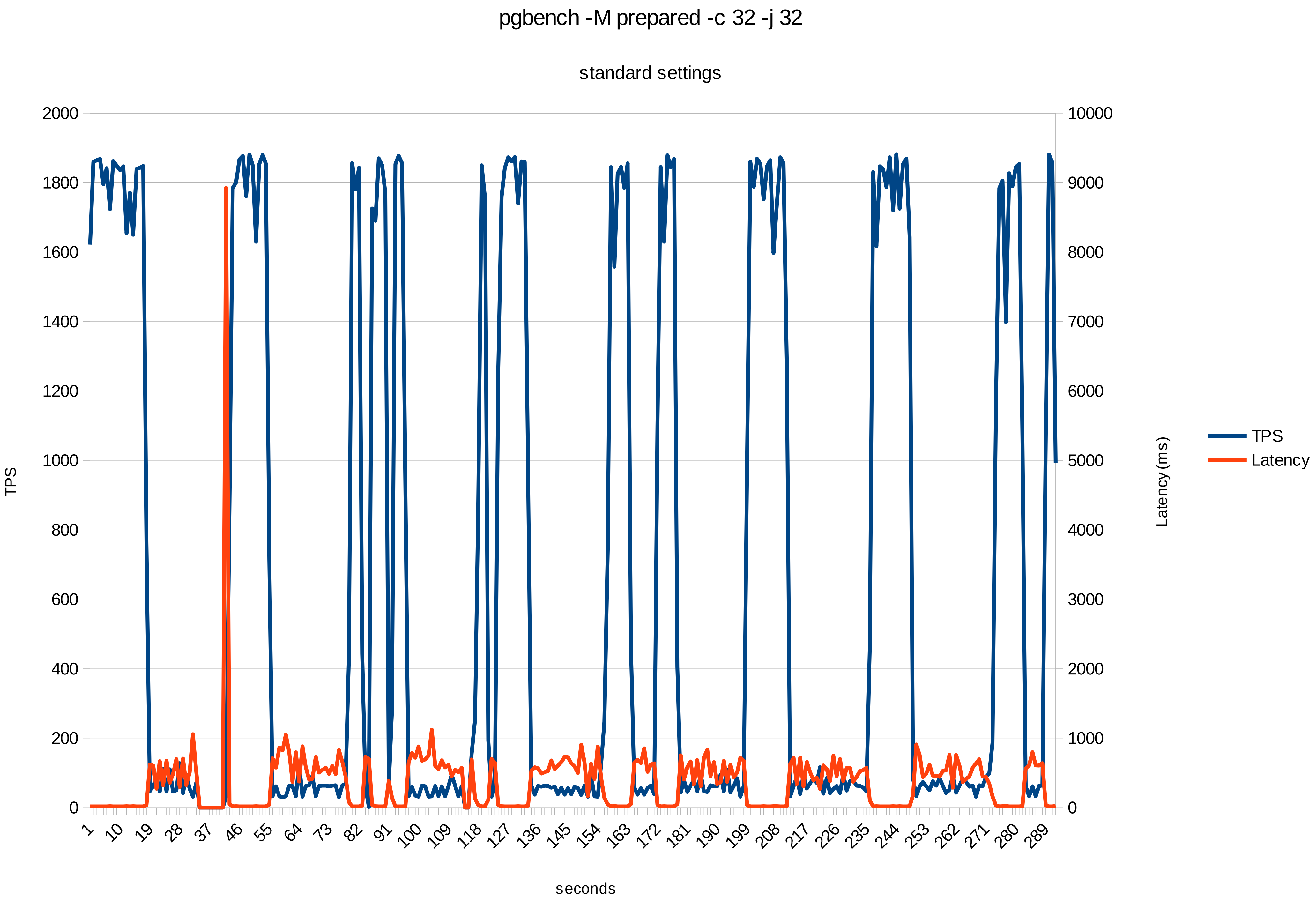
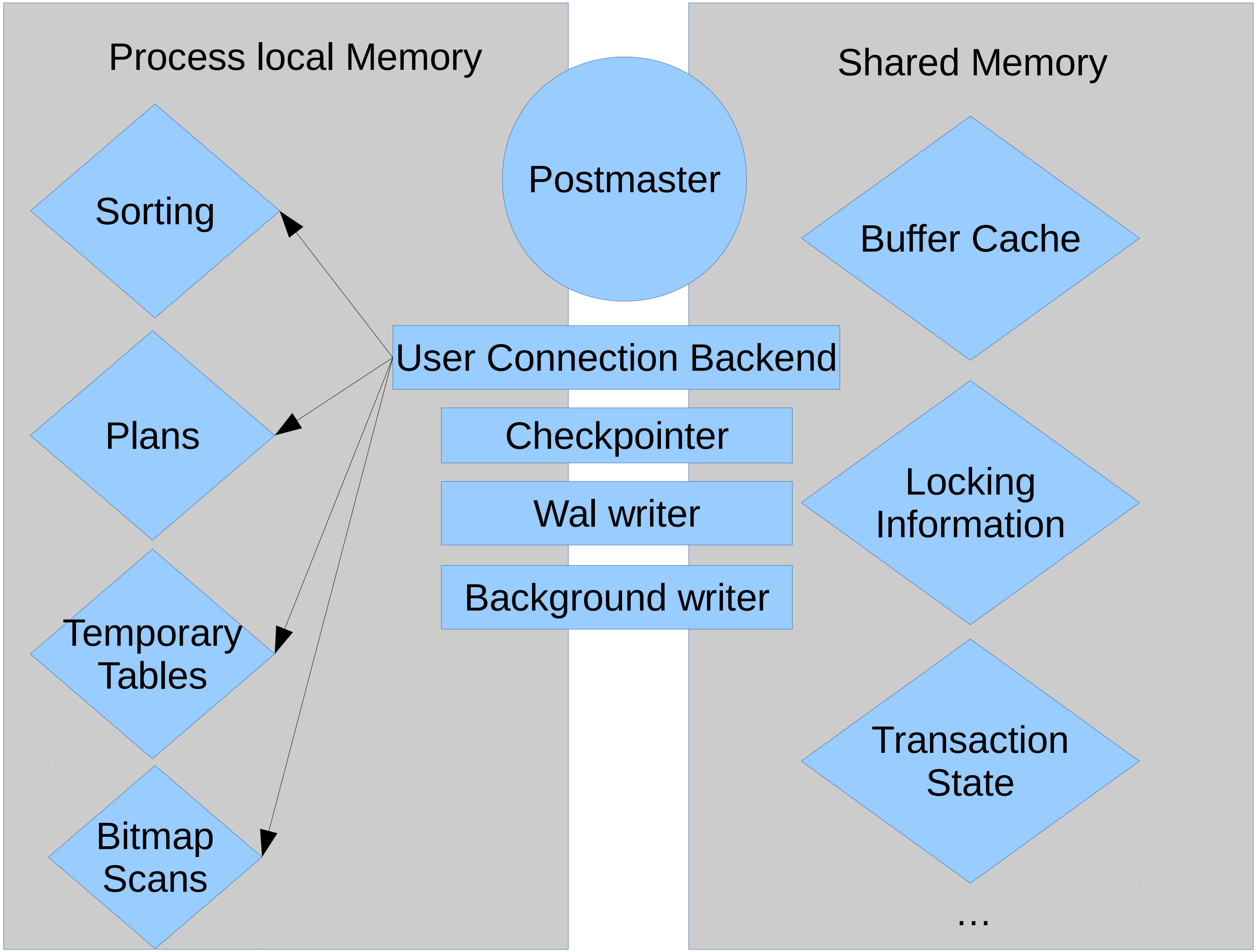
# Postgres' IO
# Architecture, Tuning, Problems

Andres Freund
PostgreSQL Developer & Committer
Citus Data – citusdata.com - @citusdata

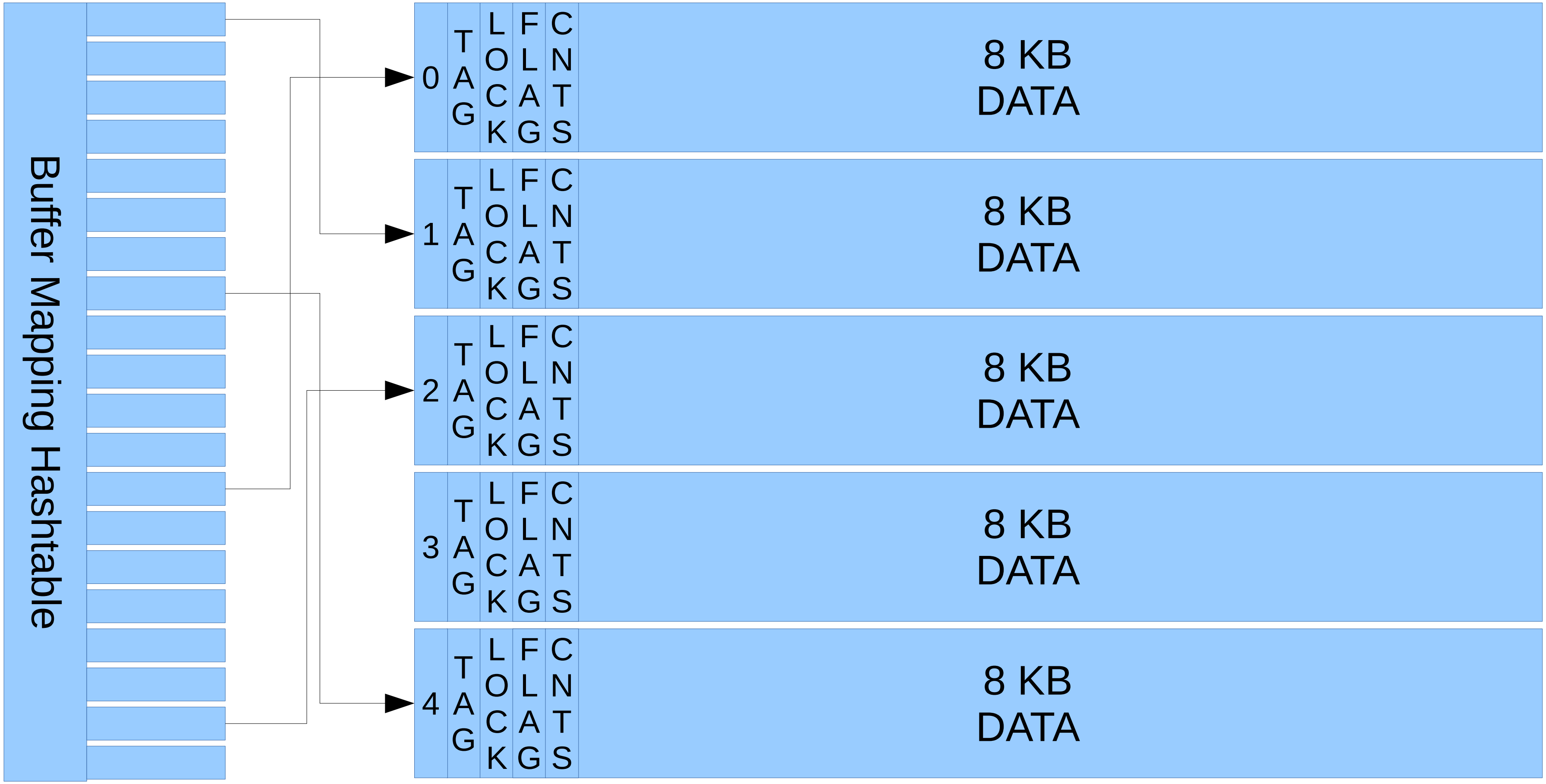
http://anarazel.de/talks/berlin-meetup-2016-01-26/io.pdf


pgbench -M prepared -c 32 -j 32
standard settings
TPS
Latency (ms)
seconds
0
200
400
600
800
1000
1200
1400
1600
1800
2000
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
1
10
19
28
37
46
55
64
73
82
91
100
109
118
127
136
145
154
163
172
181
190
199
208
217
226
235
244
253
262
271
280
289
TPS
Latency

# Memory Architecture

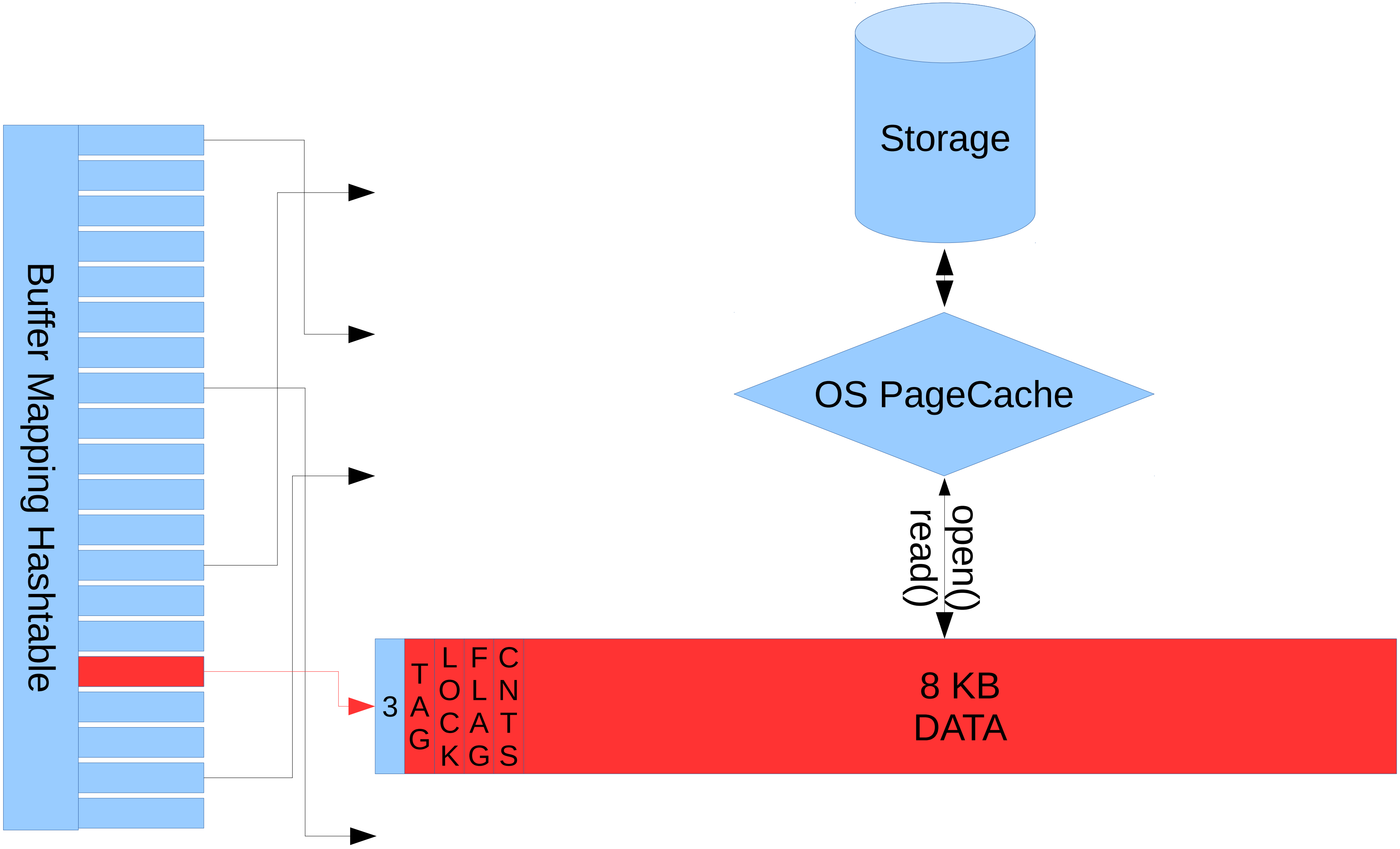


# Shared Buffers



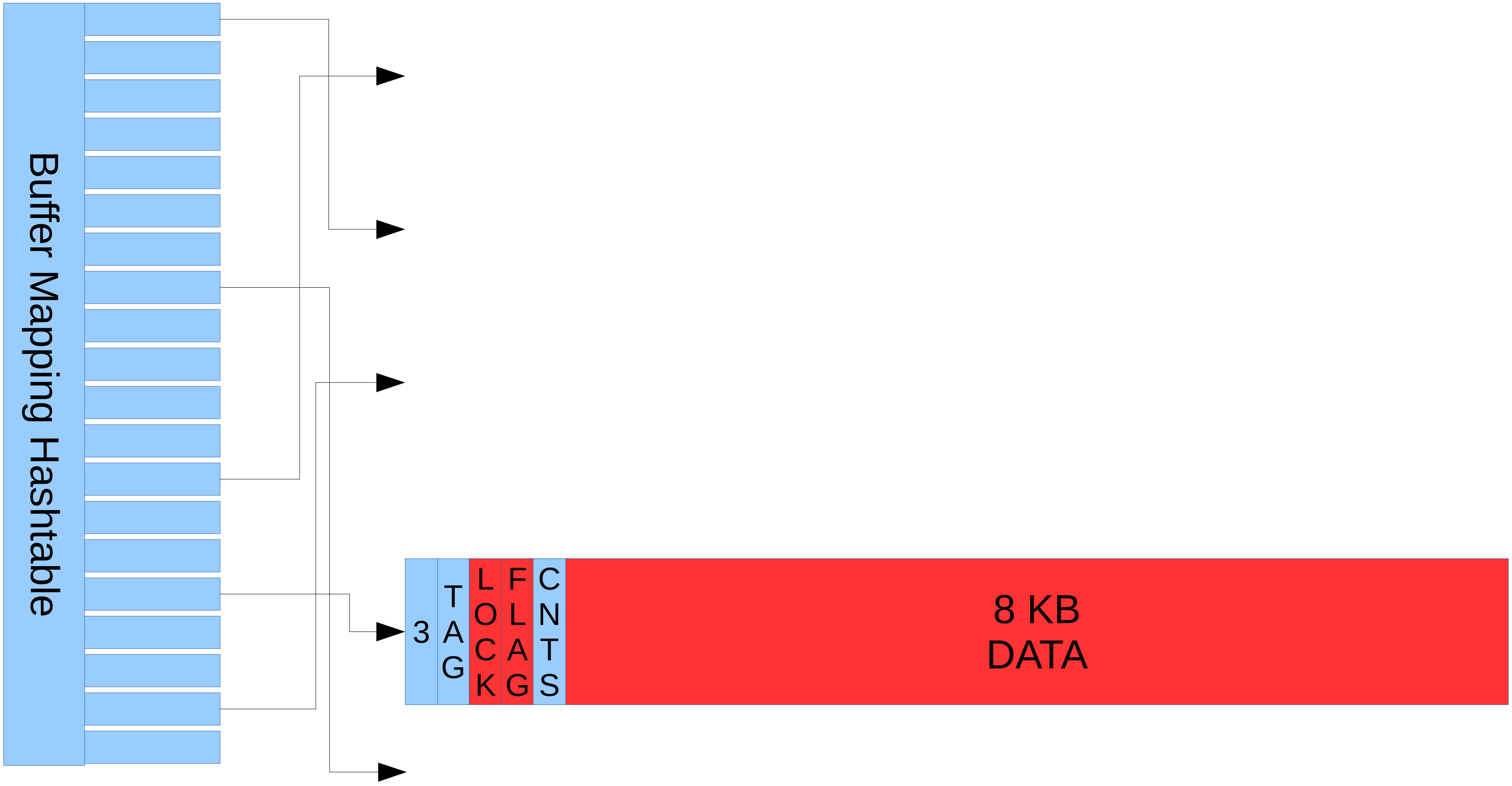
citusdata

# Reading Data

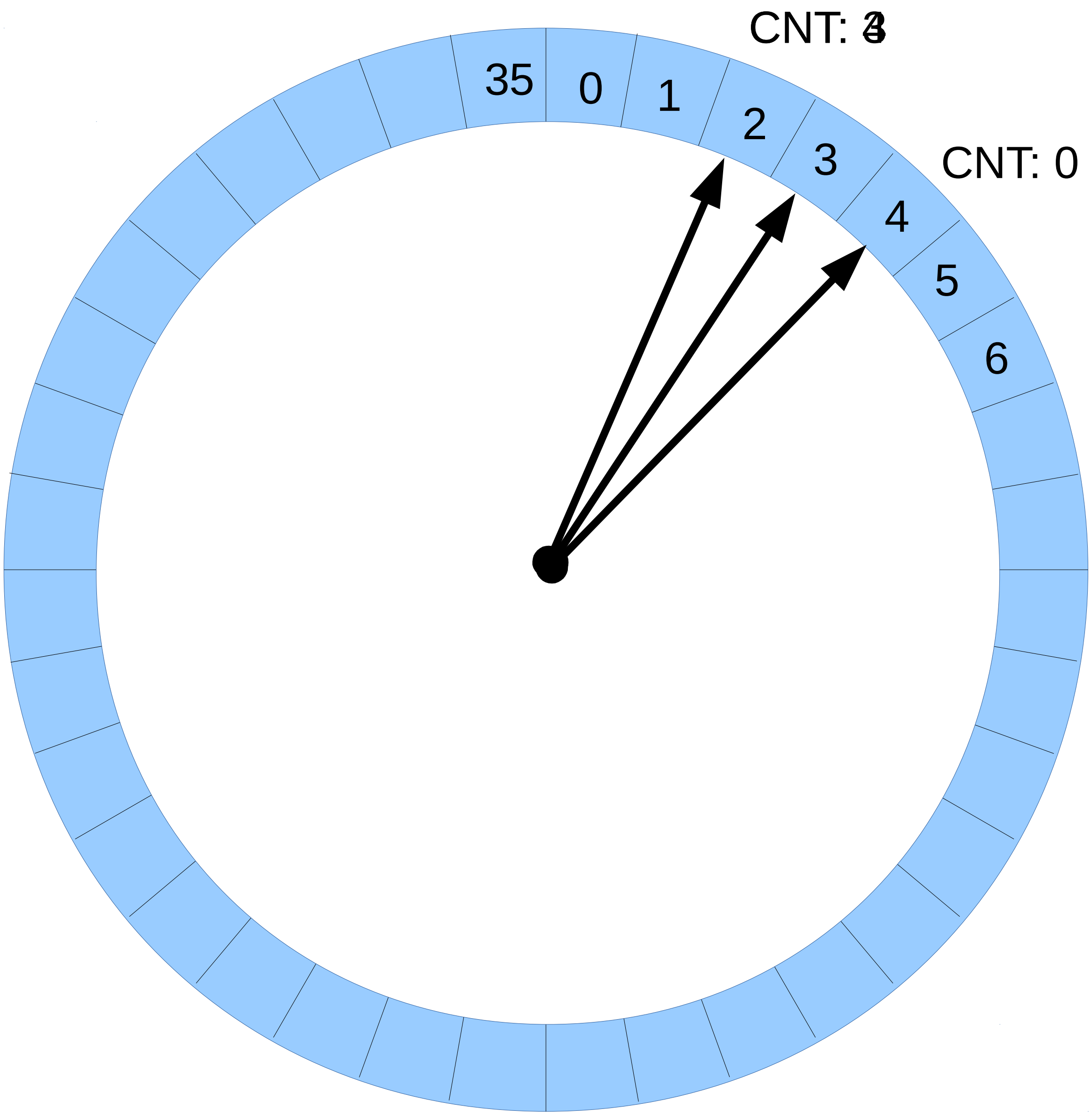


citusdata

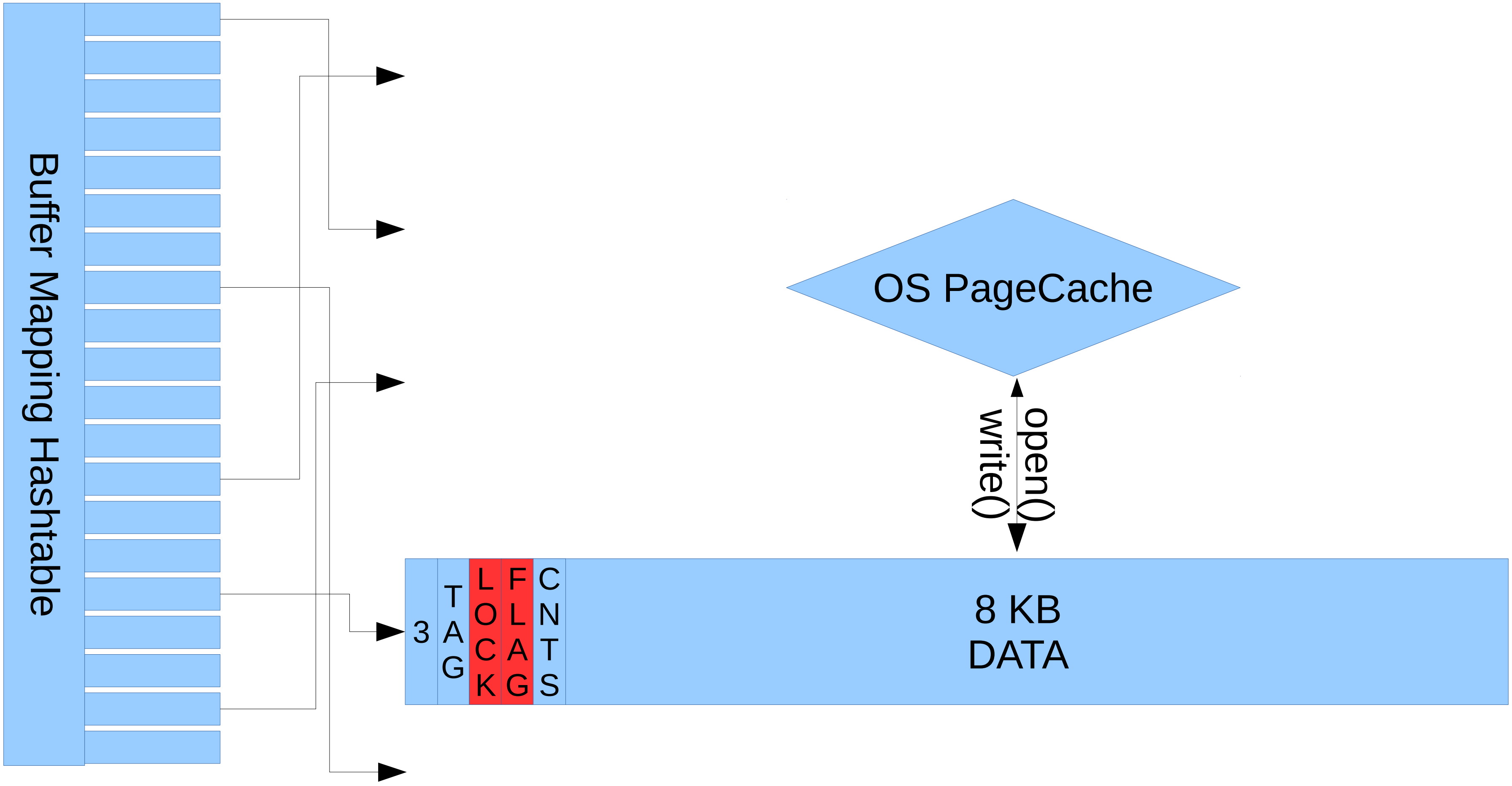
# Writing Data



# Clock-Sweep



# Writing Data Out



citusdata

# Need for WAL logging

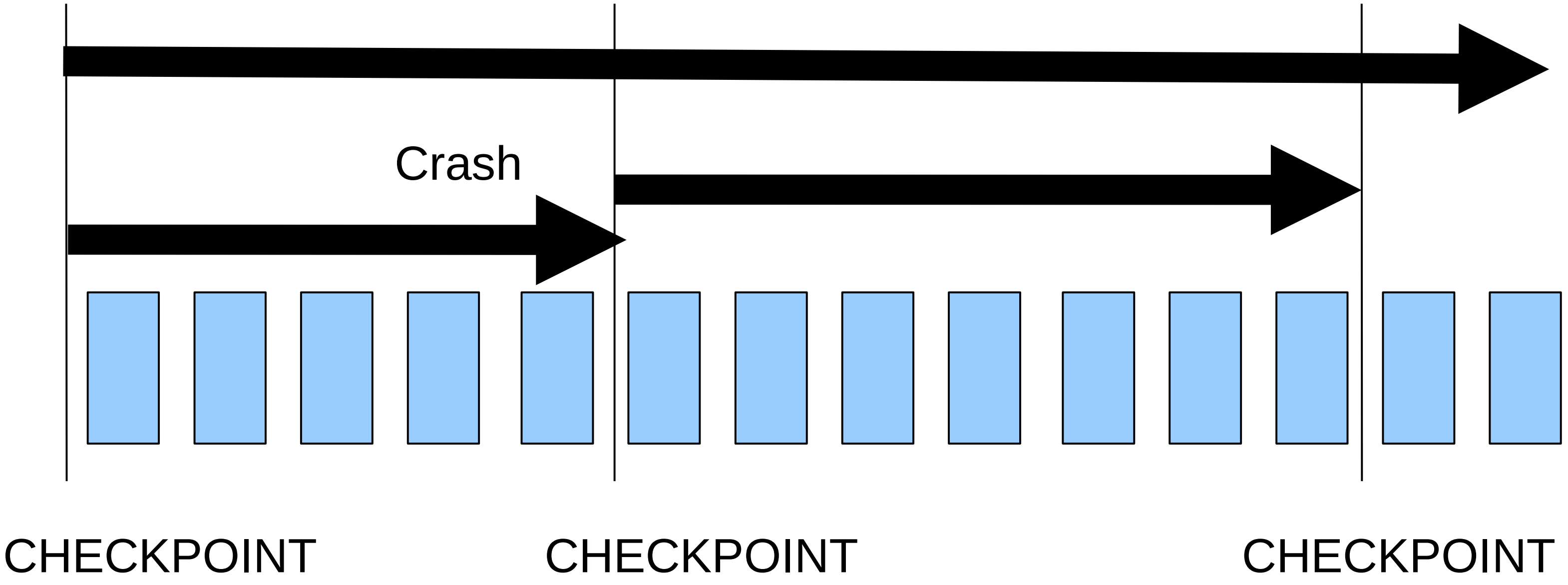
| | | | | | |
|---|---|---|---|---|---|
| 0 | TAG | LOCK | FLAG | CNTS | 8 KB DATA |
| 1 | TAG | LOCK | FLAG | CNTS | 8 KB DATA |
| 2 | TAG | LOCK | FLAG | CNTS | 8 KB DATA |
| 3 | TAG | LOCK | FLAG | CNTS | 8 KB DATA |
| 4 | TAG | LOCK | FLAG | CNTS | 8 KB DATA |

# WAL Logging

```
BEGIN;
INSERT INTO mytbl VALUES (…);
COMMIT;
```

```
rmgr: Heap        tx:  344576442, lsn: 317/A2333ED8, desc: INSERT off 3 blkref
< actually modify heap >
rmgr: Btree       tx:  344576442, lsn: 317/A2333F18, desc: INSERT_LEAF off 2
< actually modify btree >
rmgr: Transaction tx:  344576442, lsn: 317/A2333F58, desc COMMIT 2016-01-25 15:17:30
< actually modify transaction >
```

# Recovery & Checkpoints



# Checkpoints

1)Remember current position in WAL

2)Do some boring things

3)Write out all dirty buffers

4)Fsync all files modified since last checkpoint

5)Write checkpoint WAL record, pg_control etc.

6)Remove old WAL

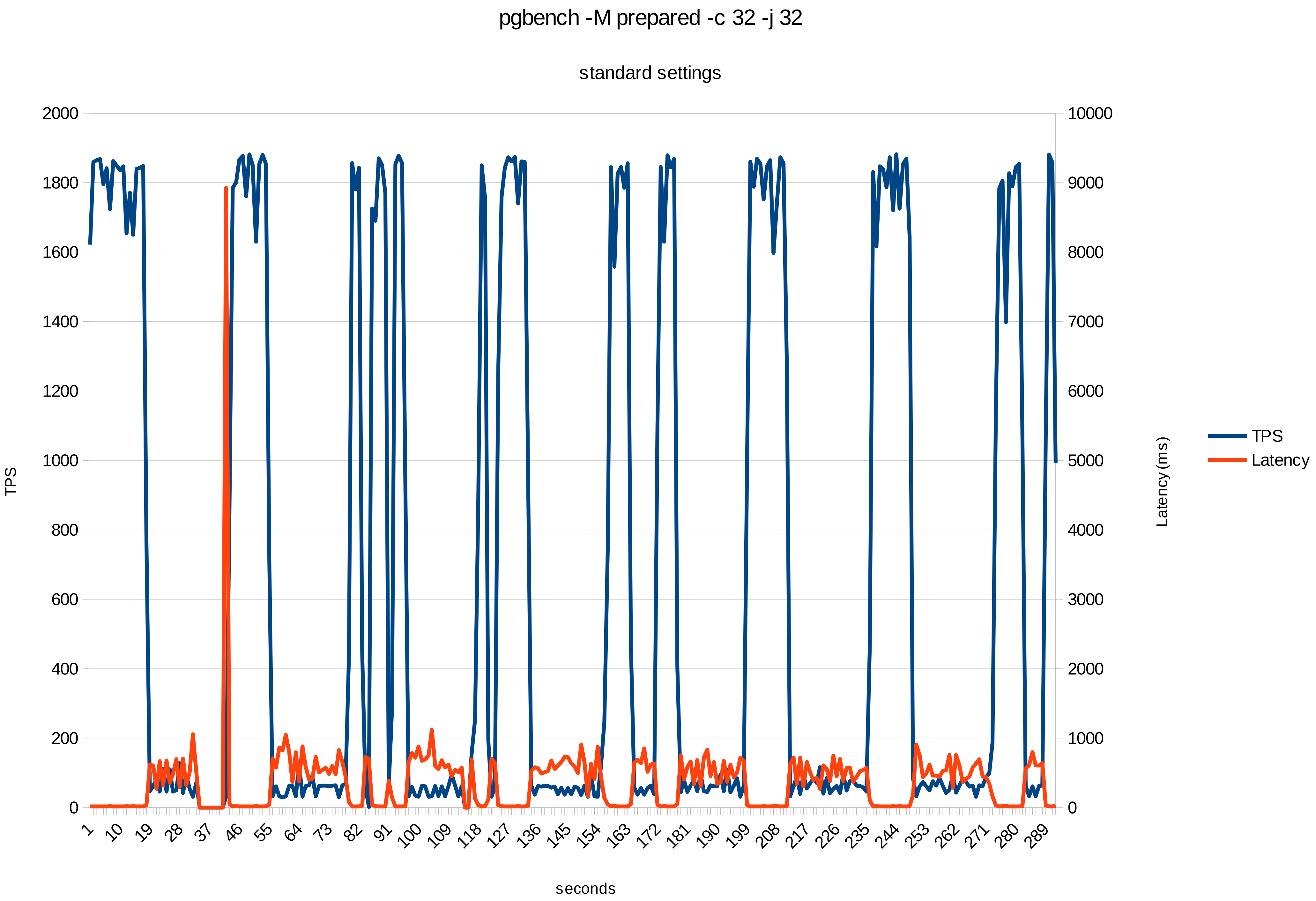
# Triggering Checkpoints

- checkpoint_timeout = 5min
  - LOG:  checkpoint starting: time
- checkpoint_segments = 3 / max_wal_size = 1GB
  - LOG:  checkpoint starting: xlog
  - LOG:  checkpoints are occurring too frequently (2 seconds apart)
- shutdown
  - LOG:  checkpoint starting: shutdown immediate
- manually (CHECKPOINT;)

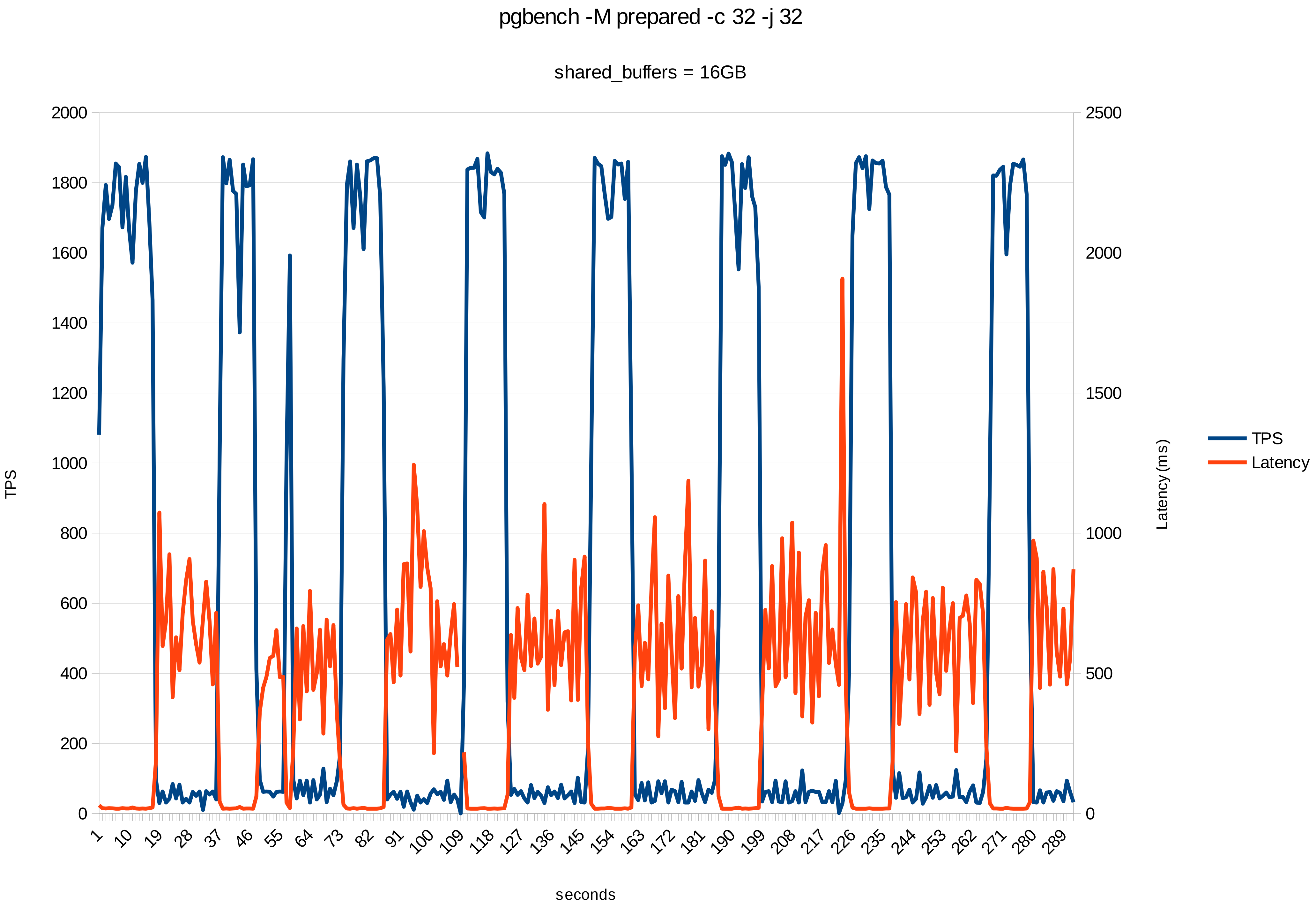
# Spreading Checkpoints

- checkpoint_completion_target = 0.5
- estimation based on
  - checkpoint_timeout
  - checkpoint_segments/max_wal_size
- Start the next checkpoint after checkpoint_completion_target * above_param

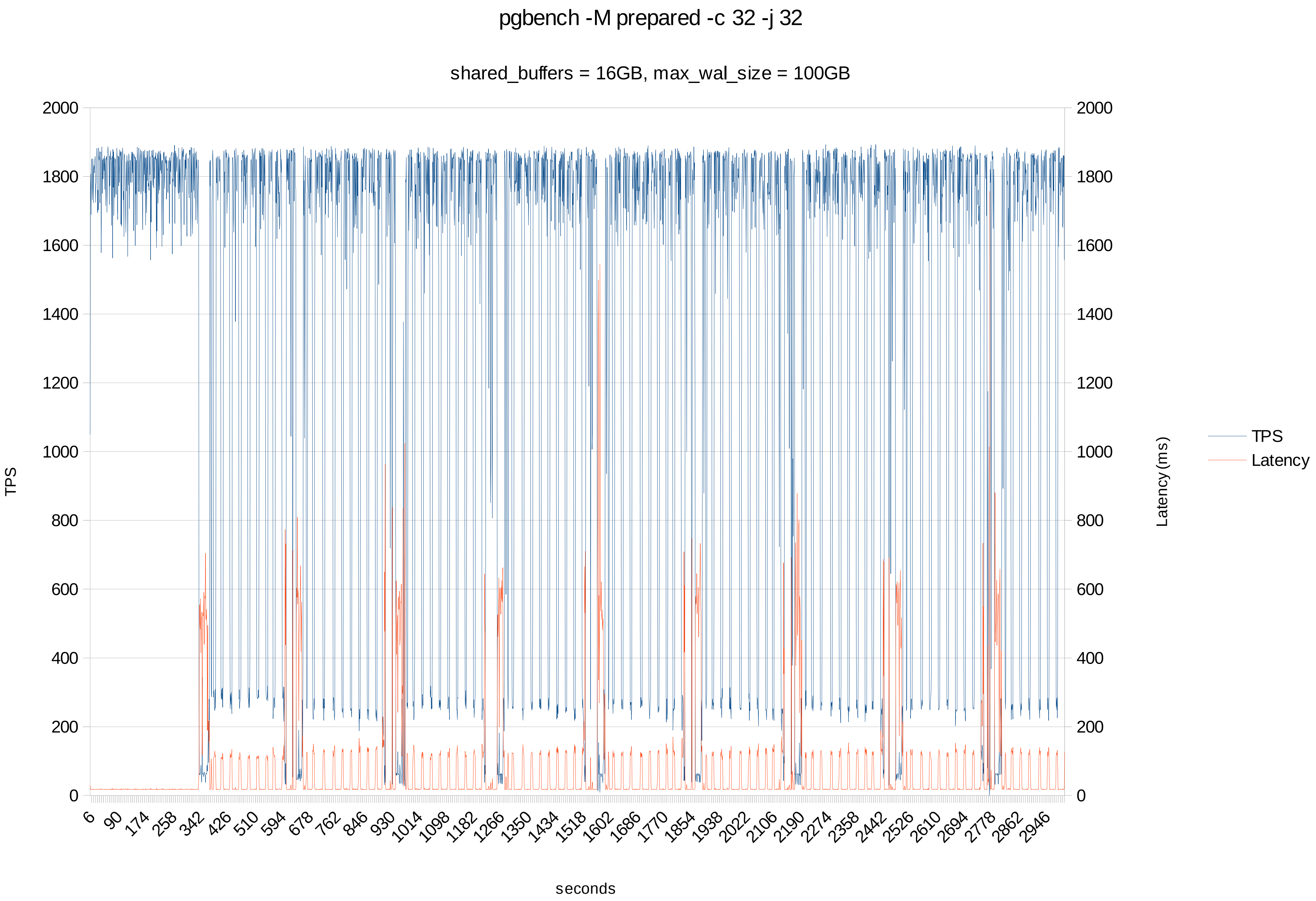
  has passed
- Try to keep pace

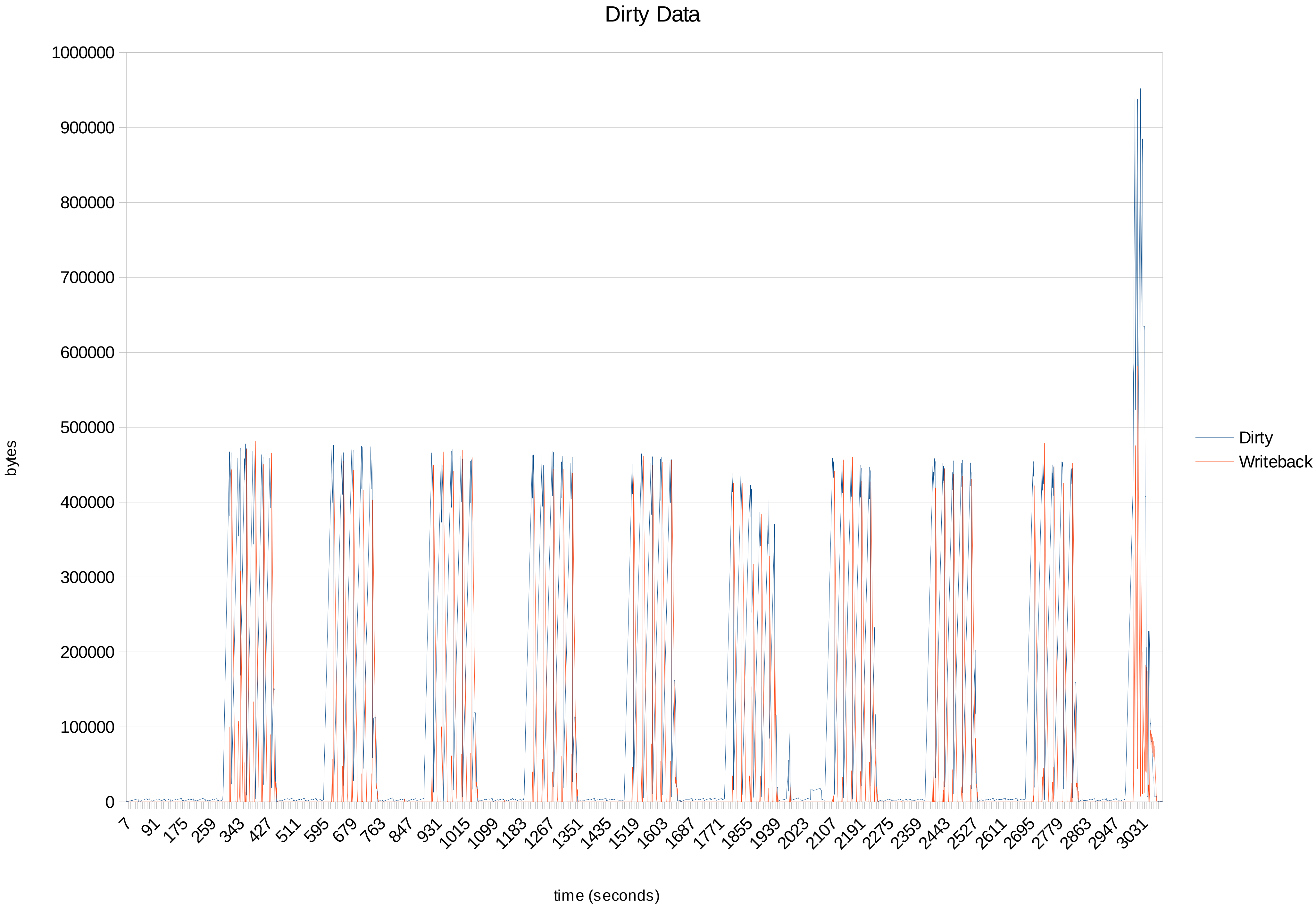


pgbench -M prepared -c 32 -j 32
standard settings
TPS
Latency (ms)
seconds
0
200
400
600
800
1000
1200
1400
1600
1800
2000
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
1
10
19
28
37
46
55
64
73
82
91
100
109
118
127
136
145
154
163
172
181
190
199
208
217
226
235
244
253
262
271
280
289
TPS
Latency


pgbench -M prepared -c 32 -j 32
shared_buffers = 16GB
TPS
2000
1800
1600
1400
1200
1000
800
600
400
200
0
Latency (ms)
2500
2000
1500
1000
500
0
1
10
19
28
37
46
55
64
73
82
91
100
109
118
127
136
145
154
163
172
181
190
199
208
217
226
235
244
253
262
271
280
289
seconds
TPS
Latency

citusdata

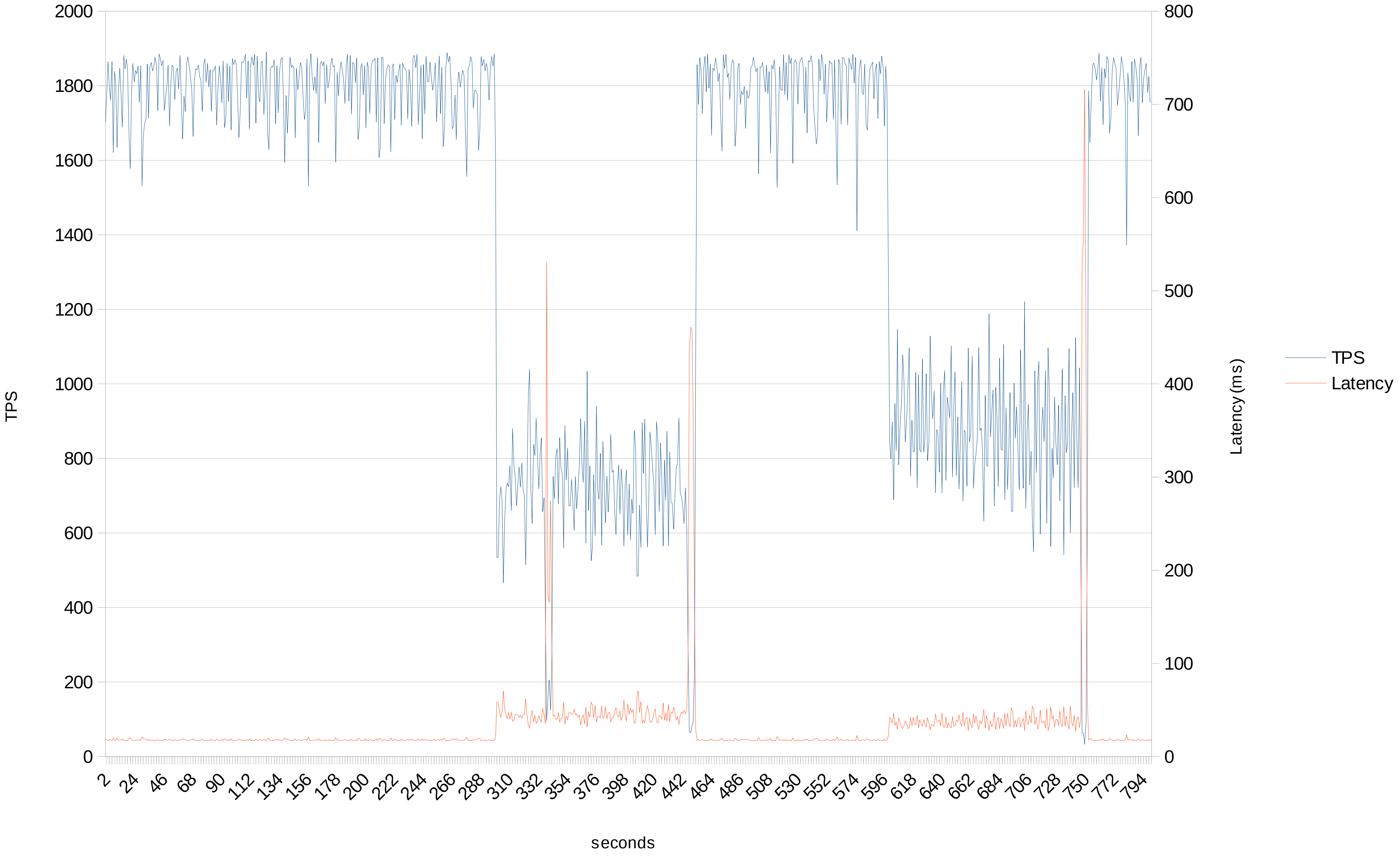

pgbench -M prepared -c 32 -j 32
shared_buffers = 16GB, max_wal_size = 100GB
TPS
Latency (ms)
seconds
2000
1800
1600
1400
1200
1000
800
600
400
200
0
6
90
174
258
342
426
510
594
678
762
846
930
1014
1098
1182
1266
1350
1434
1518
1602
1686
1770
1854
1938
2022
2106
2190
2274
2358
2442
2526
2610
2694
2778
2862
2946
TPS
Latency



Dirty Data
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
bytes
7
91
175
259
343
427
511
595
679
763
847
931
1015
1099
1183
1267
1351
1435
1519
1603
1687
1771
1855
1939
2023
2107
2191
2275
2359
2443
2527
2611
2695
2779
2863
2947
3031
time (seconds)
Dirty
Writeback

pgbench -M prepared -c 32 -j 32

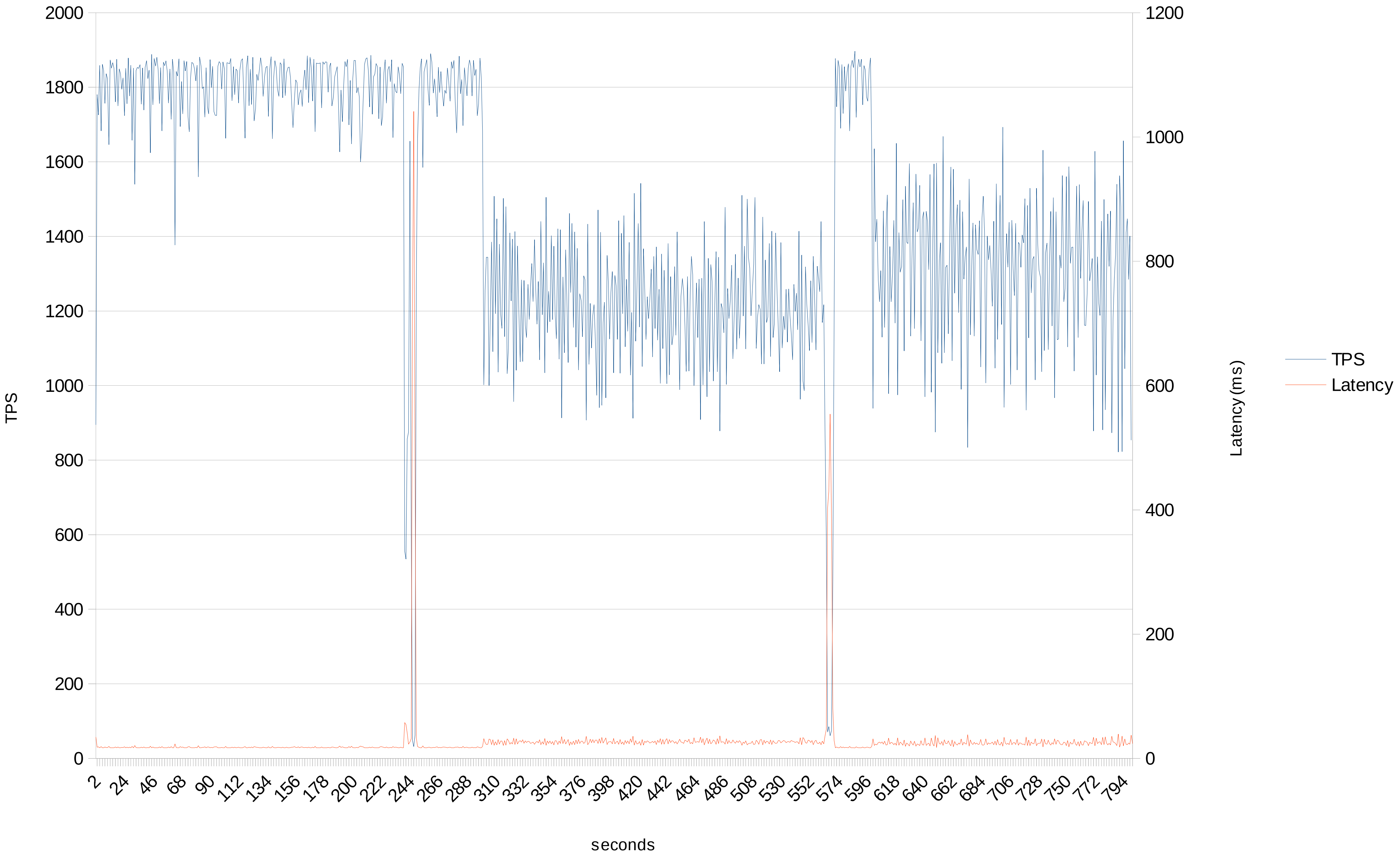
shared_buffers = 16GB, max_wal_size = 100GB, OS tuning (no dirty)



citusdata

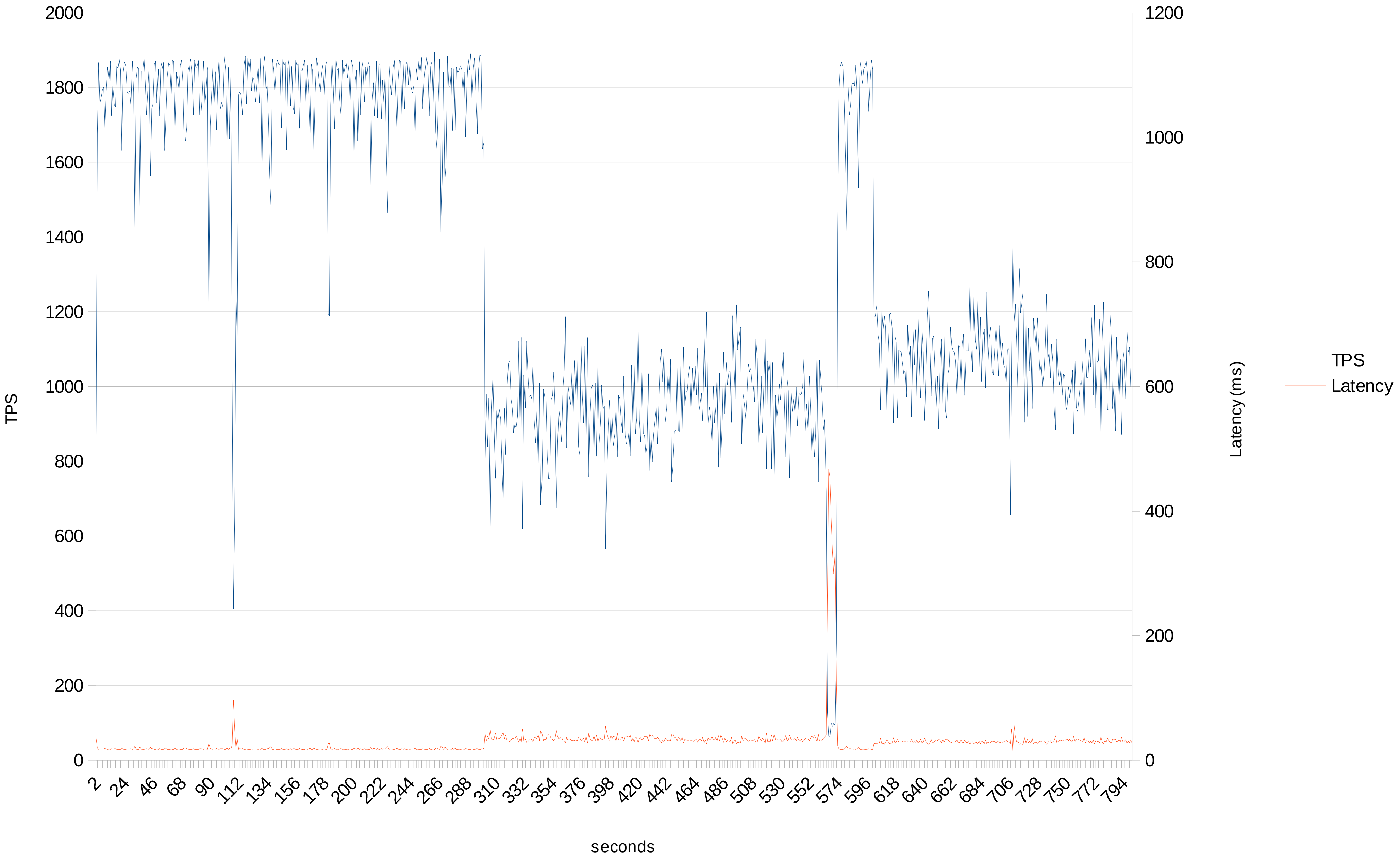
## pgbench -M prepared -c 32 -j 32

shared_buffers = 16GB, max_wal_size = 100GB, target = 0.9; OS tuning (no dirty)



## pgbench -M prepared -c 32 -j 32

shared_buffers = 16GB, max_wal_size = 100GB, target = 0.9; 9.6 flushing



# OS Dirty Data Tuning

- dirty_writeback_centisecs => lower
  - how often to check for writeback
- dirty_bytes/dirty_ratio => lower
  - when to block writing data
- dirty_background_bytes => lower
  - when to write data back in the background
- Increases random writes!
- Often slows total throughput, but improves latency

# WAL tuning

- Checkpoints should be triggered by time!
  - high enough checkpoint_segments/wal_max_size
  - Monitor!
- Except maybe at night, during batch runs or such
- Consider recovery time → less frequent checkpoints, crash recovery takes longer
- Consider full page writes → more frequent checkpoints mean much much more WAL
- separate pg_xlog can help a lot!

# WAL Writer

- Writes WAL instead backends
- Important for synchronous_commit = off
- Otherwise boring

# Background Writer

- Write dirty buffers before backends
- Not very good
- All random writes
- Defaults write max 4MB/s
- bgwriter_delay → lower, wakes up more often
- bgwriter_lru_maxpages → increases, writes more at once

# Problem – Bad Benchmarks

- pgbench has unrealistic workload
- hard to measure regressions
- contribute!

# Problem – Dirty Buffers in Kernel

- Massive Latency Spikes, up to hundreds of seconds
- Force flush using sync_file_range() or msync()
  - Decreases jitter
  - Increases randomness
- Sort checkpointed buffers
  - Decreases randomness
  - Increases Throughput
- Hopefully 9.6

# Problem – Hashtable

- Can't efficiently search for the next buffer
  - need to sort for checkpoints
  - can't write combine to reduce total number of writes
- Expensive Lookups
  - Cache inefficient datastructure
- Possible Solution: Radix Tree
- Hopefully 9.7

# Problem - Cache Replacement Scales Badly

- Single Lock for Clock Sweep!
  - fixed in 9.5
- Every Backend performs Clock Sweep
  - fixed in 9.6
- Algorithm is fundamentally expensive
  - UH, Oh.

# Problem - Cache Replacement Replaces Badly

- Usagecount of 5 (max) reached very quickly
  - Often all buffers have 5
- Increasing max usagecount increases cost, the worst case essentially is

  O(NBuffer * max_usagecount)
- Hard to solve, patent issues

# Problem: Kernel Page Cache

- Double buffering decreases effective memory utilization
- Use O_DIRECT?
  - Requires lots of performance work on our side
  - Considerably faster in some scenarios
  - Less Adaptive
  - Very OS specific