
root@tantor:~# echo 1 > /proc/sys/vm/drop_caches
real    0m19.380s
real    0m18.777s
```

Число пяти чтений каждой таблицы ~550942 непропорционально увеличилось, по сравнению со значениями из 9 пункта 14402.

При использовании кэша буферов размером 128Мб время выполнения теста было ~15 секунд.

Статистика **exec_minflts** не показывает чтение страниц с диска, которые отсутствуют в страничном кэше.

Часть 3. Производительность при использовании direct i/o

1) Включите прямой ввод-вывод (direct i/o), выполните тест и посмотрите результат теста:

```
postgres@tantor:~$
psql -c "alter system set debug_io_direct = data;"
psql -c "alter system reset shared_buffers;"
psql -c "select pg_stat_statements_reset();"
psql -c "select pg_stat_kcache_reset();"
psql -c "select pg_stat_reset_shared('io');"
psql -c "select pg_stat_reset_shared('bgwriter');"
sudo restart
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null
psql -c "select d.datname database, round(s.total_exec_time::numeric, 0) time, s.calls, pg_size_pretty(exec_minflts*4096) recl, pg_size_pretty(exec_majflts*4096) faults, pg_size_pretty(k.exec_reads) reads, pg_size_pretty(k.exec_writes) writes, round(k.exec_user_time::numeric, 2) user, round(k.exec_system_time::numeric, 2) sys, k.exec_nvcsws vsw, k.exec_nivcsws isw, left(s.query, 18) query from pg_stat_statements s join pg_stat_kcache() k using (userid, dbid, queryid) join pg_database d on s.dbid = d.oid where query like 'select%' order by 12 desc limit 11;"
psql -c "select backend_type, context, writes w, round(write_time::numeric,2) wt, writebacks wb, round(writeback_time::numeric,2) wbt, extends ex, round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs, round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or evictions>0 or reuses>0 or hits>0;"
psql -c "alter system reset debug_io_direct;"
sudo restart
```

```
real	0m56.905s
user	0m0.010s
sys	0m0.000s

real	0m56.558s
user	0m0.009s
sys	0m0.000s

 database | time  | calls |  recl  |faults|  reads  | writes  | user | sys  |  vsw   | isw |       query
----------+-------+-------+--------+------+---------+---------+------+------+--------+-----+--------------------
 postgres |     0 |     2 |     10 |    0 | 0 bytes | 0 bytes | 0.00 | 0.00 |      0 |   0 | select pg_stat_res
 postgres |     0 |     1 |     13 |    0 | 0 bytes | 0 bytes | 0.00 | 0.00 |      0 |   0 | select pg_stat_kca
 postgres | 11673 |     2 |   2316 |    0 | 844 MB  | 0 bytes | 2.47 | 5.66 | 108018 |  21 | select * from test
 postgres | 11648 |     2 |   2301 |    0 | 844 MB  | 0 bytes | 2.14 | 5.49 | 108023 |  32 | select * from test
 postgres | 11819 |     2 |   2343 |    0 | 844 MB  | 0 bytes | 2.02 | 5.77 | 108014 |  30 | select * from test
 postgres | 11401 |     2 |   2345 |    0 | 844 MB  | 0 bytes | 2.05 | 5.74 | 108019 |  24 | select * from test
 postgres | 11391 |     2 |   2311 |    0 | 844 MB  | 0 bytes | 1.98 | 5.79 | 108015 |  31 | select * from test
 postgres | 11754 |     2 |   2313 |    0 | 844 MB  | 0 bytes | 2.13 | 5.84 | 108014 |  42 | select * from test
 postgres | 11825 |     2 |   8232 |    0 | 844 MB  | 0 bytes | 2.49 | 5.60 | 108018 |  86 | select * from test
 postgres | 11425 |     2 |   2344 |    0 | 844 MB  | 0 bytes | 2.70 | 5.52 | 108019 |  84 | select * from test
 postgres | 11444 |     2 |   2340 |    0 | 844 MB  | 0 bytes | 2.42 | 5.71 | 108018 |  32 | select * from test
(11 rows)

    backend_type     | context  | w |  wt  | wb | wbt  | ex | et | hits | ev |   ru   | fs | fst
---------------------+----------+---+------+----+------+----+----+------+----+--------+----+-----
 autovacuum launcher | normal   | 0 | 0.00 |  0 | 0.00 |    |    |   10 |  0 |        |  0 |   0
 autovacuum worker   | normal   | 0 | 0.00 |  0 | 0.00 |  0 |  0 | 1663 |  0 |        |  0 |   0
 client backend      | bulkread | 0 | 0.00 |  0 | 0.00 |    |    |  385 |  0 | 361427 |    |
 client backend      | normal   | 0 | 0.00 |  0 | 0.00 |  0 |  0 | 3091 |  2 |        |  0 |   0
 background worker   | bulkread | 0 | 0.00 |  0 | 0.00 |    |    |  575 |  0 | 716793 |    |
 background worker   | normal   | 0 | 0.00 |  0 | 0.00 |  0 |  0 | 6471 |  0 |        |  0 |   0
```

```
(6 rows)
```

Время выполнения команд при включении **direct i/o** существенно увеличилось - до `0m56`. Результат теста: не стоит включать **direct i/o ,** используя экспериментальный параметр `debug_io_direct`, так как его использование существенно снизит производительность.

Статистики `faults` нулевые. Статистики `reclaims` стали небольшими.

Статистики `writes` нулевые потому, что команды select не грязнили блоки и блоки не записывались.

Практика к главе 16

Часть 1. Использование расширения pg_wait_sampling

1) Проверьте, есть ли следующие библиотеки в директории, чтобы библиотеки могли загрузиться:

```
postgres@tantor:~$ ls `pg_config --pkglibdir` | egrep
"statements|sampling|qualstats|plans|prewarm|kcache"
pg_prewarm.so
pg_qualstats.so
pg_stat_kcache.so
pg_stat_statements.so
pg_store_plans.so
pg_wait_sampling.so
```

В команде используются символы обратного апострофа, а не апостроф.

2) Уберите из следующей команды библиотеки, которых нет в директории и загрузите те, которые есть:

```
postgres@tantor:~$ psql -c "alter system set shared_preload_libraries =
pg_stat_statements, pg_wait_sampling, pg_qualstats, pg_store_plans, pg_prewarm,
pg_stat_kcache;"
ALTER SYSTEM
```

Если какой-то из файлов библиотек будет отсутствовать, то экземпляр не запустится. Команду alter system не удастся выполнить, так как она выполняется на запущенном экземпляре. Чтобы поменять значение параметра `shared_preload_libraries` придётся редактировать текстовый файл в файловой системе. Не стоит останавливать экземпляр не имея доступа к файловой системе, так как он может не запуститься.

В команде указано 6 библиотек.

Порядок перечисления библиотек в параметре `shared_preload_libraries` может быть важен. Например, библиотека **`pg_wait_sampling` должна быть указана позже `pg_stat_statements`**, чтобы `pg_wait_sampling` не перезаписывала идентификаторы запросов (`queryid`), которые используются `pg_wait_sampling`.

Библиотеки могут зависеть друг от друга. Например, библиотека **`pg_stat_kcache`** требует наличия библиотеки **`pg_stat_statements`**, иначе экземпляр не запустится:

```
postgres@tantor:~$ psql -c "alter system set shared_preload_libraries =
pg_wait_sampling, pg_qualstats, pg_store_plans, pg_prewarm, pg_stat_kcache;"
ALTER SYSTEM
postgres@tantor:~$ pg_ctl stop
waiting for server to shut down.... done
server stopped
postgres@tantor:~$ pg_ctl start
waiting for server to start....
[552186] LOG:  Auto detecting pg_stat_kcache.linux_hz parameter...
[552186] LOG:  pg_stat_kcache.linux_hz is set to 62500
[552186] FATAL:  unrecognized configuration parameter "pg_stat_statements.max"
[552186] HINT:  make sure pg_stat_statements is loaded and make sure pg_stat_kcache is
present after pg_stat_statements in the shared_preload_libraries setting
[552186] LOG:  database system is shut down
 stopped waiting
pg_ctl: could not start server
```

3) Перезапустите экземпляр, чтобы библиотеки были загружены:

```
postgres@tantor:~$ sudo systemctl restart tantor-se-server-16
```

Перезагрузка экземпляра эквивалентна остановке экземпляра с финальной контрольной точкой и запуск экземпляра. На экземплярах с большим кэшем буферов (большим числом грязных блоков) финальная контрольная точка может выполняться долго. Для уменьшения простоя перед рестартом экземпляра стоит выполнять контрольную точку или **дожидаться выполнения контрольной точки по времени и останавливать экземпляр сразу после завершения контрольной точки.**

Пример сообщений из диагностического журнала:

```
   [552332] LOG:  received fast shutdown request
   [552332] LOG:  aborting any active transactions
   [552340] LOG:  pg_wait_sampling collector shutting down
   [552332] LOG:  background worker "logical replication launcher" (PID 552341) exited with exit code 1
   [552334] LOG:  shutting down
   [552334] LOG:  checkpoint starting: shutdown immediate
   [552334] LOG:  checkpoint complete: wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled;
write=0.016 s, sync=0.005 s, total=0.057 s; sync files=2, longest=0.003 s, average=0.003 s; distance=0 kB,
estimate=0 kB; lsn=44/1A8598B0, redo lsn=44/1A8598B0
   LOG:  database system is shut down
   [552367] LOG:  starting Tantor Special Edition 16.6.1 a74db619 on x86_64-pc-linux-gnu, compiled by gcc (Astra
12.2.0-14.astra3+b1) 12.2.0, 64-bit
   [552367] LOG:  listening on IPv4 address "127.0.0.1", port 5432
   [552367] LOG:  listening on IPv6 address "::1", port 5432
   [552367] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
   [552371] LOG:  database system was shut down at
   [552367] LOG:  database system is ready to accept connections
   [552375] LOG:  pg_wait_sampling collector started
   [552374] LOG:  autoprewarm successfully prewarmed 947 of 947 previously-loaded blocks
```

При рестарте экземпляра сначала были прерваны активные транзакции, новые сессии не могли создаваться, начался простой обслуживания клиентских приложений. Началась финальная контрольная точка. После завершения контрольной точки экземпляр был снова запущен и простой обслуживания прекратился.

4) Установите расширение в базу данных postgres, посмотрите какие объекты входят в расширение, какие параметры конфигурации есть у расширения:

```
postgres=# create extension pg_wait_sampling;
CREATE EXTENSION
postgres=# \dx+ pg_wait_sampling
    Objects in extension "pg_wait_sampling"
              Object description
-----------------------------------------------
 function pg_wait_sampling_get_current(integer)
 function pg_wait_sampling_get_history()
 function pg_wait_sampling_get_profile()
 function pg_wait_sampling_reset_profile()
 view pg_wait_sampling_current
 view pg_wait_sampling_history
 view pg_wait_sampling_profile
(7 rows)
postgres=# select name, setting, context, min_val, max_val, boot_val,
pending_restart r from pg_settings where name like '%pg_wait_sampling%';
               name               | setting |  context   | min_val |  max_val   | boot_val | r
----------------------------------+---------+------------+---------+------------+----------+--
 pg_wait_sampling.history_period  | 10      | superuser  | 1       | 2147483647 | 10       | f
 pg_wait_sampling.history_size    | 5000    | superuser  | 100     | 2147483647 | 5000     | f
 pg_wait_sampling.profile_period  | 10      | superuser  | 1       | 2147483647 | 10       | f
 pg_wait_sampling.profile_pid     | on      | superuser  |         |            | on       | f
 pg_wait_sampling.profile_queries | top     | superuser  |         |            | top      | f
 pg_wait_sampling.sample_cpu      | on      | superuser  |         |            | on       | f
(6 rows)
```

У расширения есть функционал истории и профиля событий ожидания. Этот функционал не зависит друг от друга.

В профиле могут быть пойманы события ожиданий, а в истории нет и наоборот. Вероятность поймать событие ожидания определяется только двумя параметрами: `pg_wait_sampling.profile_period` и `pg_wait_sampling.history_period`. Чем меньше значение, тем выше вероятность поймать редкое событие ожидания. Профиль не забирает данные из истории, а параметр `pg_wait_sampling.profile_period` не зависит от `pg_wait_sampling.history_period` и `pg_wait_sampling.history_size`.

5) Посмотрите сколько разделяемой памяти использует расширение:

```
postgres=# select * from (select *, lead(off) over(order by off)-off as true
from pg_shmem_allocations) as a where name like '%wait%';
        name        |    off    | size  | allocated_size | true
--------------------+-----------+-------+----------------+-------
 pg_wait_sampling | 148145920 | 17568 |          17664 | 17664
(1 row)
```

Три структуры расширения используют 17664 байт. Большая часть занята очередью (MessageQueue) фиксированного размера 16Кб, памятью под список PID, памятью под идентификаторы команд (queryid), выполняющихся процессами. Размер структуры под хранение списка PID процессов определяется максимальным числом процессов экземпляра. Максимальное число процессов экземпляра зависит от значений параметров: `max_connections, autovacuum_max_workers, max_worker_processes, max_wal_senders`.

6) Выполните команды:

```
postgres=# select * from pg_wait_sampling_history limit 10;
select * from pg_wait_sampling_history where queryid<>0;
select pg_backend_pid();
  pid   |              ts                | event_type |        event         | queryid
--------+--------------------------------+------------+----------------------+---------
 552370 | 2025-07-10 13:17:39.562626+03 | Activity   | BgWriterHibernate    |       0
 552372 | 2025-07-10 13:17:39.562626+03 | Activity   | WalWriterMain        |       0
 553742 | 2025-07-10 13:17:39.57326+03  | Client     | ClientRead           |       0
 552373 | 2025-07-10 13:17:39.57326+03  | Activity   | AutoVacuumMain       |       0
 552376 | 2025-07-10 13:17:39.57326+03  | Activity   | LogicalLauncherMain  |       0
 552374 | 2025-07-10 13:17:39.57326+03  | Extension  | Extension            |       0
 552369 | 2025-07-10 13:17:39.57326+03  | Activity   | CheckpointerMain     |       0
 552370 | 2025-07-10 13:17:39.57326+03  | Activity   | BgWriterHibernate    |       0
 552372 | 2025-07-10 13:17:39.57326+03  | Activity   | WalWriterMain        |       0
 553742 | 2025-07-10 13:17:39.583412+03 | Client     | ClientRead           |       0
(10 rows)

  pid   |              ts                | event_type | event |       queryid
--------+--------------------------------+------------+-------+---------------------
 553742 | 2025-07-10 13:17:43.228602+03 |            |       | 7356378217053506569
(1 row)

 pg_backend_pid
----------------
         553742
(1 row)
```

Вторая команда обычно выдаёт ноль строк, но если выполнить её повторно несколько раз, то с какой-то вероятностью она выдаст строку, относящуюся к серверному процессу, который обслуживает текущую сессию.

Расширение использует фоновый процесс pg_wait_sampling collector, который с частотой заданной параметром pg_wait_sampling.history_period или pg_wait_sampling.profile_period (по умолчанию 10 миллисекунд) опрашивает состояние всех процессов экземпляра и сохраняет 5000 (значение по умолчанию pg_wait_sampling.history_size) событий.

На экземпляре с множеством активных сессий история на 5000 событий может перезаписываться за доли секунды. В истории сохраняются события ожиданий всех процессов. Если серверные процессы не сталкиваются с блокировками, то 99.98% событий ожидания будет заполнено фоновыми процессами и не связано с запросами.

7) Убедитесь, что в истории ровно 5000 событий:

```
postgres=# select count(*), max(ts)-min(ts) duration  from
pg_wait_sampling_history;
 count |    duration
-------+-----------------
  5000 | 00:00:07.572279
(1 row)
```

На простаивающем экземпляре история хранит данные всего лишь за 7 секунд.

8) Поскольку история храни немного данных, можно использовать данные из профиля:

```
postgres=# select * from pg_wait_sampling_profile where queryid>0;
  pid   | event_type |    event     |       queryid        | count
--------+------------+--------------+----------------------+-------
 553742 |            |              | 6003029174018016234 |     2
 553742 | IO         | DataFileRead | 2425453347952179914 |     1
 553742 |            |              | 7356378217053506569 |     2
 553742 |            |              |  153783292416311340 |     1
 553742 | IO         | DataFileRead | 6003029174018016234 |     2
(5 rows)
```

Фоновый процесс аккумулирует (группирует, агрегирует) события ожиданий и подсчитывает их количество (count). Это называется "профиль ожиданий" - число событий ожиданий, сгруппированных по:
1) типам и видам событий (event_type, event)
2) процессам (pid) если pg_wait_sampling.profile_pid=true
3) типам команд (queryid), если pg_wait_sampling.profile_queries=true.

Данные сохраняются по всем процессам, в том числе по серверным процессам завершившихся сессий, которые уже отсутствуют в операционной системе. Если `pg_wait_sampling.profile_pid=on`, то число строк в представлении только увеличивается.

9) Запустите во втором терминале нагрузку:

```
postgres@tantor:~$ pgbench -C -c 90 -T 900 -P 10
pgbench (16.6)
starting vacuum...end.
progress: 10.0 s, 188.2 tps, lat 429.108 ms stddev 469.398, 0 failed
progress: 20.0 s, 196.1 tps, lat 468.844 ms stddev 638.098, 0 failed
```

Параметр -C порождает большое число сессий.

10) Проверьте, что число строк в представлении только увеличивается:

```
postgres=# select count(*) from pg_wait_sampling_profile;
\watch 10

 count
-------
 50453
(1 row)
```

```
 count
-------
 55256
(1 row)

^C
postgres=#
```

Строки находятся в структуре памяти и если число строк будет большим, памяти может быть занято много. Расширение не очищает память, используемую этой структурой.

Нужно периодически освобождать память вызовом функции `pg_wait_sampling_reset_profile()`, либо установить `pg_wait_sampling.profile_pid=false`.

11) Перед выполнением сравнительных тестов бывает удобно сбросить статистики, чтобы накопленные с предыдущего теста данные не попадали в результат. Для этого используются функции сброса статистик. Выполните сброс всех статистик:

```
postgres=#
select pg_stat_reset();
select pg_stat_reset_shared(null);
select pg_stat_reset_shared('bgwriter');
select pg_stat_reset_shared('archiver');
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('wal');
select pg_stat_reset_shared('recovery_prefetch');
select pg_stat_reset_slru(null);
select pg_stat_statements_reset();
select pg_stat_kcache_reset();
select pg_wait_sampling_reset_profile();
select pg_qualstats_reset();

 pg_stat_reset
---------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_shared
----------------------

(1 row)

 pg_stat_reset_slru
--------------------

(1 row)
```

```
   pg_stat_statements_reset
-------------------------------
 2025-02-10 13:53:27.95483+03
(1 row)

 pg_stat_kcache_reset
----------------------

(1 row)

 pg_wait_sampling_reset_profile
--------------------------------

(1 row)

ERROR:  function pg_qualstats_reset() does not exist
LINE 1: select pg_qualstats_reset();
```

Последней функции нет, так как расширение `pg_qualstats` не было установлено.