Узкие места PostgreSQL

Александр Коротков

Postgres Professional

2019

Дисклеймер

- Я не расскажу про все узкие места, а только про некоторые. Те которые встретились в жизни, и/или зацепили взгляд в коде.
- Истины не существует, о том, что насколько значимо и/или типично можно долго спорить. Всё, что я излагаю – сугубо моё ИМХО.
- Ну, что поделать!

Пример №1: интенсивный UPDATE

Пример №1: DDL

Простая задача – считаем хиты.

```
                Table "public.url"
 Column  |  Type   | Collation | Nullable | Default
---------+---------+-----------+----------+---------
 id      | integer |           | not null |
 href    | text    |           | not null |
 param1  | text    |           |          |
.........................................................
 param20 | text    |           |          |
 hits    | bigint  |           | not null | 0
Indexes:
    "url_pkey" PRIMARY KEY, btree (id)
    "url_href_index" btree (href)
    "url_param1_idx" btree (param1)
.........................................................
    "url_param20_idx" btree (param20)
```

Пример №1: Нагрузочное тестирование

```
\set id random(1, 100000000)
UPDATE url SET hits = hits + 1 WHERE id = :id + 1;
```

```
$ pgbench -c 60 -j 60 -M prepared -f script1.sql \
  -T 1000 -P 1 postgres
```

- ~200 000 TPS
- ~20 MB/sec writes

Пример №1: Суровая реальность

```
\set id random_zipfian(1, 100000000, 1.5)
UPDATE url SET hits = hits + 1 WHERE id = :id + 1;
```

```
$ pgbench -c 60 -j 60 -M prepared -f script1.sql \
  -T 1000 -P 1 postgres
```

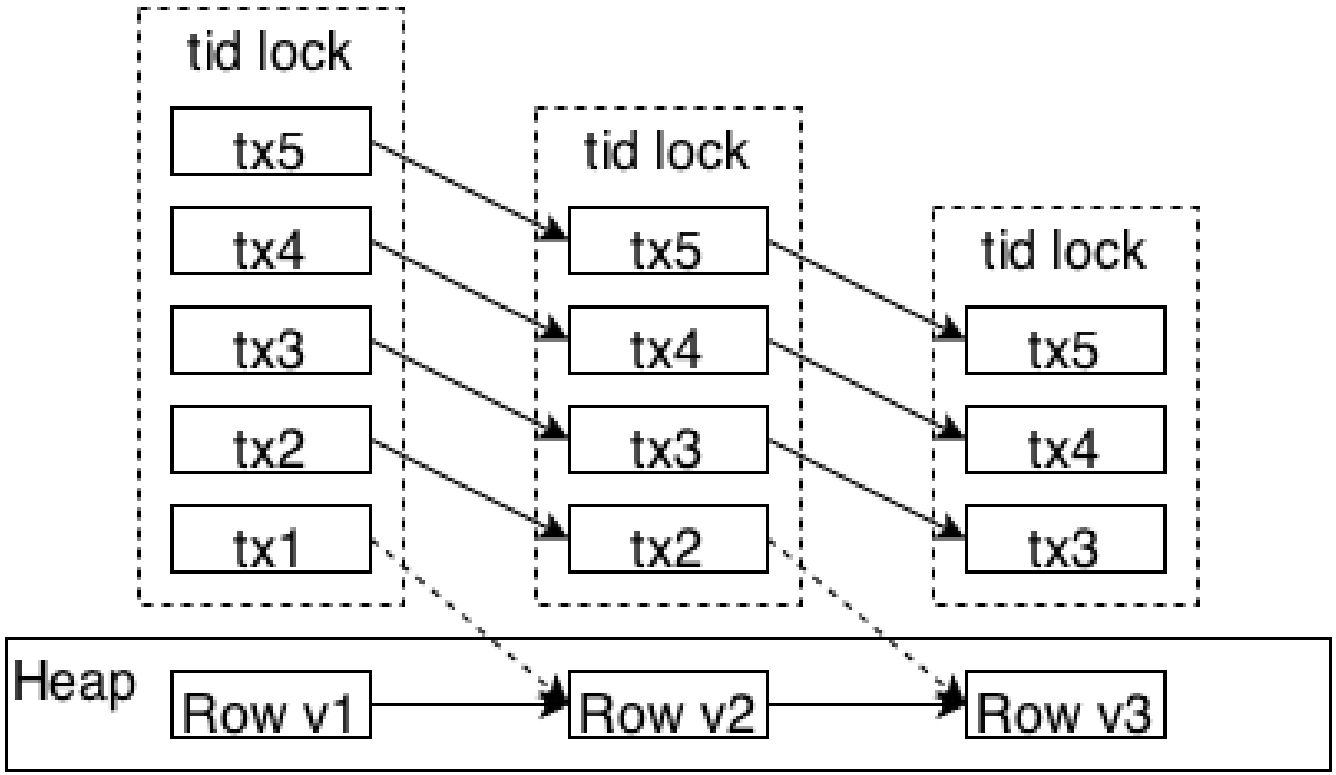
- ~15 000 TPS (в ~10 раз ниже)
- ~5 MB/sec writes (в ~5 раз выше bytes/TPS)

Пример №1: Что пошло не так?

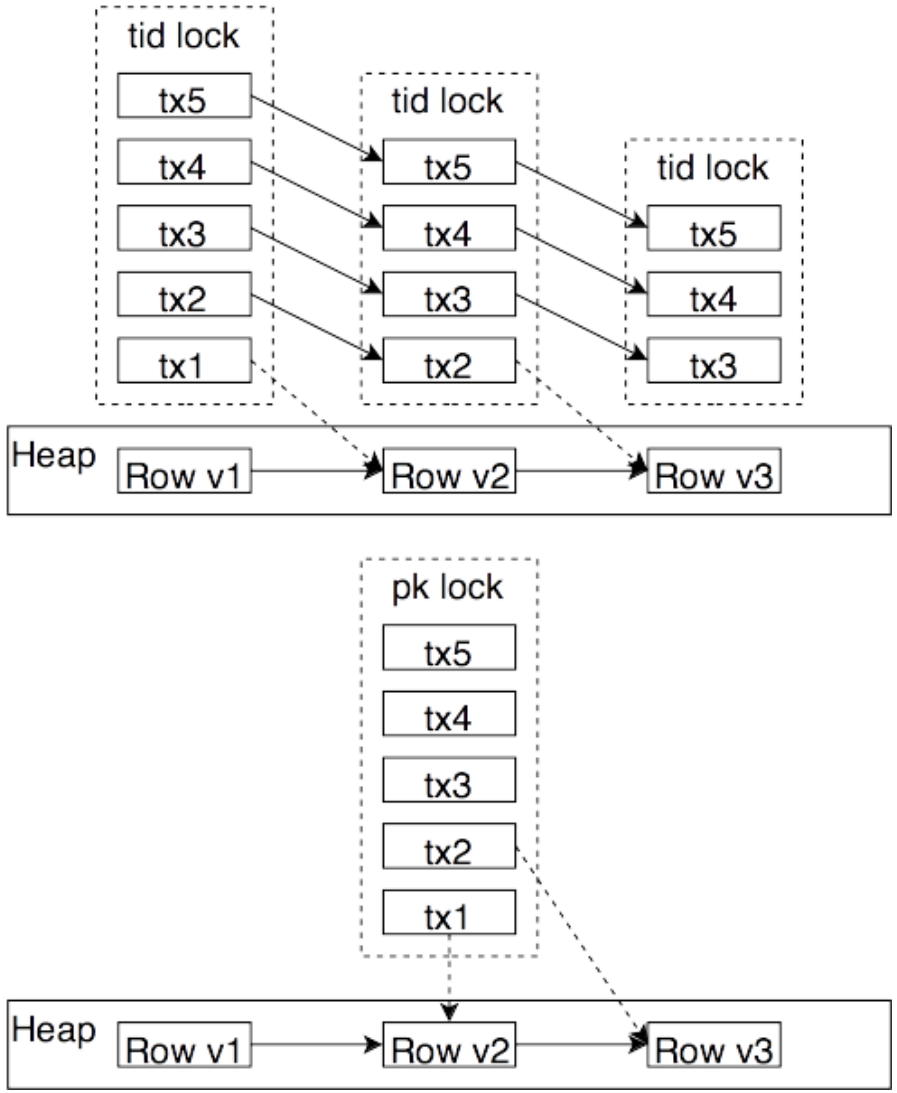
Неравномерное распределение, из-за этого:

- Большая конкуренция за строки,
- Хуже работает HOT.

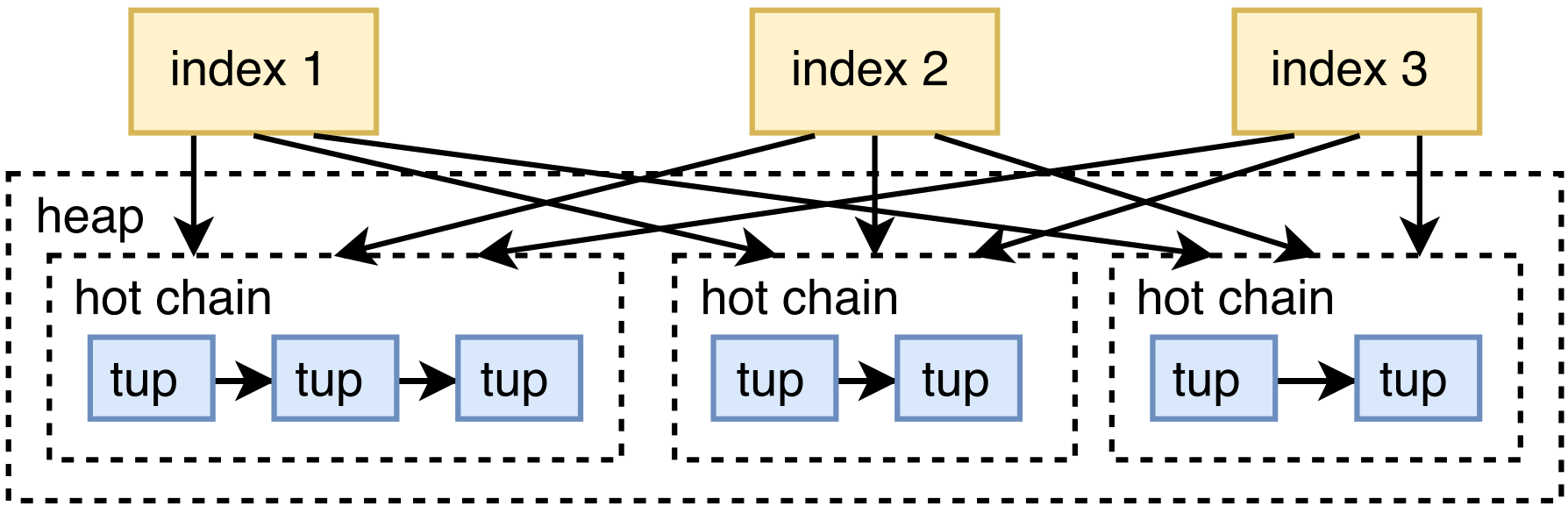
Tuple lock: проблема



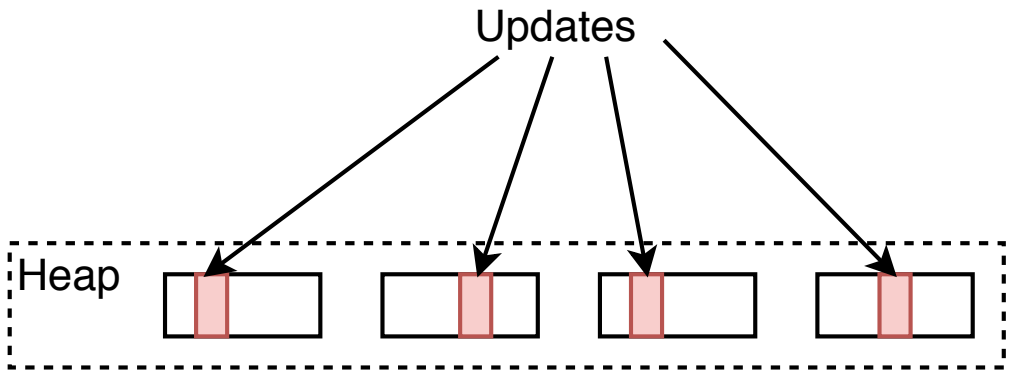
Tuple lock: патч



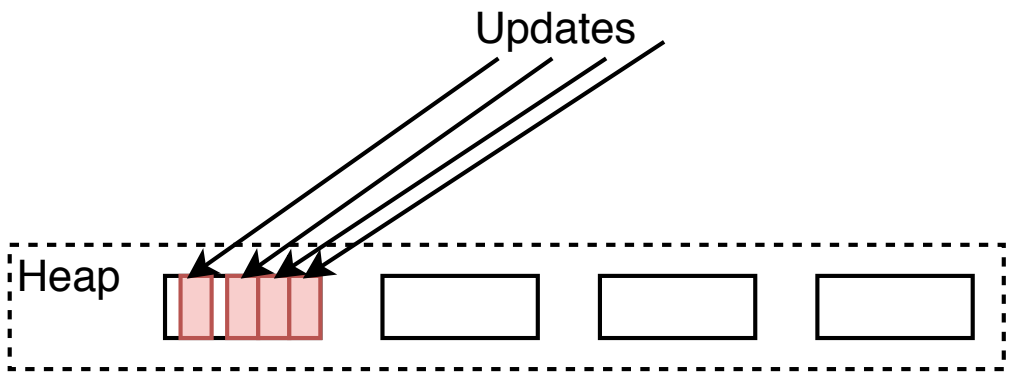
HOT


index 1
index 2
index 3
heap
hot chain
tup
tup
tup
hot chain
tup
tup
hot chain
tup
tup

Хорошо для HOT



Плохо для HOT



Костыль: table split

```
BEGIN;
ALTER TABLE url DROP COLUMN hits;
CREATE TABLE url_hits (id INTEGER PRIMARY KEY,
                       hits BIGINT NOT NULL);
COMMIT;
```

```
\set id random_zipfian(1, 100000000, 1.5)
UPDATE url_hits SET hits = hits + 1 WHERE id = :id;
```

- ~30 000 TPS (в ~2 раза лучше)
- ~4 MB/sec writes (в ~3 раз лучше bytes/TPS)

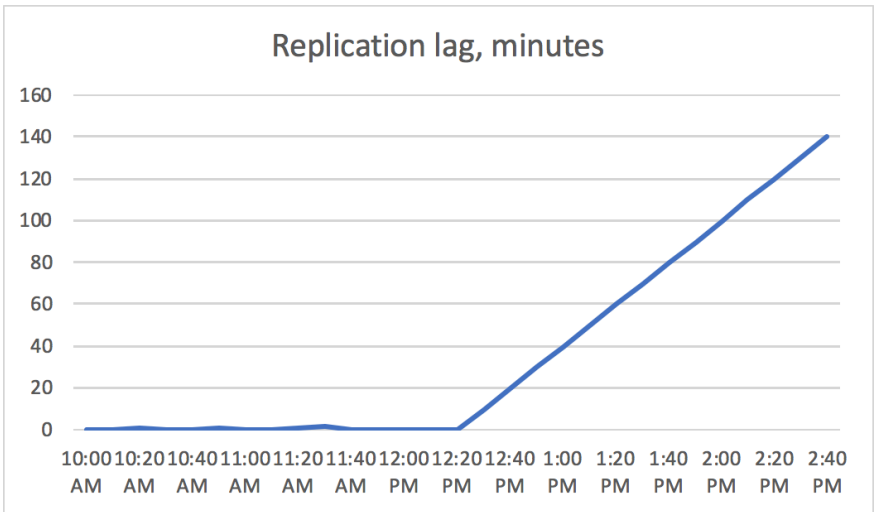
Пример №1: Резюме

- Проводите нагрузочное тестирование на разных распределениях нагрузки и данных. В жизни (чаще всего) распределения будут “косыми” и коррелированными.
- На “косых” распределениях возникает конкуренция за отдельные строки.
- На “косых” распределениях хуже работает HOT. Можно выделить часто обновляемую часть колонок в отдельную таблицу, тогда не нужно будет обновлять много индексов.
- Мониторинг: pg_locks, pg_stat_activity(.wait_event .wait_event_type), pg_stat_all_tables(.n_tup_hot_upd .n_tup_upd)

Пример №2: проблемы с shared_buffers

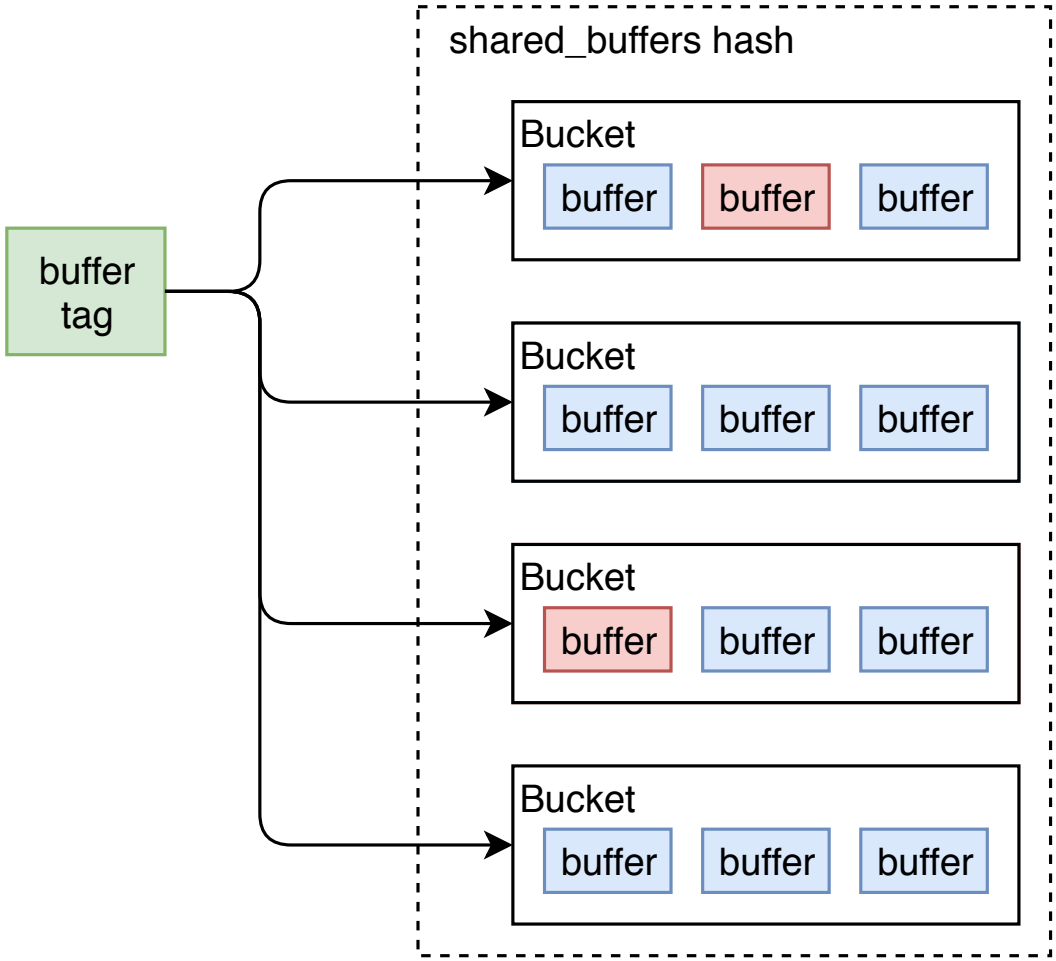
Пример №2

- Удалю-ка я несколько ненужных таблиц, где-то 1000-1500 всего.
- Что-то тормозит, я лучше в параллель.
- Ой, а где моя реплика???!!!



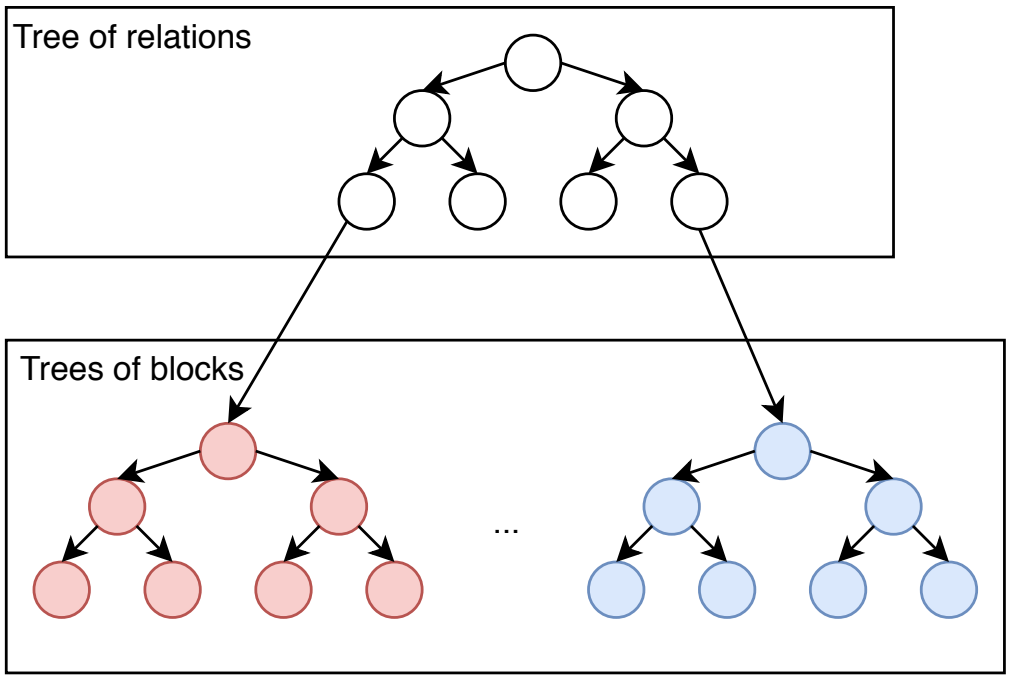
Проблемы с shared_buffers

- shared_buffers – хэш-таблица
- Поиск конкретного buffer tag’а за O(1) :)
- Поиск всех буферов таблицы за O(N) :(



Светлое будущее: переделка shared_buffers

- shared_buffers – radix tree
- Поиск конкуретного buffer tag'а за O(log(n)), зато есть локальность
- Поиск всех буферов таблицы за O(n) :)



Небольшое облегчение в PostgreSQL 12

Если несколько DROP TABLE сгруппировано в одну транзакцию, то на реплике они вычищаются из shared_buffers за один проход.

Теперь вместо

```
DROP TABLE tab1; ... DROP TABLE tabN;
```

лучше писать

```
BEGIN;
DROP TABLE tab1; ... DROP TABLE tabN;
COMMIT;
```

Пример №2: Резюме

- Реплике тяжело последовательно сканировать shared_buffers в один поток.
- В некоторых случаях можно оптимизировать нагрузку на реплику (DROP TABLE в одной транзакции).
- А ещё можно уменьшить на реплике shared_buffers :)
- Мониторинг: perf, gdb (sampling) :)

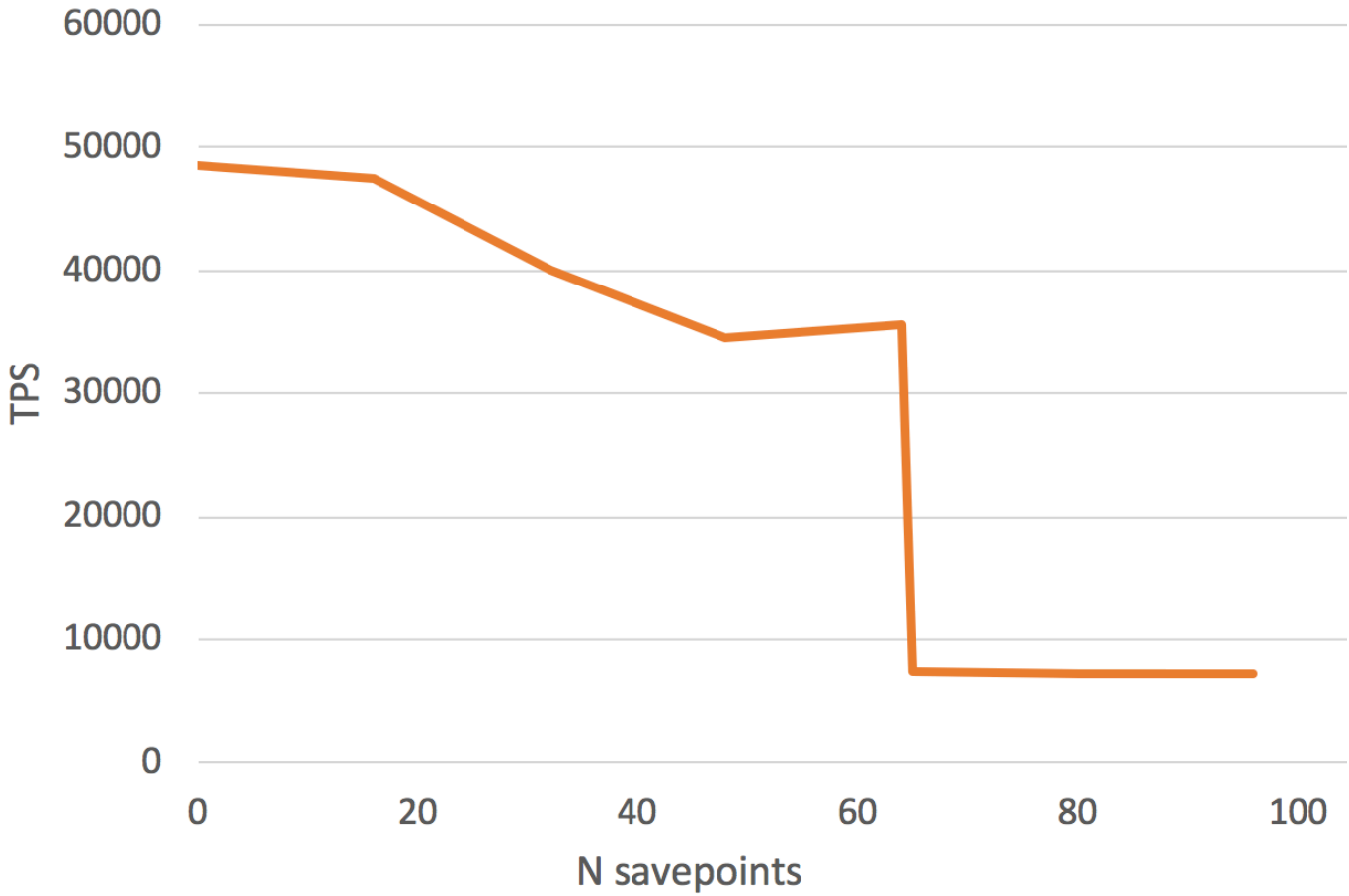
Пример №3: проблемы с SAVEPOINT’ами

Пример №3 (1/2)

```
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-5000, 5000)
BEGIN;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
SAVEPOINT s1;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
....
SAVEPOINT sN;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
SELECT pg_sleep(1.0);
END;
```

Пример №3 (2/2)

```
pgbench -c 2 -j 2 -M prepared -f script.sql@1 -T 1000 -P 1 postgres &
pgbench -c 60 -j 60 -M prepared -T 1000 -P 1 postgres
```



o_O

Subtransactions (1/2)

```
/*
 * Each backend advertises up to PGPROC_MAX_CACHED_SUBXIDS TransactionIds
 * for non-aborted subtransactions of its current top transaction.  These
 * have to be treated as running XIDs by other backends.
 *
 * We also keep track of whether the cache overflowed (ie, the transaction has
 * generated at least one subtransaction that didn't fit in the cache).
 * If none of the caches have overflowed, we can assume that an XID that's not
 * listed anywhere in the PGPROC array is not a running transaction.  Else we
 * have to look at pg_subtrans.
 */
#define PGPROC_MAX_CACHED_SUBXIDS 64    /* XXX guessed-at value */

struct XidCache
{
    TransactionId xids[PGPROC_MAX_CACHED_SUBXIDS];
};
```

```
Snapshot
GetSnapshotData(Snapshot snapshot)
{
.....................................................................
                if (pgxact->overflowed)
                    suboverflowed = true;
```

Subtransactions (2/2)

```
bool
XidInMVCCSnapshot(TransactionId xid, Snapshot snapshot)
.....................................................................
         *
         * We start by searching subtrans, if we overflowed.
         */
        if (snapshot->suboverflowed)
        {
            /*
             * Snapshot overflowed, so convert xid to top-level.  This is safe
             * because we eliminated too-old XIDs above.
             */
            xid = SubTransGetTopmostTransaction(xid);
```

- Сабтранзакции хранятся в pg_subtrans SLRU.
- Помните, каждый TRY/CATCH в PL/pgSQL – это SAVEPOINT!

Что такое SLRU?

- Структура данных для маленького числа буферов (не более 128).
- Поиск нужного буфера последовательный.
- Доступ к разделяемой памяти защищён LWLock’ов. IO каждого буфера защищено своим LWLock’ом.

Откуда взялись SLRU? (1/2)

```
commit 2589735da08c4e597accb6eab5ae65b6339ee630
Author: Tom Lane <tgl@sss.pgh.pa.us>
Date:   Sat Aug 25 18:52:43 2001 +0000

    Replace implementation of pg_log as a relation accessed through the
    buffer manager with 'pg_clog', a specialized access method modeled
    on pg_xlog.  This simplifies startup (don't need to play games to
    open pg_log; among other things, OverrideTransactionSystem goes away),
    should improve performance a little, and opens the door to recycling
    commit log space by removing no-longer-needed segments of the commit
    log.  Actual recycling is not there yet, but I felt I should commit
    this part separately since it'd still be useful if we chose not to
    do transaction ID wraparound.
```

Откуда взялись SLRU? (2/2)

```
commit 0abe7431c6d7a022e7f24a4f145c702900f56174
Author: Bruce Momjian <bruce@momjian.us>
Date:   Wed Jun 11 22:37:46 2003 +0000

    This patch extracts page buffer pooling and the simple
    least-recently-used strategy from clog.c into slru.c.  It doesn't
    change any visible behaviour and passes all regression tests plus a
    TruncateCLOG test done manually.

    Apart from refactoring I made a little change to SlruRecentlyUsed,
    formerly ClogRecentlyUsed:  It now skips incrementing lru_counts, if
    slotno is already the LRU slot, thus saving a few CPU cycles.  To make
    this work, lru_counts are initialised to 1 in SimpleLruInit.

    SimpleLru will be used by pg_subtrans (part of the nested transactions
    project), so the main purpose of this patch is to avoid future code
    duplication.

    Manfred Koizar
```

Пример №3: Что делать?

- Не делать более 64 SAVEPOINT’ов :)
- Увеличить PGPROC_MAX_CACHED_SUBXIDS в исходниках (~50% шутки)
- Ждать zheap и других undo-based storage
- Купить у Postgres Pro секретный костыль
- Разработчикам постгреса – уносить всё с SLRU в buffer manager
- Мониторить pg_stat_activity(.wait_event .wait_event_type)

Пример №4: очень плохой (для постгреса) UPDATE

Пример №4 (1/3)

```
UPDATE   tab
SET      col1 = val1, -- Все колонки, даже те,
         col2 = val2, -- которые не менялись
         col3 = val3,
         col4 = val4,
         col5 = val5;
```

Очень плохо, но так могут делать ORM! Из-за этого не работает HOT!

Пример №4 (2/3)

```
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-5000, 5000)
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta, bid = bid
WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta, bid = bid
 WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta
WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
```

```
CREATE INDEX pgbench_accounts_bid_idx ON pgbench_accounts (bid);
CREATE INDEX pgbench_tellers_bid_idx ON pgbench_tellers (bid);
```

Пример №4 (3/3)

```
$ pgbench -c 60 -j 60 -M prepared -f script.sql \
  -T 1000 -P 1 postgres
```

~7 000 TPS

```
$ pgbench -c 60 -j 60 -M prepared -T 1000 -P 1 postgres
```

~115 000 TPS

Пример №4: Резюме

- Не обновляйте колонки, которые не обновляете!
- Если не получается в приложении, то для удобства можно завести хранимку.
- Мониторить pg_stat_all_tables(.n_tup_hot_upd .n_tup_upd).

Пример №5: не бесплатные row-level locks

Пример №5

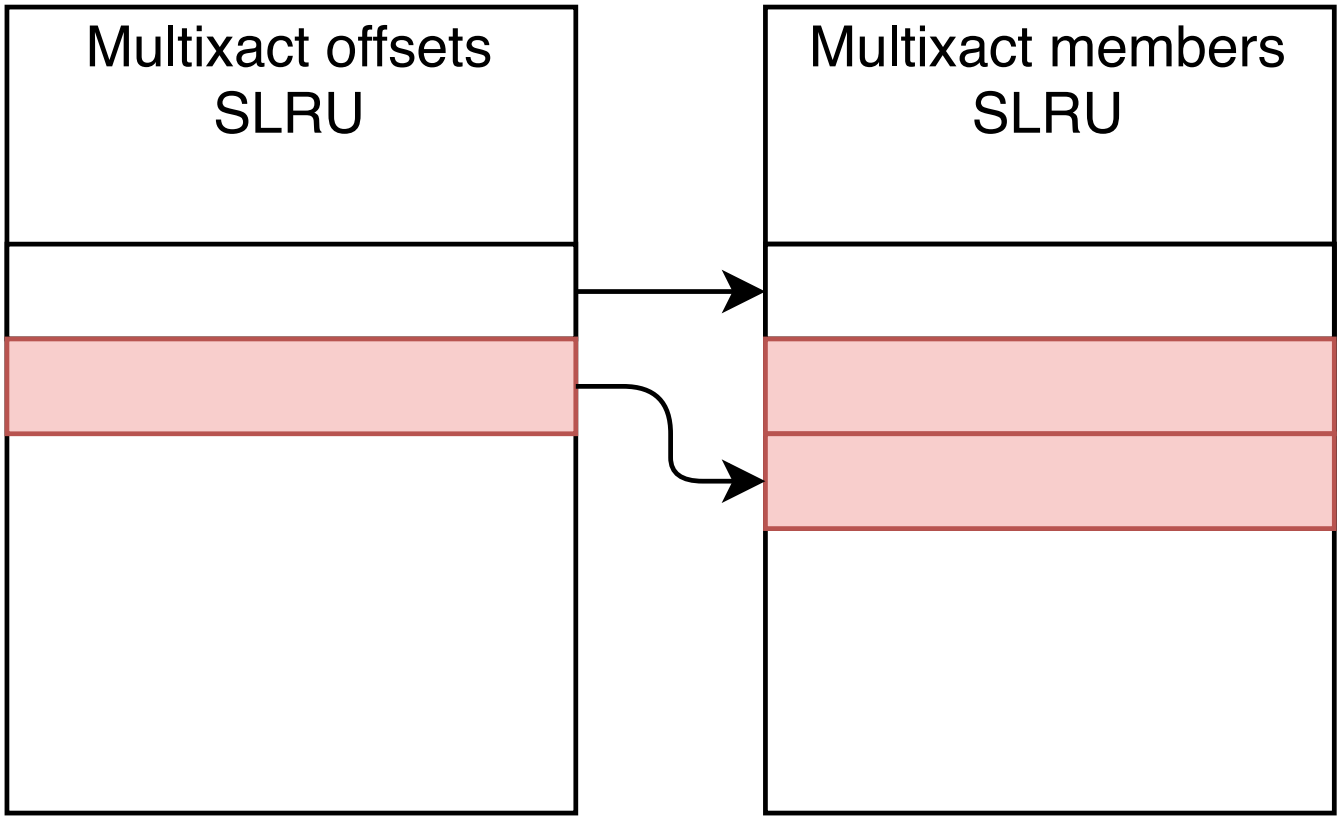
```
\set aid random_zipfian(1, 100000 * :scale, 1.5)
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

620 000 TPS, 0 MB\sec write

```
\set aid random_zipfian(1, 100000 * :scale, 5.0)
SELECT abalance FROM pgbench_accounts
WHERE aid = :aid FOR SHARE;
```

125 000 TPS, 15 MB\sec write

Multixacts



23163 multixact/sec, 2.06 offsets/multixact

Костыль: advisory locks

```
\set aid random_zipfian(1, 100000 * :scale, 1.5)
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

620 000 TPS, 0 MB\sec write

```
\set aid random_zipfian(1, 100000 * :scale, 5.0)
SELECT abalance FROM pgbench_accounts
WHERE aid = :aid FOR SHARE;
```

125 000 TPS, 15 MB\sec write,

```
\set aid random_zipfian(1, 100000 * :scale, 1.5)
BEGIN;
SELECT pg_advisory_xact_lock_shared(:aid);
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
COMMIT;
```

290 000 TPS, 0 MB\sec write

Пример №5: Резюме

- Если брать много row-level shared lock’ов, то есть риск распухания multixacts.
- Multixacts пишутся на диск.
- Костыль в виде advisory locks может помочь.
- Мониторить pg_stat_activity(.wait_event .wait_event_type), pg_control_checkpoint.

Вместо заключения.

Postgres PROFESSIONAL

Спасибо за внимание!