Узкие места PostgreSQL #2

Александр Коротков

Postgres Professional

2020

О докладе

- Коллекция кейсов, как постгрес внезапно и неожиданно для всех загнулся.
- Первая часть лежит здесь: https://pgconf.ru/2019/247964
- В прошлый раз зашло не плохо, решил повторить тот же формат со свежим материалом.

План

- Упёрлись в BufMappingLock
- Упёрлись в ProcArrayLock
- Коллизии в fast path locking

Кейс №1: (Почти) вечное ожидание BufMappingLock

Воспроизведение: setup

- c5d.18xlarge (72 VCPU)
- postgresql.conf

```
shared_buffers = 8GB
synchronous_commit = off
max_connections = 300
```

```
create table test1 (id serial primary key,
                    value int8 not null);
create table test2 (id serial primary key,
                    value int8 not null);
insert into test1 (value)
  (select i from generate_series(1, 30) i);
```

Воспроизведение: pgbench

- script1.sql

```
select * from test1 where id in (1,2, ... , 30);
```

- script2.sql

```
\set value random(1, 100000)
insert into test2 (value) values (:value);
```

- Запускаем в параллель

```
pgbench -M prepared -f script1.sql -c 150 \
        -j 150 -T 1000000 -P 1 postgres
pgbench -M prepared -f script2.sql -c 1 \
        -j 1 -T 1000000 -P 1 postgres
```

Наблюдаемое поведение

- script1.sql

  Стабильно 64-65 kTPS.

- script2.sql

```
progress: 21.0 s, 2741.9 tps, lat 0.364 ms stddev 0.502
progress: 22.0 s, 2076.7 tps, lat 0.393 ms stddev 0.537
progress: 23.0 s, 0.0 tps, lat 0.000 ms stddev 0.000
progress: 24.0 s, 0.0 tps, lat 0.000 ms stddev 0.000
progress: 25.0 s, 0.0 tps, lat 0.000 ms stddev 0.000
.....................................................
progress: 846.0 s, 0.0 tps, lat 0.000 ms stddev 0.000
```

Было 2.5-3.5 kTPS, стало 0.

PROFESSIONAL Postgres

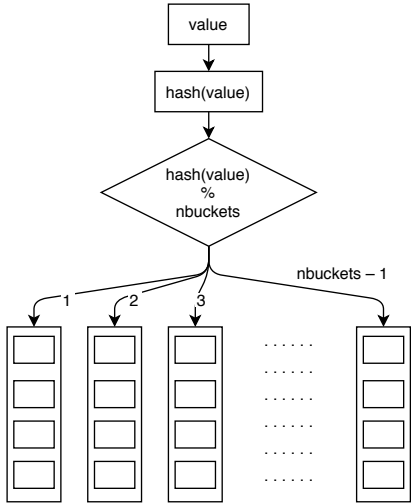
Кто виноват?

Ищем виновного: easy mode

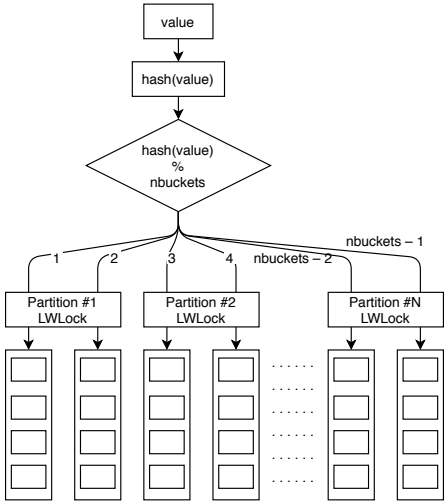
```
# select query, wait_event_type, wait_event
  from pg_stat_activity;
-[ RECORD 1 ]---+---------------------------------------
query           | insert into test2 (value) values ($1);
wait_event_type | LWLock
wait_event      | buffer_mapping
```

Кто такой buffer_mapping?

Buffer mapping: hash table
block_number => buffer_number

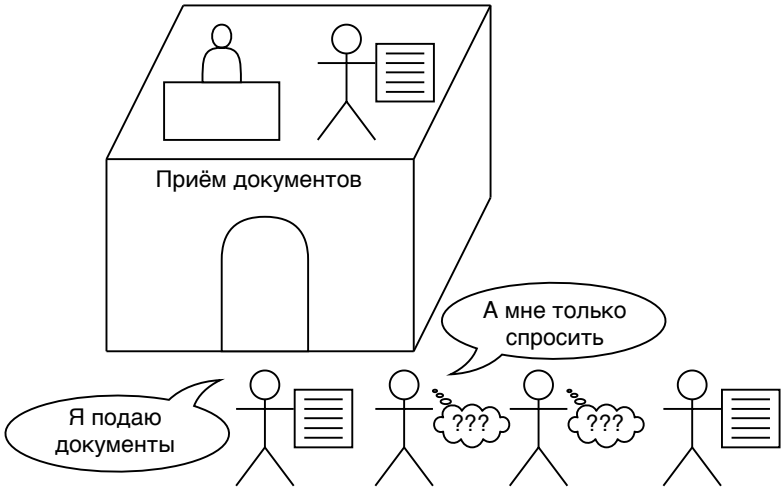


Buffer mapping: shared hash table
block_number => buffer_number

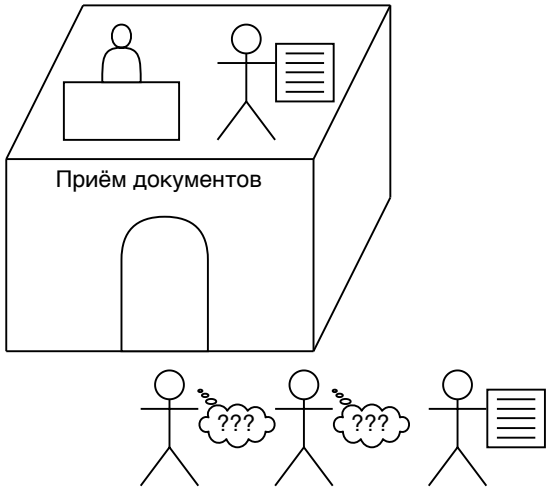


Кто такой LWLock?

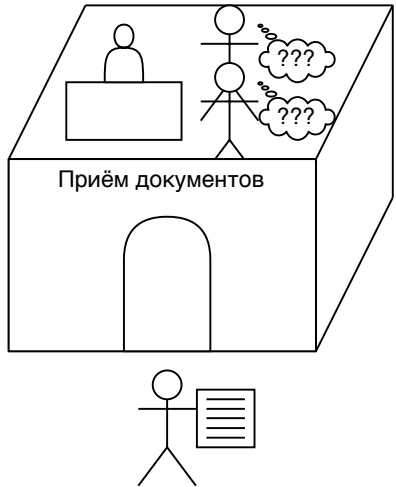
LWLock: работа очереди (1/4)



LWLock: работа очереди (2/4)



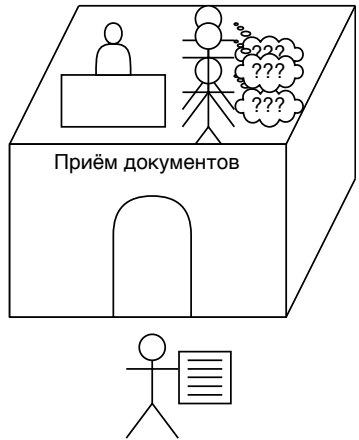
LWLock: работа очереди (3/4)



LWLock: работа очереди (4/4)

LWLock: shared приходит вне очереди (1/2)

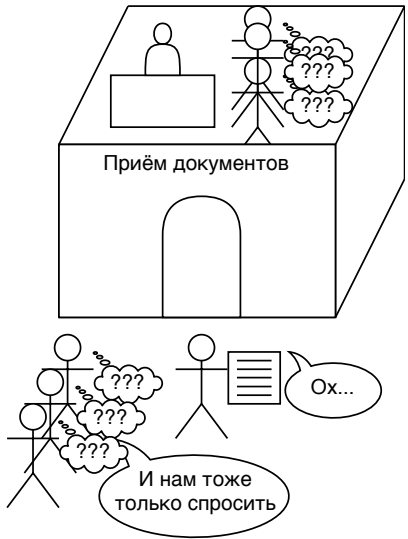
LWLock: shared приходит вне очереди (2/2)



Postgres PROFESSIONAL

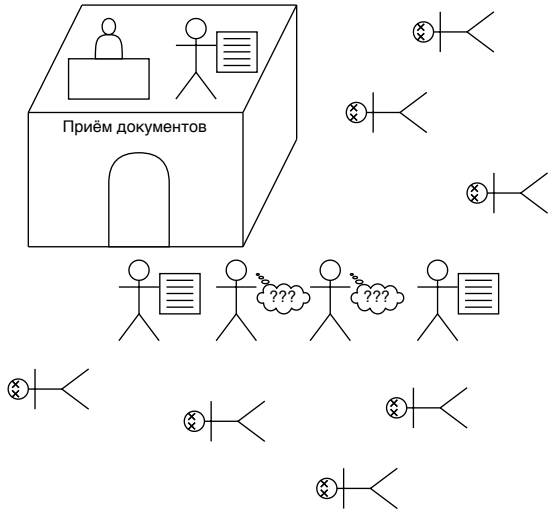
Что может пойти не так?

Exclusive LWLock waiter starvation



Что делать?

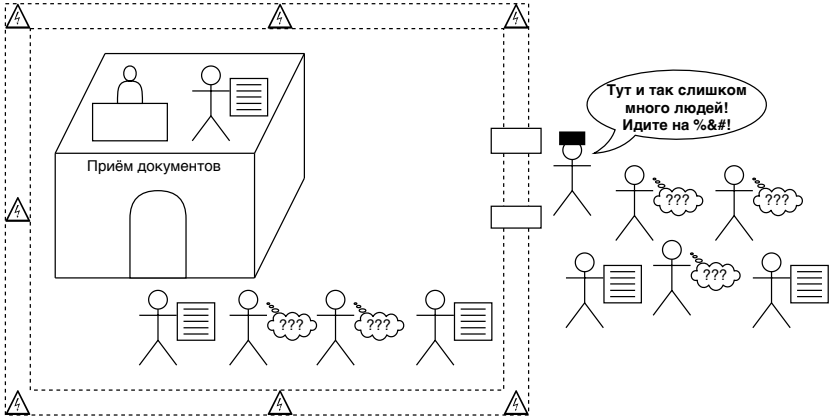
Решение 1: коронавирус



Решение 1: техника

- В современных процессорах слишком много ядер! А Вы ещё вставили слишком много процессоров в свой сервер!
- Купите (арендуйте) сервер по-хуже!
- Или ограничьте использование CPU со стороны PostgreSQL.

Решение 2: ограничить доступ



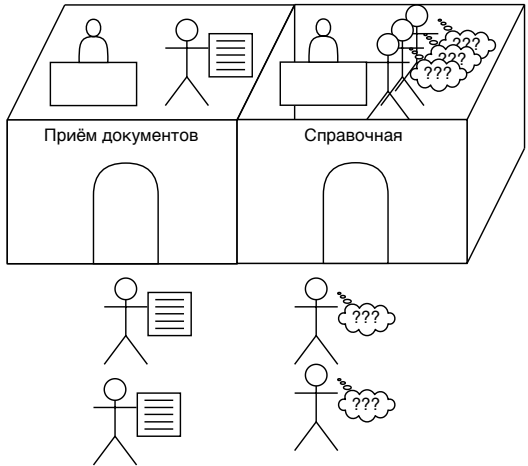
Решение 2: техника

- У вас слишком много read-only нагрузки!
- Ограничьте read-only нагрузку, которая вызывает проблему.
  - На уровне приложения
  - Балансировщиком нагрузки: pgbouncer, odyssey, ...

Это же не наш метод!

Давайте чинить СУБД!

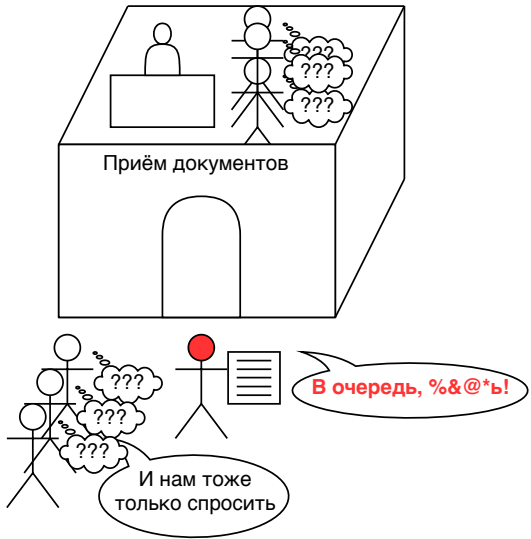
Решение 3: убрать конфликт



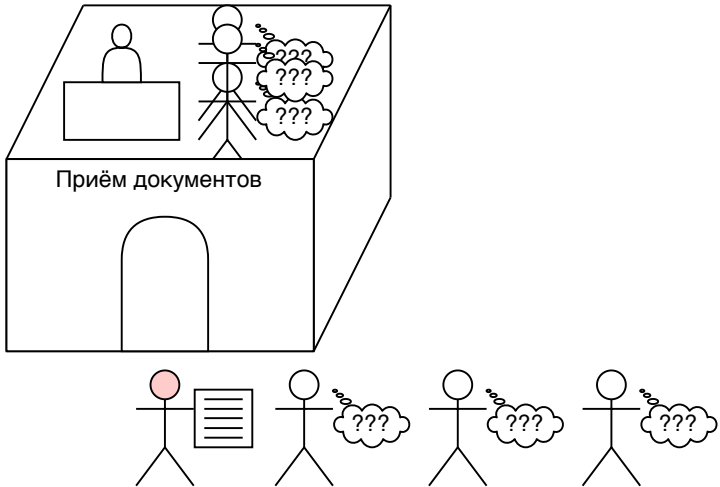
Решение 3: техника

- Можно починить LWLock use cases.
  - BufMappingLock: lockless trie for buffer mapping
  - ProcArrayLock: CSN snapshots
  - WALInsertLock: lockless queue for WAL writer
- Круто!
- Но трудоёмко, а значит долго. А нам нужно решение прямо сейчас!

Решение 4: “честная очередь” (1/2)



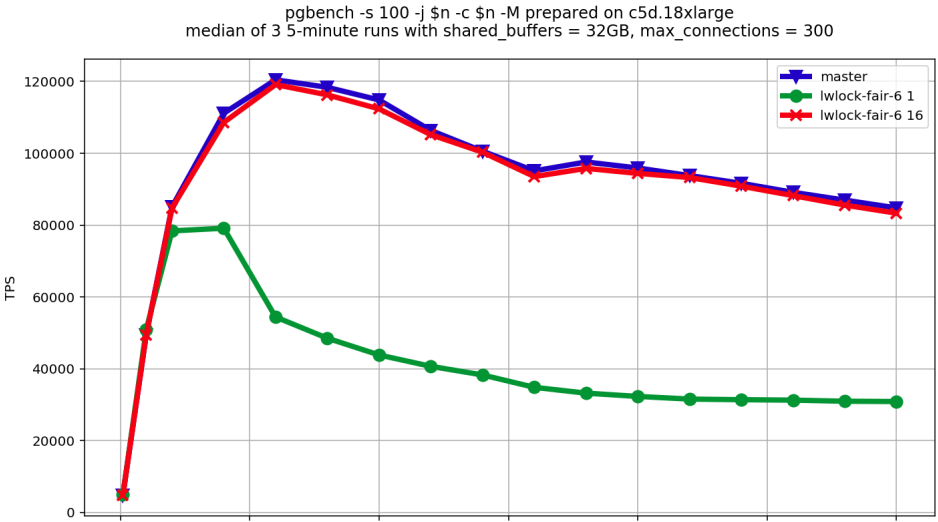
Решение 4: “честная очередь” (2/2)



Решение 4: техника

- Патч “more fair” LWLock: `http://bit.ly/2uPUdpU`
- После того, как проходит lwlock_shared_limit последовательных shared locker’ов, переводим LWLock в “честный” режим.
- В “честном” режиме, shared locker’ы становятся в очередь.
- Когда очередь доходит до exclusive locker’а, он снимает “честный” режим.

“More fair” LWLock benchmark: regression



“More fair” LWLock benchmark: наш кейс

- script1.sql
  Стабильно 64-65 kTPS.
- script2.sql
  Стабильно 2.7-3.0 kTPS.
- То, что надо!

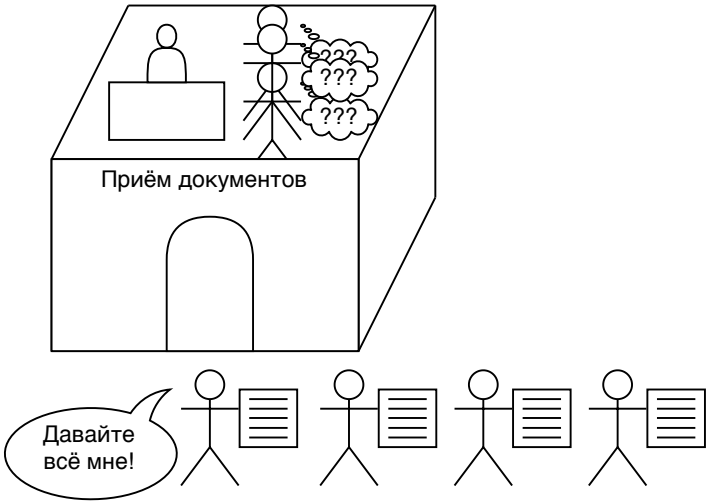
Но патч не закоммичен!

Как использовать “more fair” LWLock

- Если есть $$$, то купить Postgres Pro Enterprise.
- Если нет $$$, то собрать себе PostgreSQL с патчем на свой страх и риск.
- А ещё быть сознательным участником сообщества и поделиться своим опытом в треде.
- И не забыть сказать: “~~Борис,~~ Андрес, ты не прав!”.

Кейс №2: И ProcArrayLock туда же

Групповые оптимизации (1/3)



Групповые оптимизации (2/3)

Групповые оптимизации (3/3)

Групповые оптимизации в PostgreSQL

- Оптимизация exclusive ProcArrayLock при commit’e (group ProcArray clear xid) – PostgreSQL 9.6
- Оптимизация exclusive ClogControlLock (group clog update) – PostgreSQL 11.
- **Помогает, но не спасает!**

Воспроизведение: pgbench

- Setup такой же как в прошлом кейсе
- script1.sql

```
select 1; select 2; ... select 30;
```

- Запускаем

```
pgbench -M prepared -f script1.sql -c 150 \
        -j 150 -T 1000000 -P 1 postgres
```

Наблюдаемое поведение

- script1.sql
  Стабильно 34-35 kTPS.
- Никто больше подключиться не может.
- Только 117 коннектов работают, 33 зависли на authentication.

Кто виноват?

ProcArrayLock

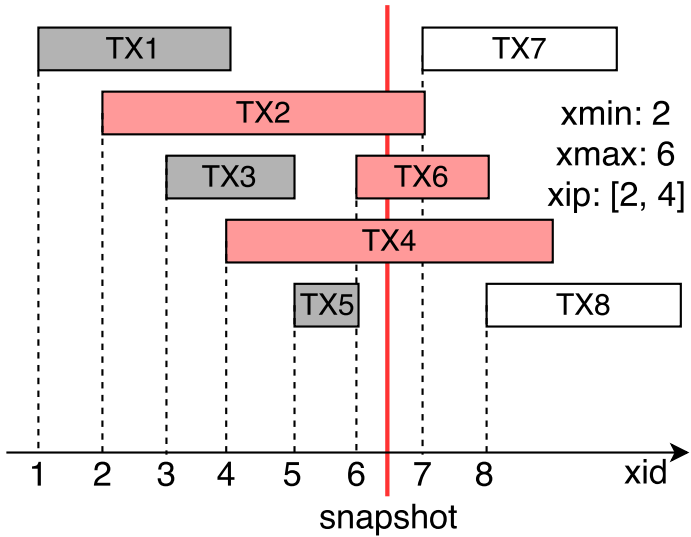
- Берётся shared на взятие снапшота.
- Берётся exclusive на коннект, на коммит транзакции.
- “SELECT const;” берёт снапшот, “;” не берёт (gotcha!).
- Бенчмарк “забил” ProcArrayLock наглухо.

Что делать?

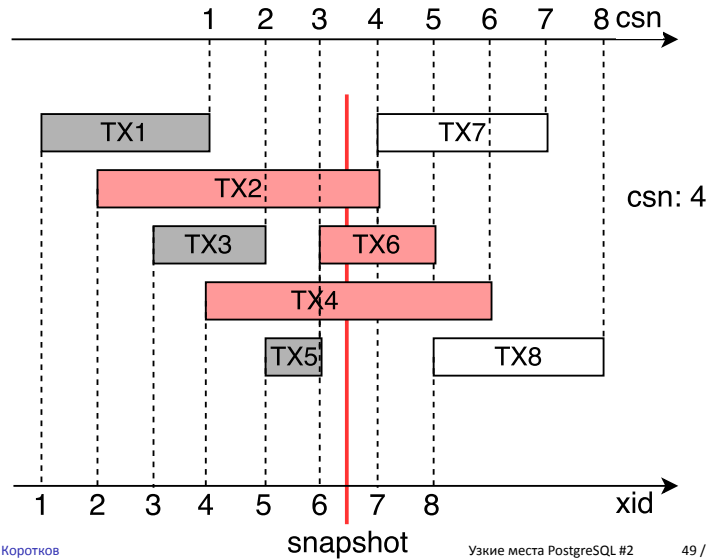
Что делать с ProcArrayLock?

- Уменьшить нагрузку (см. предыдущий кейс).
- Использовать “more fair” LWLock (см. предыдущий кейс).
- Применить CSN патч: `http://bit.ly/2S9ma3V`
- Оптимизировать нагрузку!

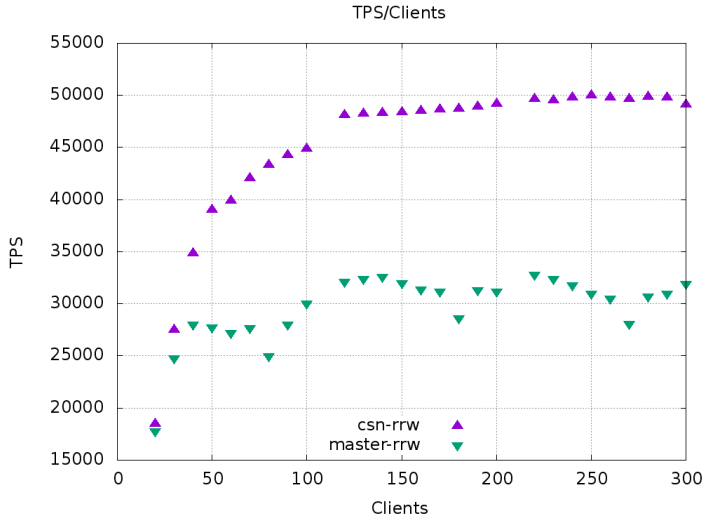
Current snapshot model



CSN model


1 2 3 4 5 6 7 8 csn
TX1
TX7
TX2
csn: 4
TX3
TX6
TX4
TX5
TX8
1 2 3 4 5 6 7 8 xid
snapshot

CSN benchmark: 90% read, 10% write



Оптимизируем нагрузку на ProcArrayLock

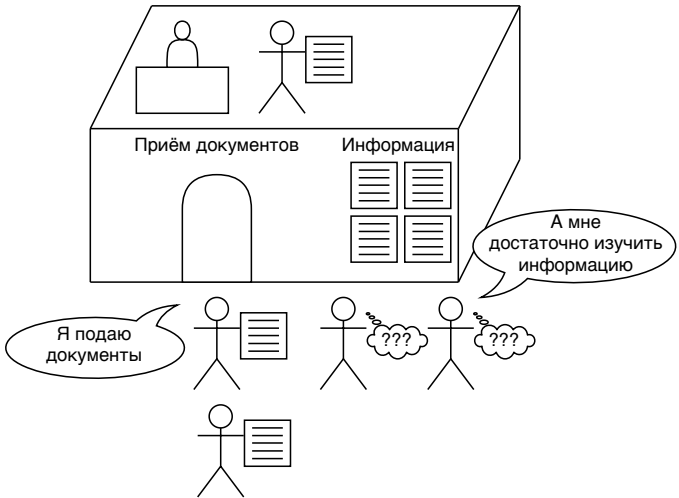
- script2.sql

```
select 1; ... select 30; select txid_current();
```

- txid_current() делает транзакцию “пишущей”, разбавляет сплошной поток shared lock’ов.
- Стабильно 33-34 kTPS (~3% замедление)
- Все подключились!

Кейс №3: Коллизия fast path locking

“Fastpath” locking



Воспроизведение: setup

- Машина и postgresql.conf такие же как в прошлых кейсах
- ```
  pgbench -i -s 1000 postgres
  ```
- Функции отсюда https://gist.github.com/akorotkov/fce8ec80e3b0bf113b68a82fe41294a3
- ```
  create table nocollision (i int);
  select make_collision();
  ```

Воспроизведение: скрипты pgbench

- nocollision.sql

```
truncate nocollision;
```

- collision.sql

```
truncate collision;
```

Воспроизведение: запускаем pgbench

- Запускаем в параллель

```
pgbench -M prepared -S -c 150 \
        -j 150 -T 1000000 -P 1 postgres
pgbench -M prepared -f nocollision.sql -c 10 \
        -j 10 -T 1000000 -P 1 postgres
```

- Запускаем в параллель

```
pgbench -M prepared -S -c 150 \
        -j 150 -T 1000000 -P 1 postgres
pgbench -M prepared -f collision.sql -c 10 \
        -j 10 -T 1000000 -P 1 postgres
```

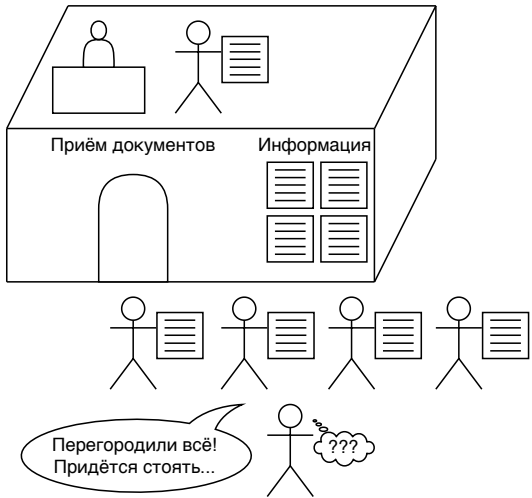
Наблюдаемое поведение

- -S – 830-840 kTPS
  -f nocollision.sql – 200-250 TPS
- -S – 145-150 kTPS
  -f collision.sql – 500-600 TPS

Postgres PROFESSIONAL

Что происходит?

Коллизия“fastpath” locking



Что делать?

Как не попадать на коллизии fast path

- Не берите strong locks (> ShareUpdateExclusive) слишком часто.
- Если запросы, которые ограничиваются weak locks (< ShareUpdateExclusive) висят с wait_type = ’lock_manager’ – это подозрительно.
- Поменяйте oid и всё пройдёт :)

Вместо заключения.

Postgres PROFESSIONAL

Спасибо за внимание!