

# Identifying Bottlenecks in Postgres Workloads

Andres Freund  
PostgreSQL Developer & Committer  
Email: [andres@anarazel.de](mailto:andres@anarazel.de)  
Email: [andres.freund@microsoft.com](mailto:andres.freund@microsoft.com)

<https://anarazel.de/talks/2026-10-22-pgconf-eu-bottlenecks/bottlenecks.pdf>

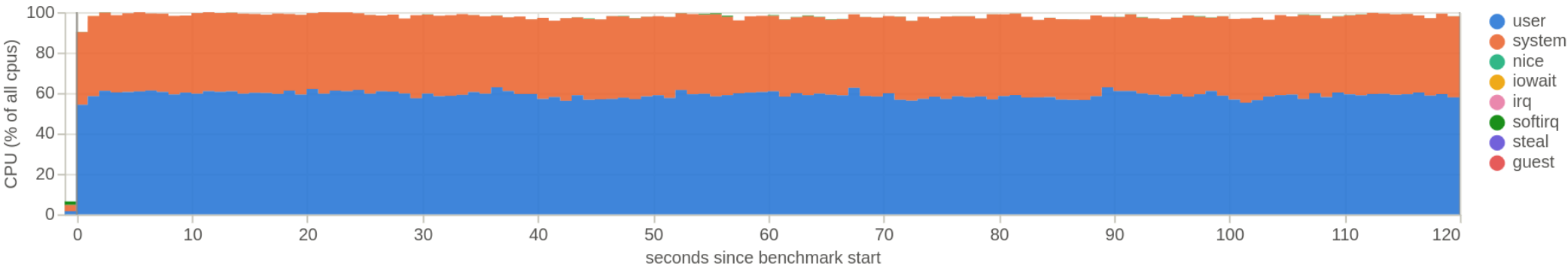
# Forms of bottlenecks

- Throughput
- Using up of {CPU, disk bandwidth / iops, ...}
- Pricing per {CPU, ...}
- WAL / Archiving / Data Storage Costs
- Future estimated limiting factor
- ...

# CPU

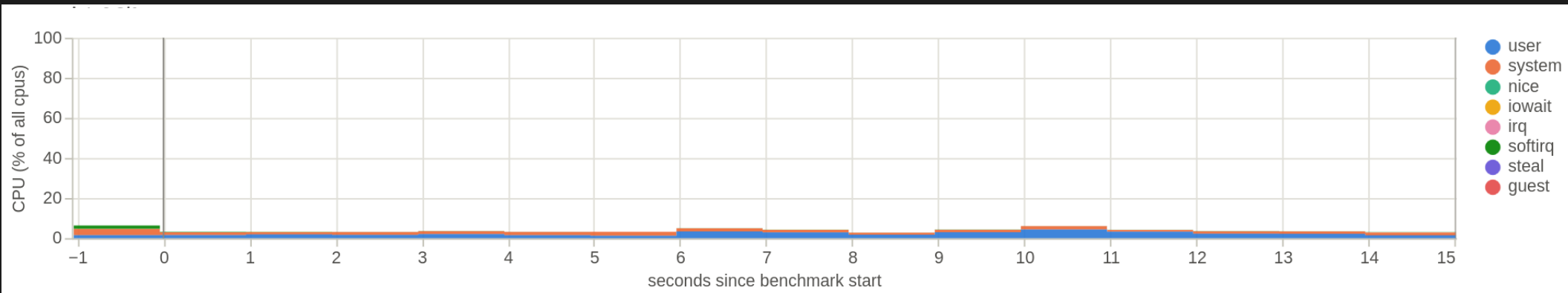
states from /proc/stat (user and nice without guest);

**pgbenchro/clients=64**



cpu stats, collect from /proc/stats, e.g. via mpstat

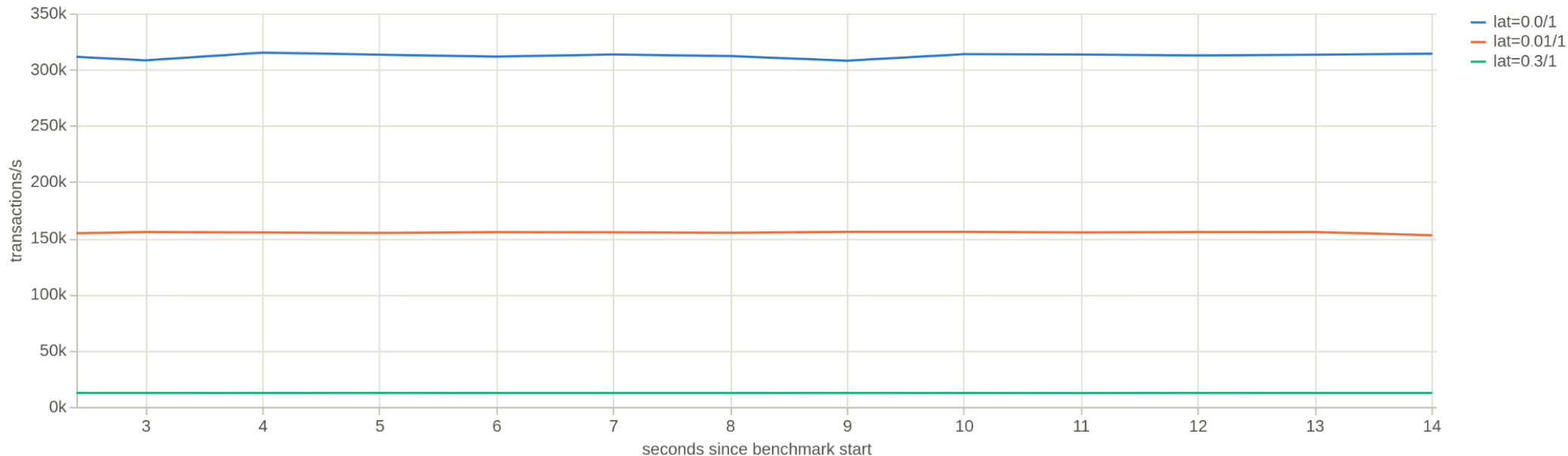
# Network Latency



pgbench -S, scale 100, 8 clients

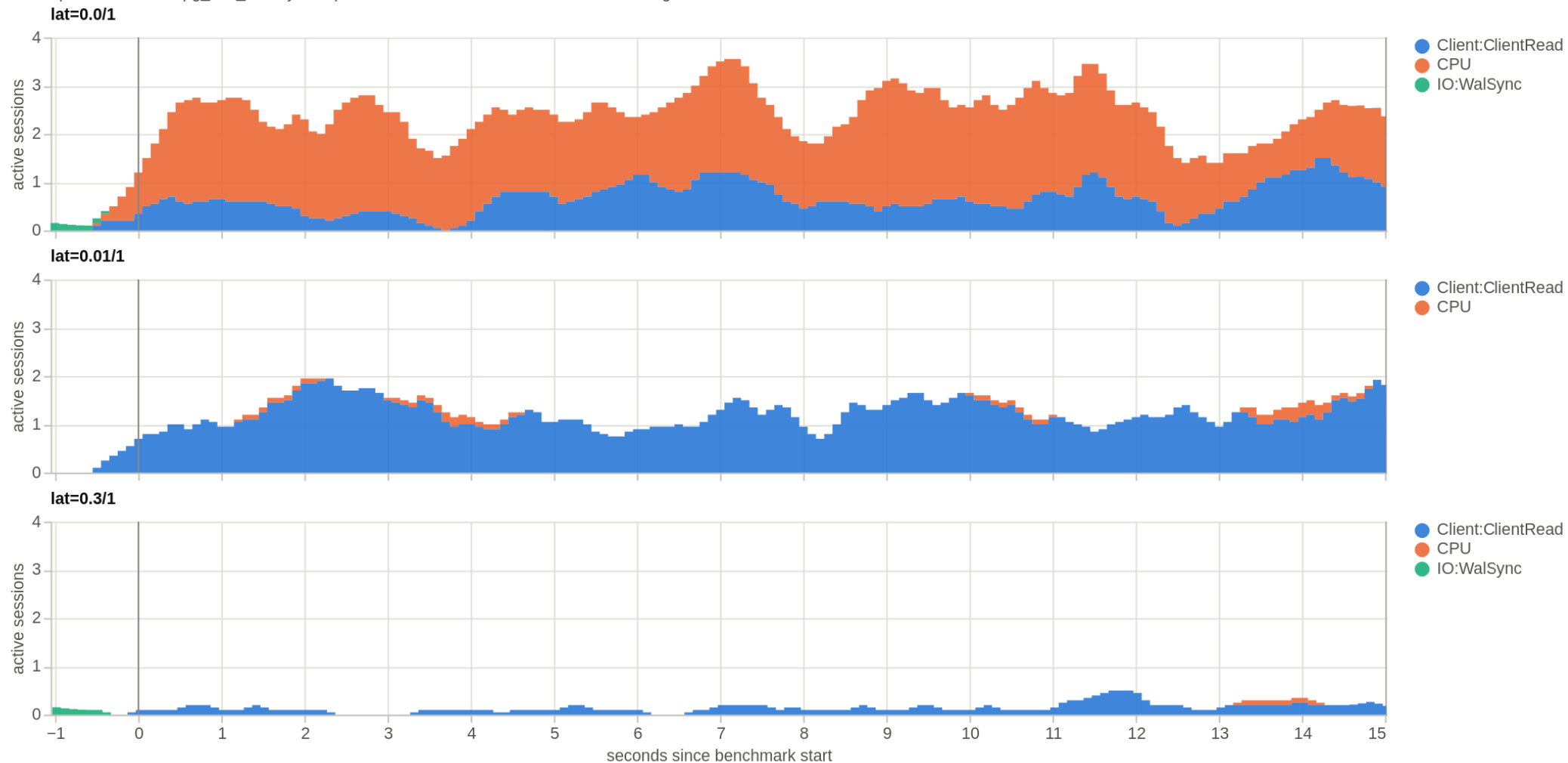
# transactions/s

gray rules: benchmark start and ends



## wait events

sessions per wait event in pg\_stat\_activity, sampled; CPU = active without a wait event; rolling mean over 1 s



wait events, collect from pg\_stat\_activity, interpret NULL wait\_event as CPU, filter by active

```
SELECT pid, greatest('0', clock_timestamp() - query_start),  
       substr(query, 1, 25)  
FROM pg_stat_activity  
WHERE state = 'active' AND pid != pg_backend_pid();
```

| pid    | greatest        | substr                    |
|--------|-----------------|---------------------------|
| 534701 | 00:00:00.000078 | SELECT abalance FROM pgbe |

(1 row)

```
SELECT total_exec_time::int, calls,  
       (mean_exec_time * 1000)::int mean_us,  
       substr(query, 1, 25)  
FROM pg_stat_statements  
ORDER BY calls DESC  
LIMIT 5;
```

| total_exec_time | calls    | mean_us | substr                    |
|-----------------|----------|---------|---------------------------|
| 71916           | 10849349 | 7       | SELECT abalance FROM pgbe |

# ss -nti 'sport 5432'

...

```
bbr wscale:10,10 rto:203.333 rtt:2.474/0.472 ato:40 mss:32768 pmtu:65536
rcvmss:536 advmss:65464 cwnd:18 bytes_sent:39322558 bytes_acked:39322558
bytes_received:56856039 segs_out:517399 segs_in:517402 data_segs_out:517398
data_segs_in:517399 bbr:
(bw:32768000000bps,mrtr:0.008,pacing_gain:2.88672,cwnd_gain:2.88672) send
1907272433bps lastsnd:4 pacing_rate 325725462640bps delivery_rate 32768000000bps
delivered:517399 app_limited busy:72936ms rcv_rtt:19.129 rcv_space:68244
rcv_ssthresh:65464 minrtt:0.008 snd_wnd:65536 rcv_wnd:65536
```

...

## rtt:2.474/0.472

Patch to add contrib/pg\_stat\_tcpinfo by Jakub Wartak

<https://postgr.es/m/CAKZiRmxd8HkOpsH1y3nBL4GPSvY09kjZTLz5B8L6iGOKuV3tNg%40mail.gmail.com>

# Network Latency: What Now?

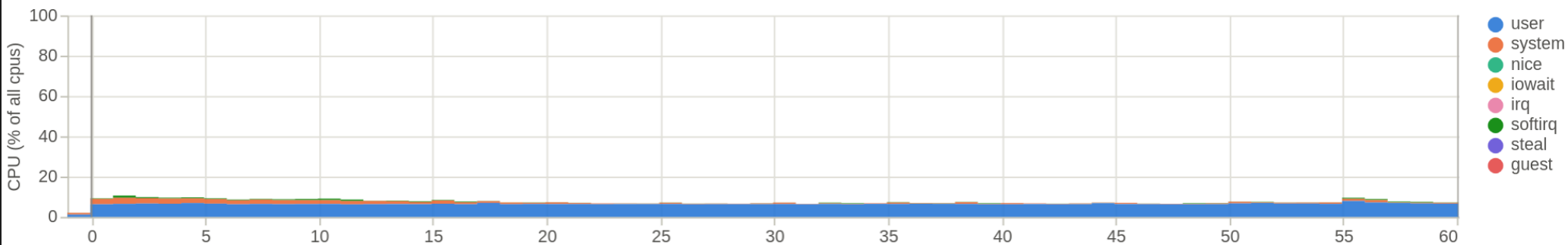
- Combine Queries / Avoid “N+1” problem
- Use Pipelining
- Move client closer to server
- Server-side querying
- Faster Networking

CPU Bound

## CPU

states from /proc/stat (user and nice without guest); gray rules: benchmark start and end

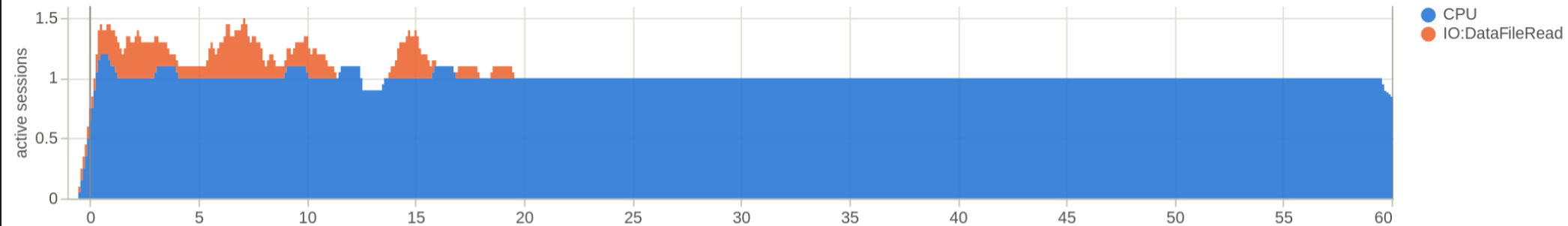
**one-thread/2**



## wait events

sessions per wait event in pg\_stat\_activity, sampled; CPU = active without a wait event; rolling mean over 1 s

one-thread/2

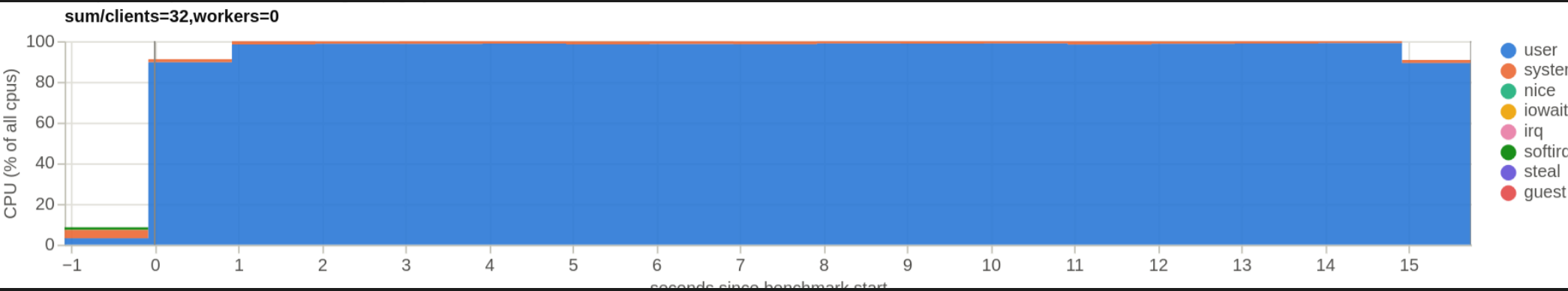
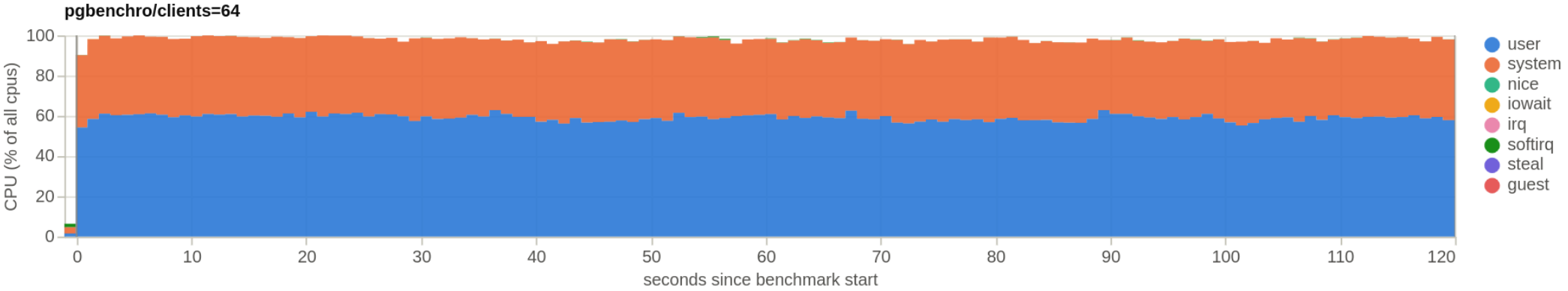


# Single Query: What Now?

- EXPLAIN ANALYZE
- Identify bottleneck
- More efficient queries
- Better schema
- Parallelism

CPU

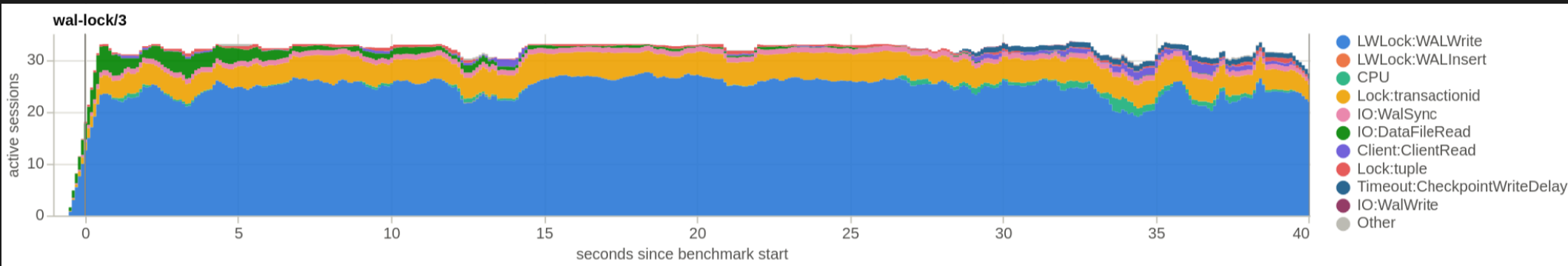
states from /proc/stat (user and nice without guest); gray rules: benchmark start and end



# Aggregate CPU: What now?

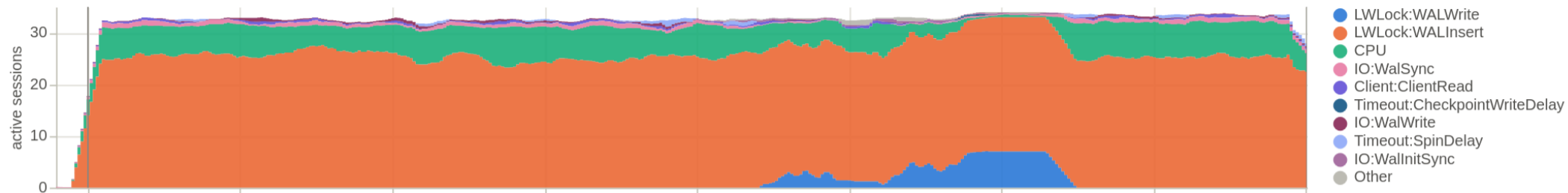
- Avoid ~100% CPU usage for extended periods
- Bigger HW
- Optimize Workload
  - System vs User
  - All single query stuff too
- **decrease parallelism**

# WAL Latency

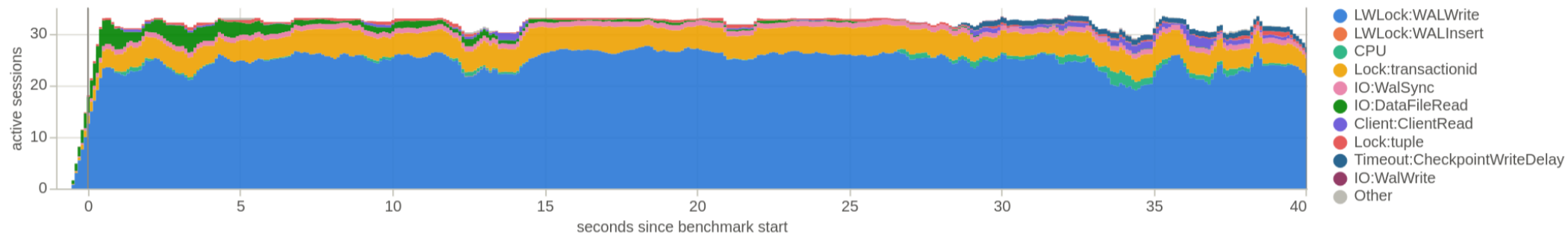


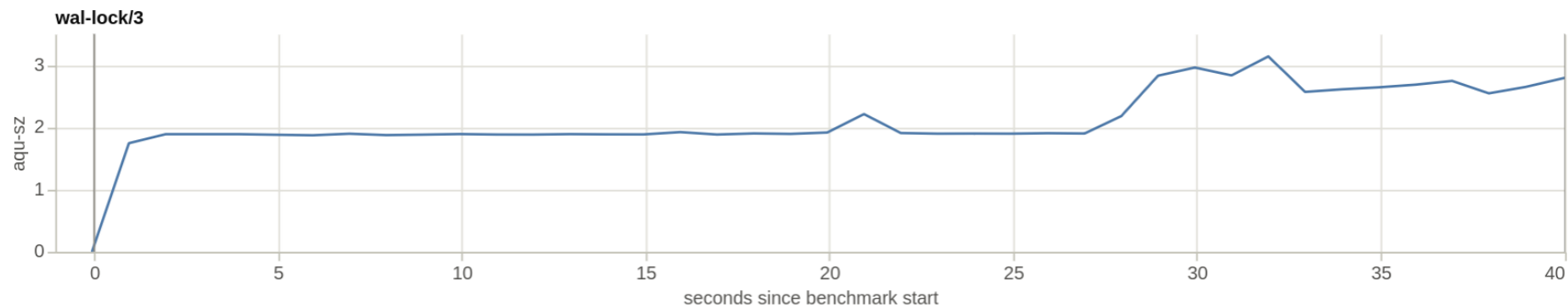
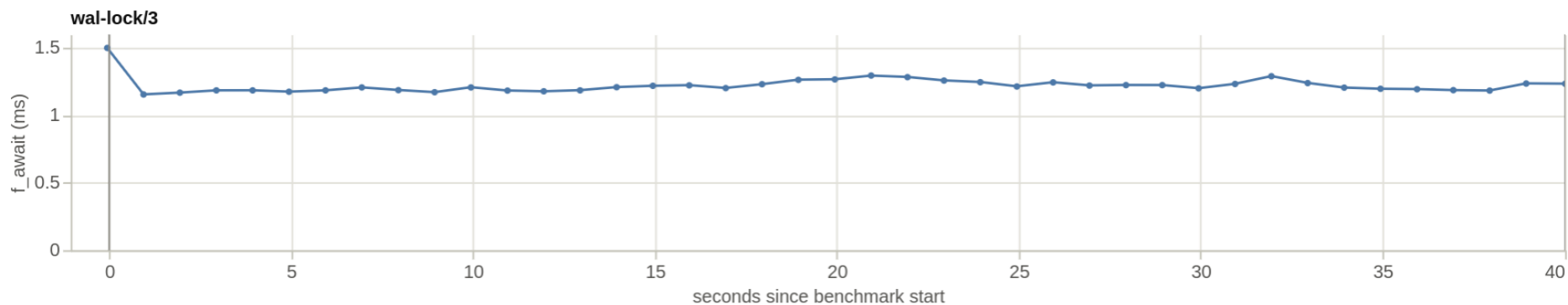
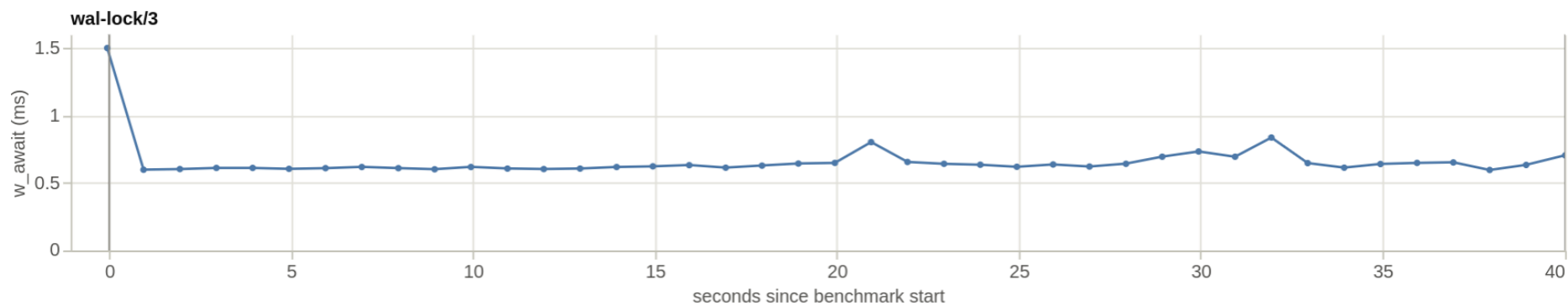
pgbench, scale 100, 32 clients

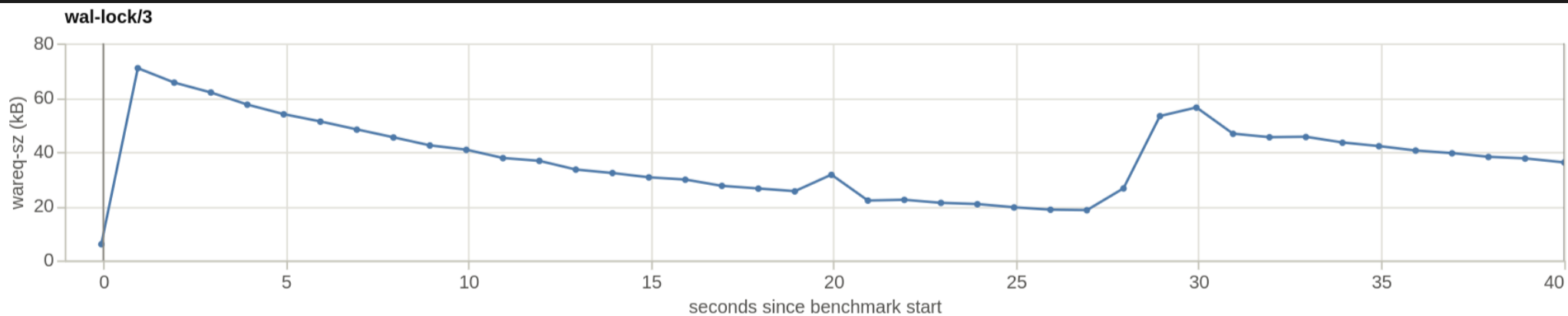
emits/clients=32



wal-lock/3







system IO statistics, from `/proc/diskstats`, e.g. via `iostat`; alternatively from cloud provider's stats

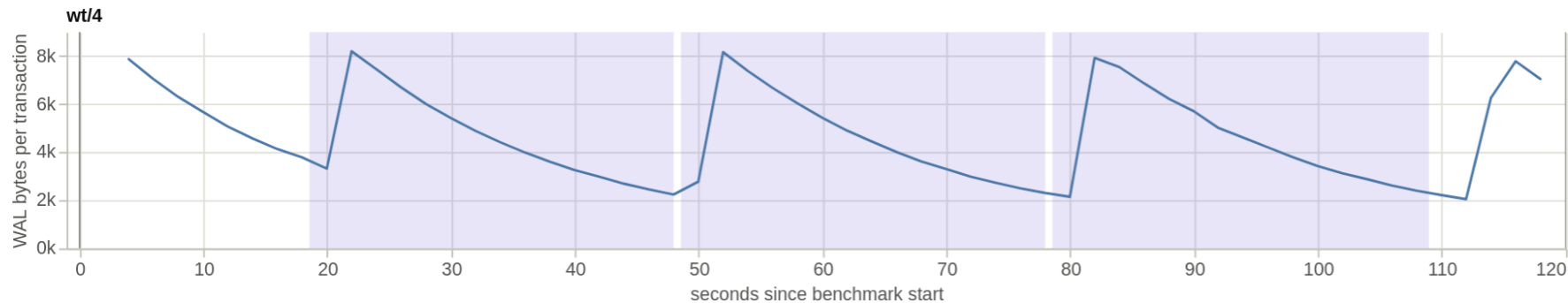
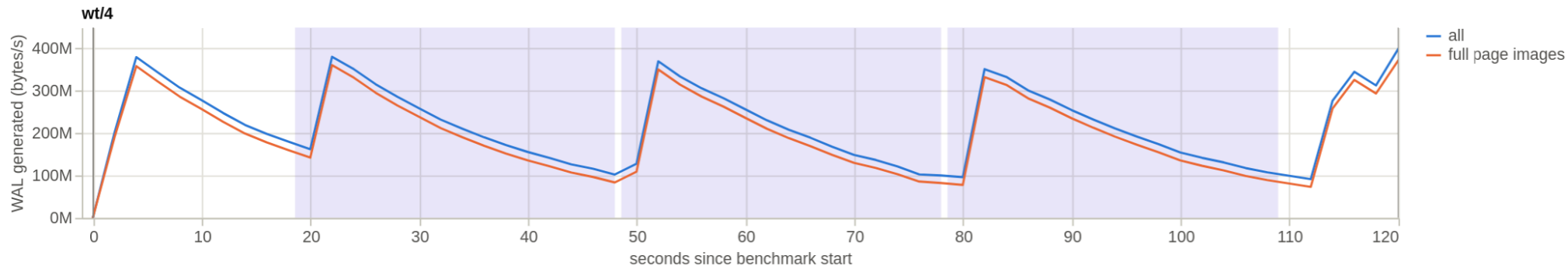
# WAL Latency: What now?

- larger transactions
- **more** concurrency
- selectively use async commits
- lower latency storage
- split WAL and data storage onto different filesystems
  - ext4's data=writeback can sometimes help

WAL Volume

## WAL

mean over 2 s buckets

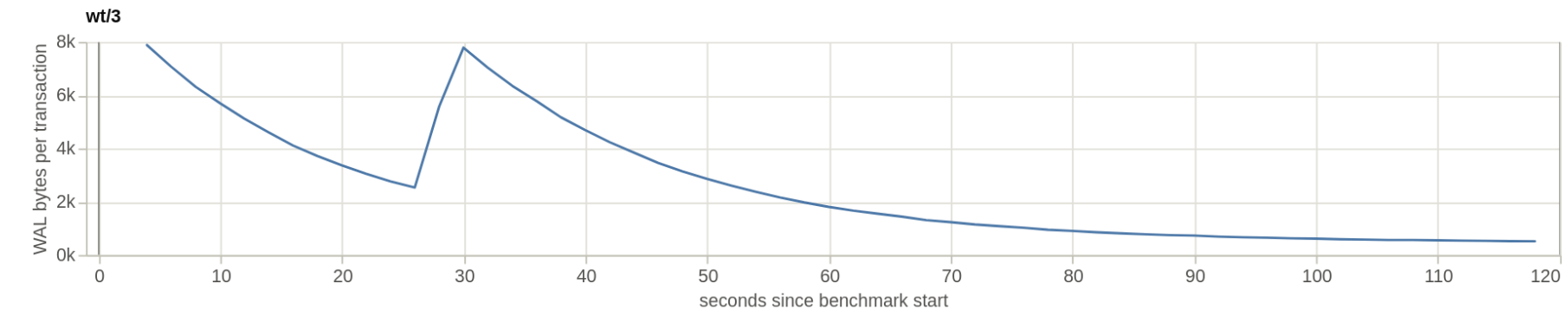
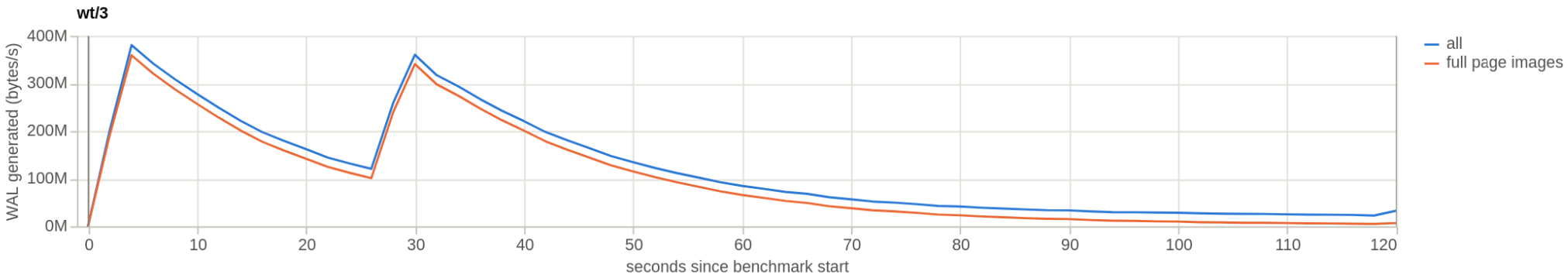


pgbench scale 500, 32 clients, checkpoint\_timeout=30s

statistics from pg\_stat\_wal, pg\_stat\_database

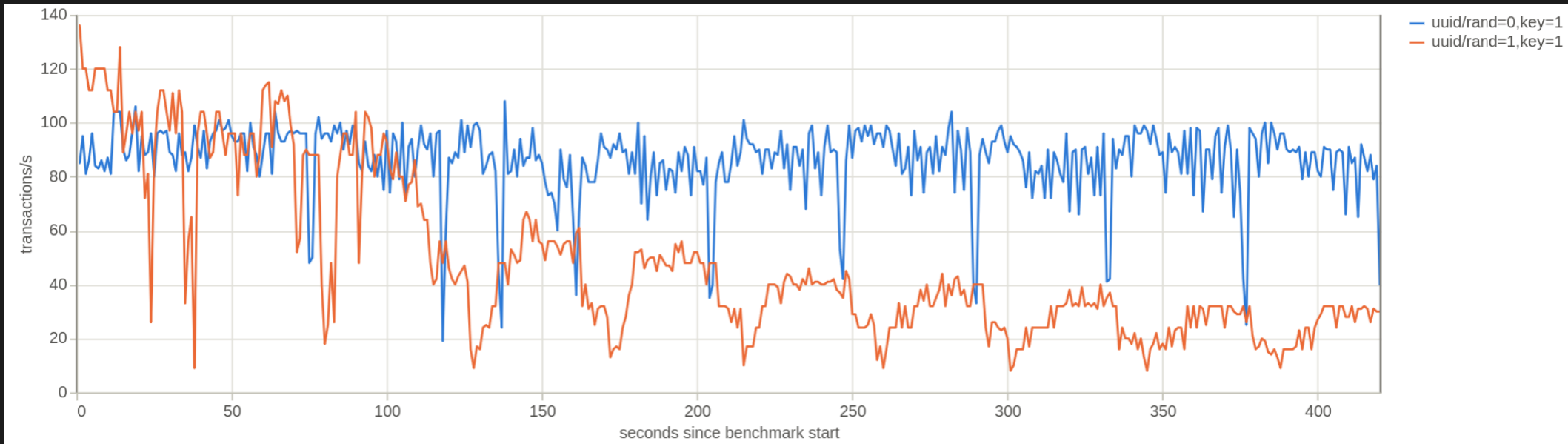
# WAL

mean over 2 s buckets;



pgbench scale 500, 32 clients, checkpoint\_timeout=300s

```
CREATE TABLE test_insert(  
    key uuid primary key,  
    data text default repeat(random()::text, 5)  
);
```

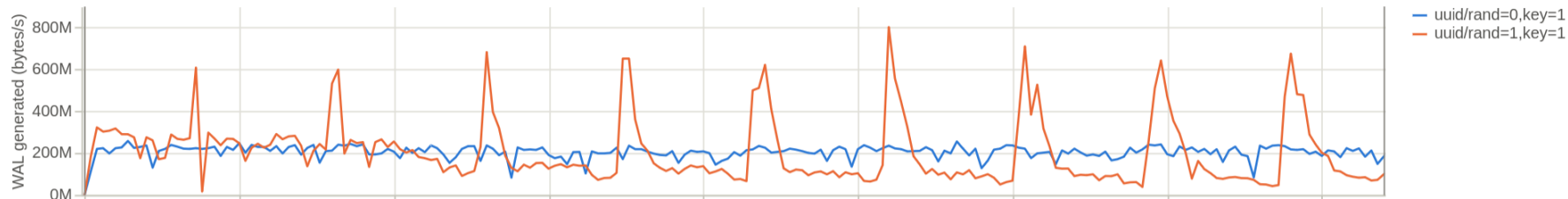


## WAL

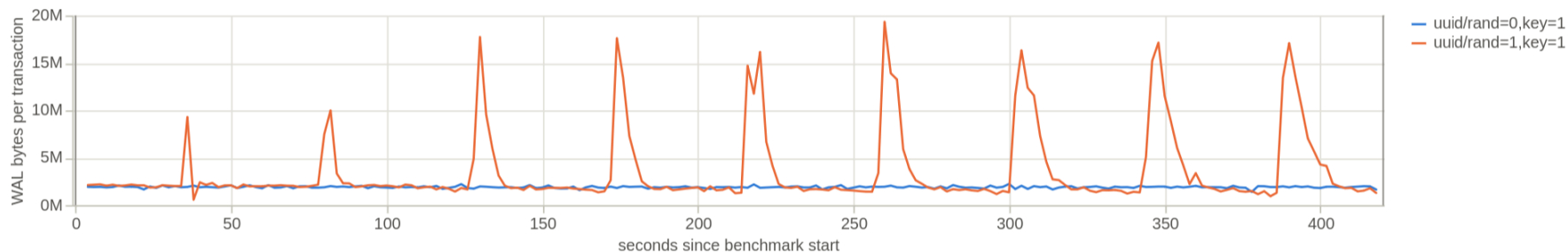
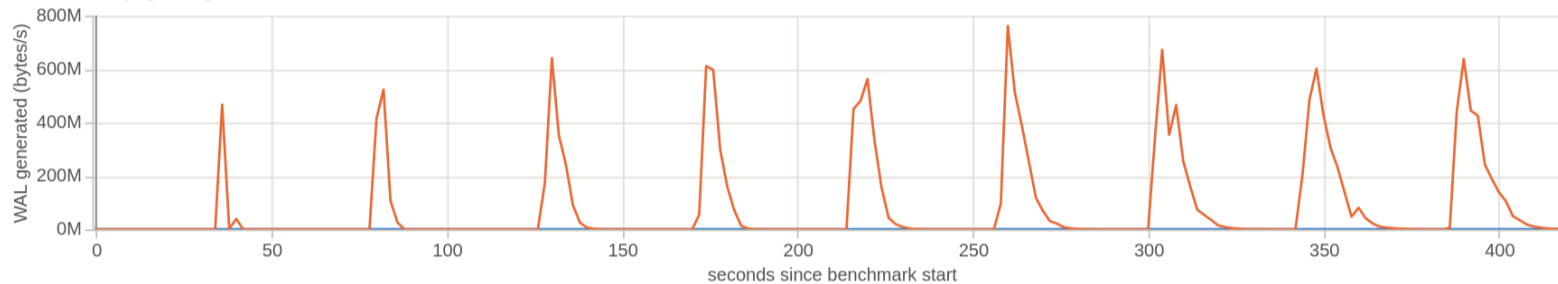
mean over 2 s buckets; backends report stats at most once per second and on exit, so the first bucket is low and the last high; transactions: commits + rollbacks in pg\_stat\_database; WAL writes and syncs from pg\_stat\_io; to



### all



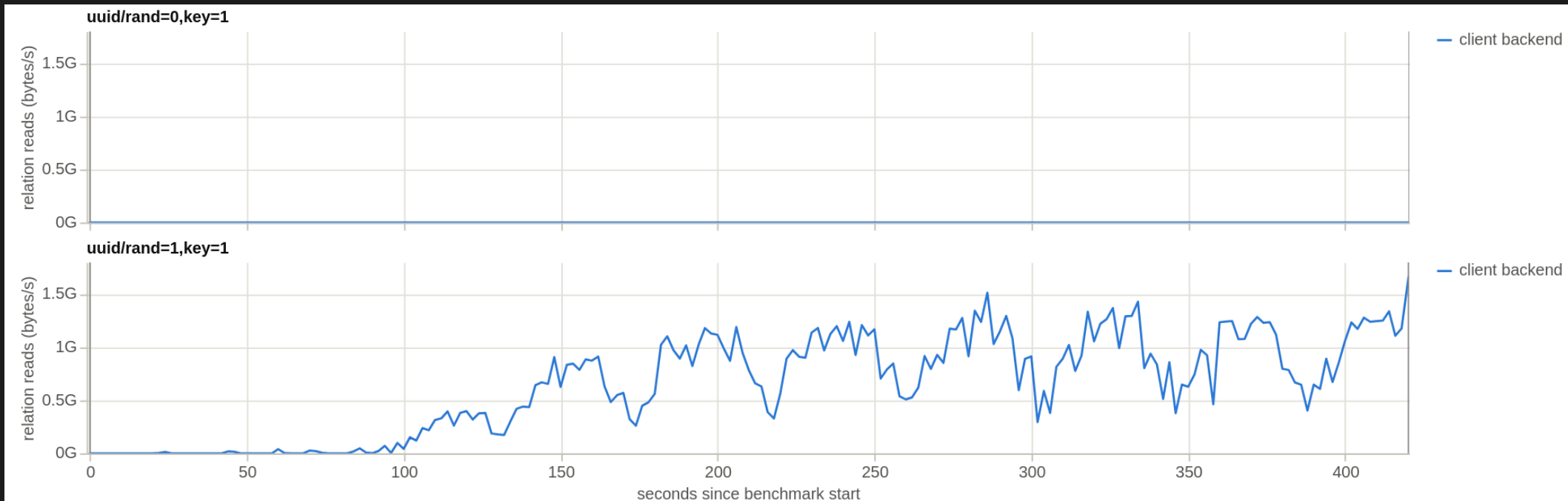
### full page images



# WAL Volume: What Now?

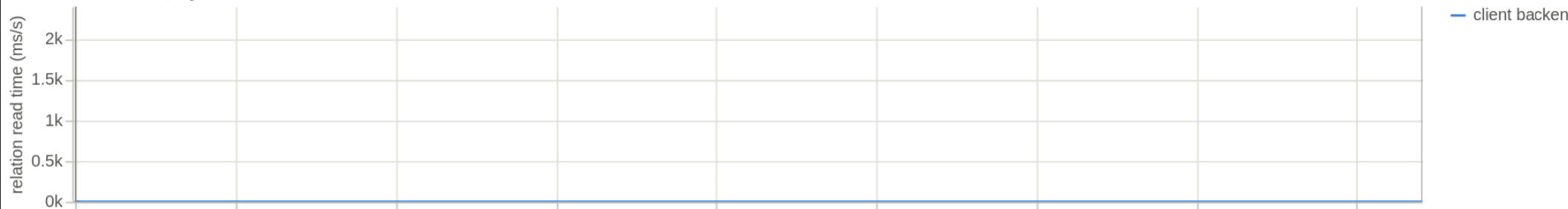
- longer checkpoint\_timeout
- wal\_compression=lz4 / zstd / off
- fewer indexes on random columns
  - don't use things like uuidv4 if at all possible
  - doesn't matter for small indexes
- fewer random changes
- bulk inserts
- schema improvements
- pg\_waldump -stats / pg\_walinspect extension

Read IOPS

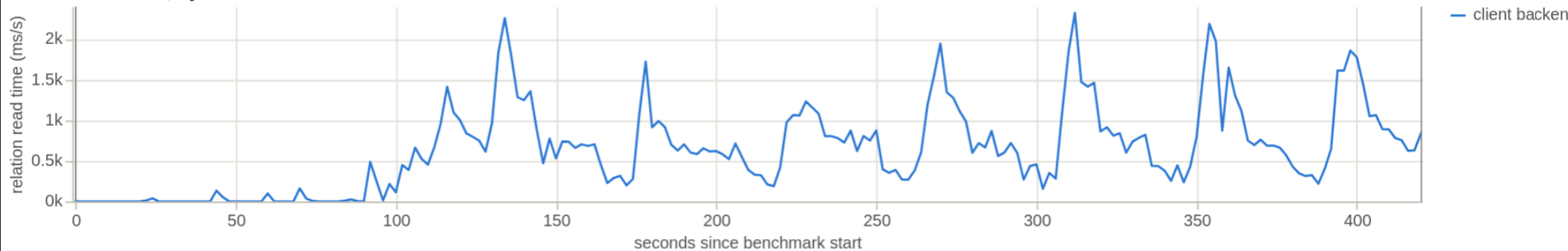


statistics from `pg_stat_io`, with `track_io_timing=on`, `track_wal_io_timing` can also be useful

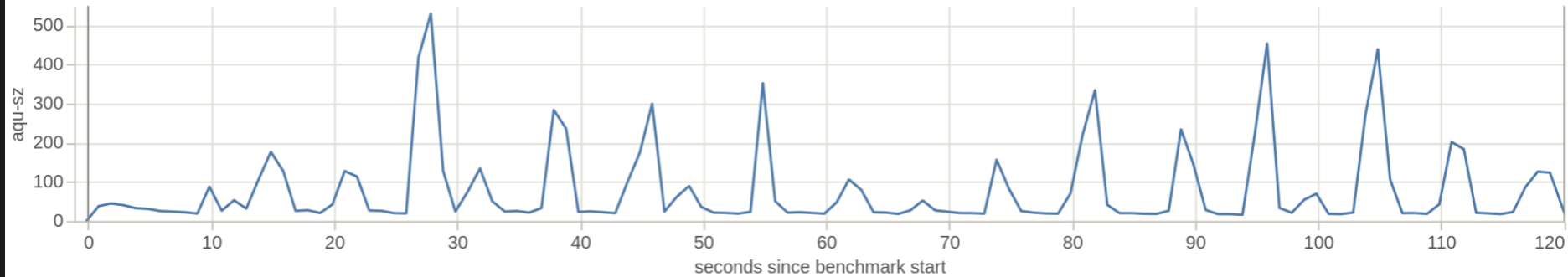
uuid/rand=0,key=1



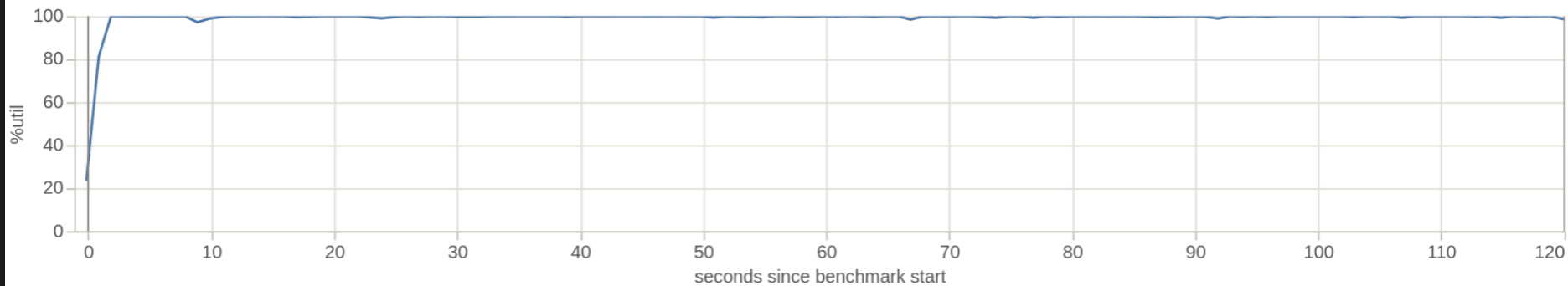
uuid/rand=1,key=1



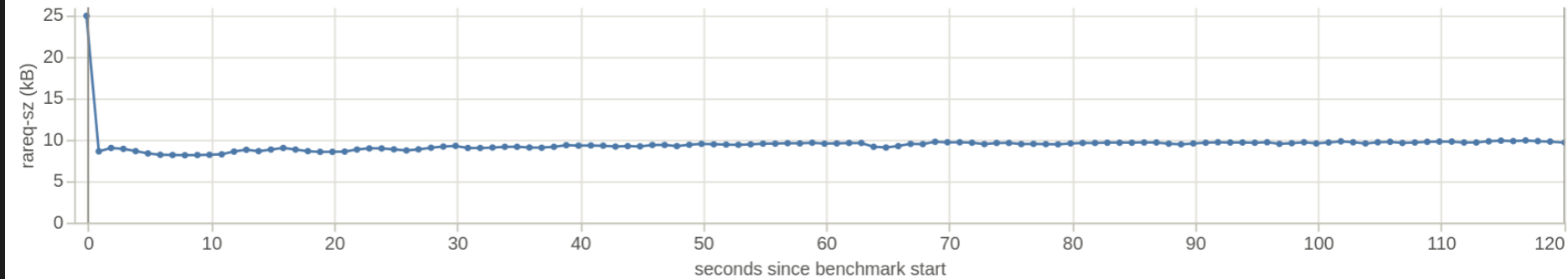
pgbenchro/scale=2000,clients=64



pgbenchro/scale=2000,clients=64



pgbenchro/scale=2000,clients=64



# Read IOPS bound: What now?

- Faster storage
- Larger shared buffers (or OS cache)
- Larger IOs
- More prefetching & combining
  - Query formulation allowing prefetching
  - PG & OS config
- Change Schema

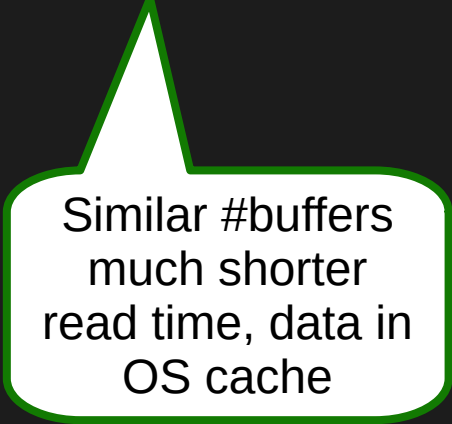
Caveat: PG read  $\neq$  device read

```
-- no parallelism, simpler analysis
SET max_parallel_workers_per_gather = 0;
-- no prefetching, can hide IO boundedness
SET effective_io_concurrency = 0;

-- COSTS OFF just for smaller display
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF) SELECT SUM(abalance) FROM
pgbench_accounts ;
```

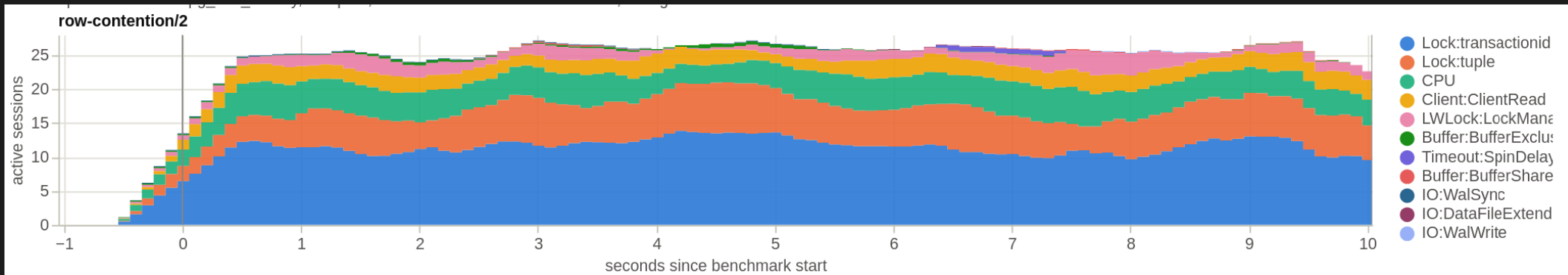
Aggregate (actual time=24529.876..24529.877 rows=1.00 loops=1)  
 Buffers: shared hit=2167 read=3276522  
 I/O Timings: shared read=16054.174  
 -> Seq Scan on pgbench\_accounts (actual time=2.x..**20552.x** ...)  
 Buffers: shared hit=2167 **read=3276522**  
 I/O Timings: shared read=**16054.174**  
Planning Time: 0.093 ms  
Execution Time: 24529.915 ms

Aggregate (actual time=10080.023..10080.024 rows=1.00 loops=1)  
 Buffers: shared hit=2263 read=3276426  
 I/O Timings: shared read=2154.384  
 -> Seq Scan on pgbench\_accounts (actual  
time=0.127..**6255.533** ...)  
 Buffers: shared hit=2263 **read=3276426**  
 I/O Timings: shared **read=2154.384**  
Planning Time: 0.139 ms  
Execution Time: 10080.059 ms



Similar #buffers  
much shorter  
read time, data in  
OS cache

# Row Contention



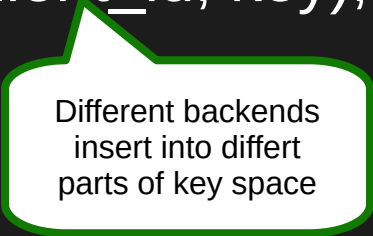
pgbench scale 5 synchronous\_commit=off, 32 clients

# Row Contention: What now?

- Avoid unnecessary/redundant updates
- Normalize schema
- Analyze transaction duration
  - shorter xacts, shorter waits
  - shorter xacts might cause slowdowns too

# Buffer Contention

```
CREATE TABLE test_insert_k2(  
  key uuid not null,  
  client_id int not null,  
  data text default repeat(random()::text, 5)  
);  
create unique index on test_insert(client_id, key);
```



Different backends  
insert into different  
parts of key space

Lots of Buffer contention,  
all backends insert into  
same index page

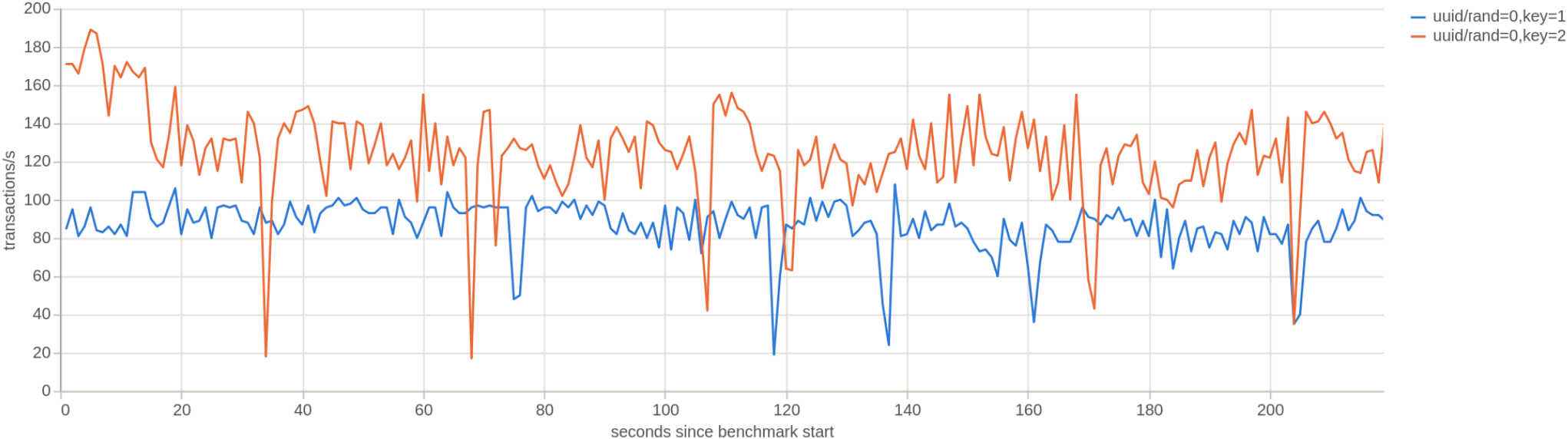


Buffer:BufferLock{Exclusive,Shared,ShareExclusive} ( $\geq 19$ )  
LWLock:BufferContent ( $\leq 18$ )

WAL & CPU  
bound

transactions/s

gray rules: benchmark start and ends



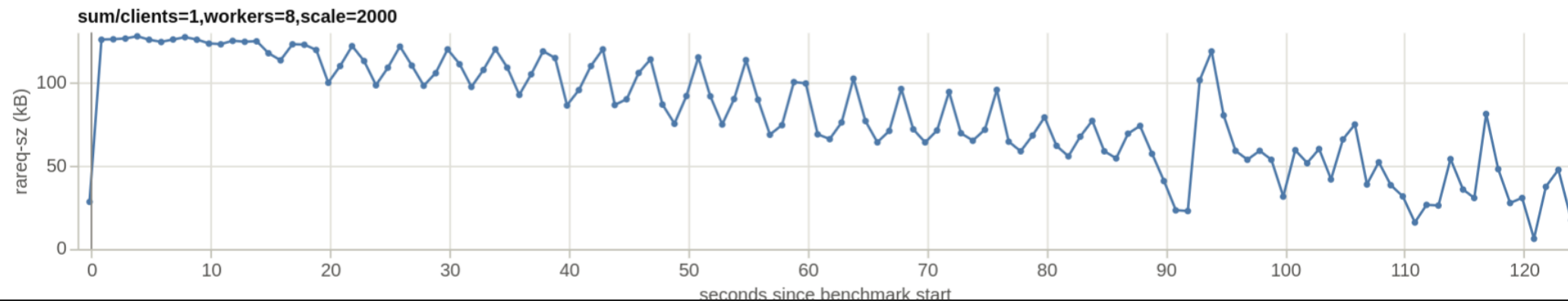
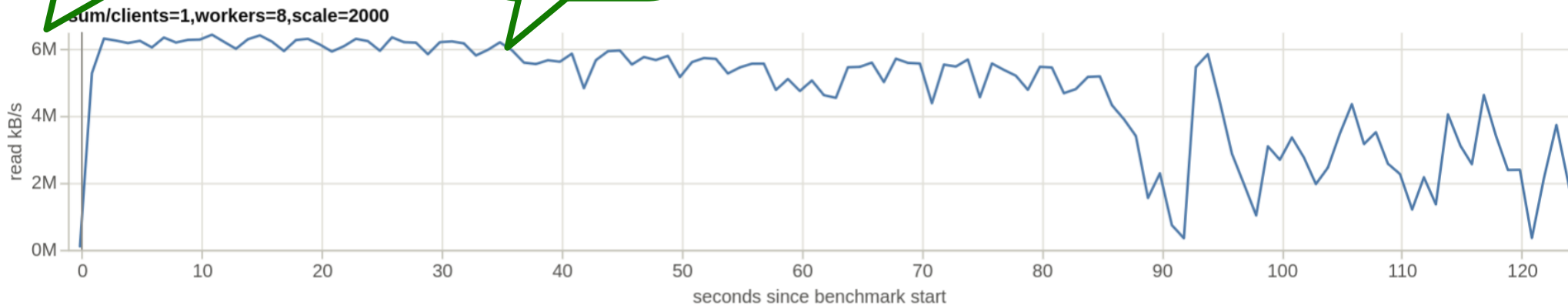
# Buffer Contention: What now?

- Uhhhm.
- ...
- Make postgres better?
- **Increase** distribution of index index insertions
- Not super common, luckily

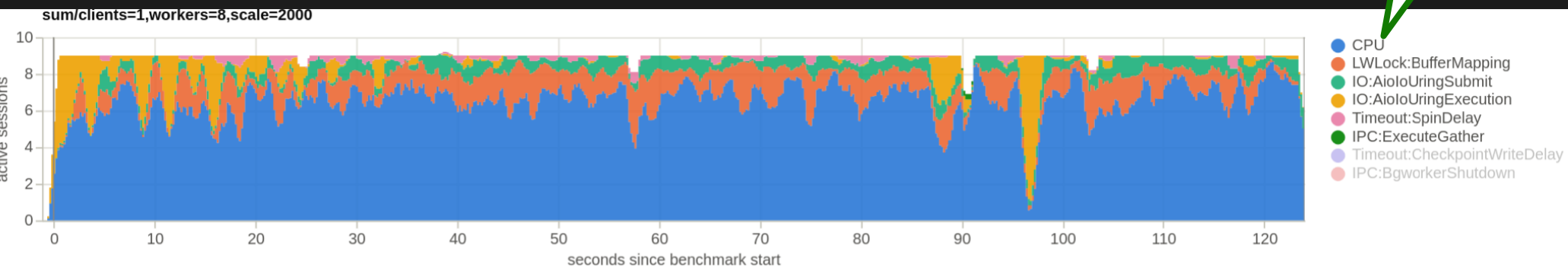
# IO Bandwidth

~6GB/s, device bandwidth  
higher than common

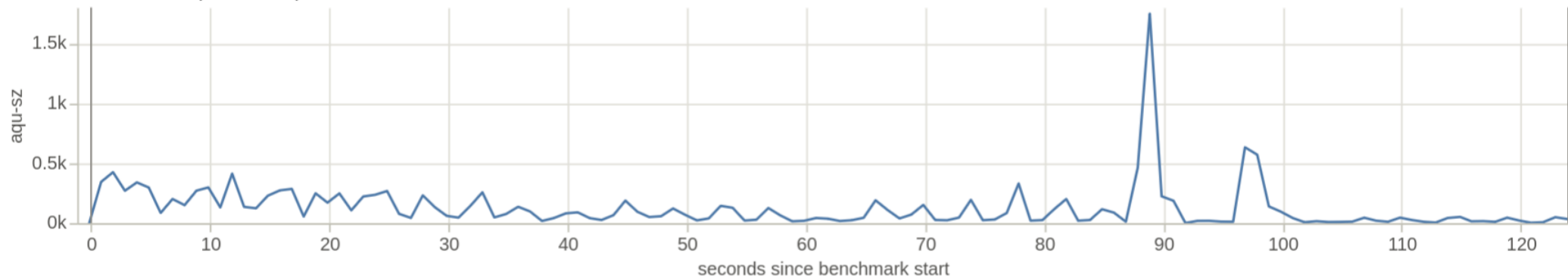
Repeat runs  
have some data  
cached



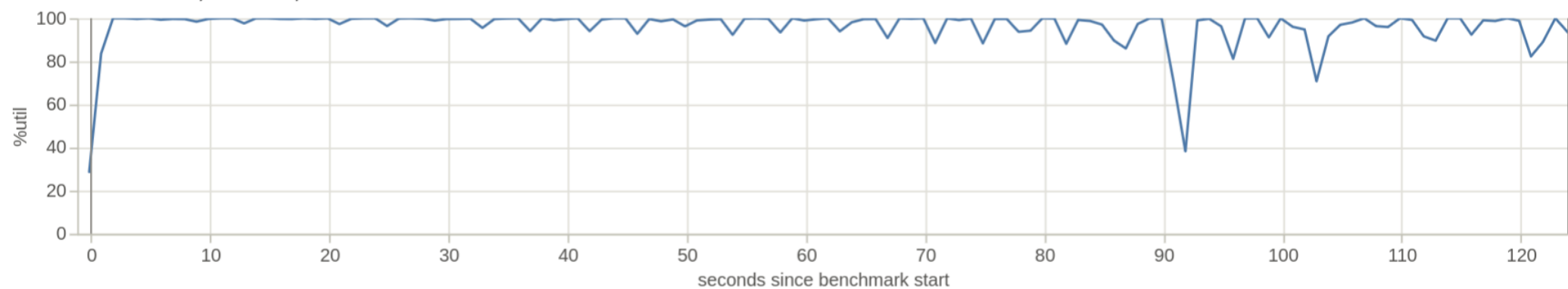
Mix of IO and  
CPU bound



sum/clients=1,workers=8,scale=2000



sum/clients=1,workers=8,scale=2000



# IO Bandwidth bound: What now?

- Faster storage
- Larger shared buffers (or OS cache)
- Change Schema
- Increase `checkpoint_timeout`

# Feedback Link

