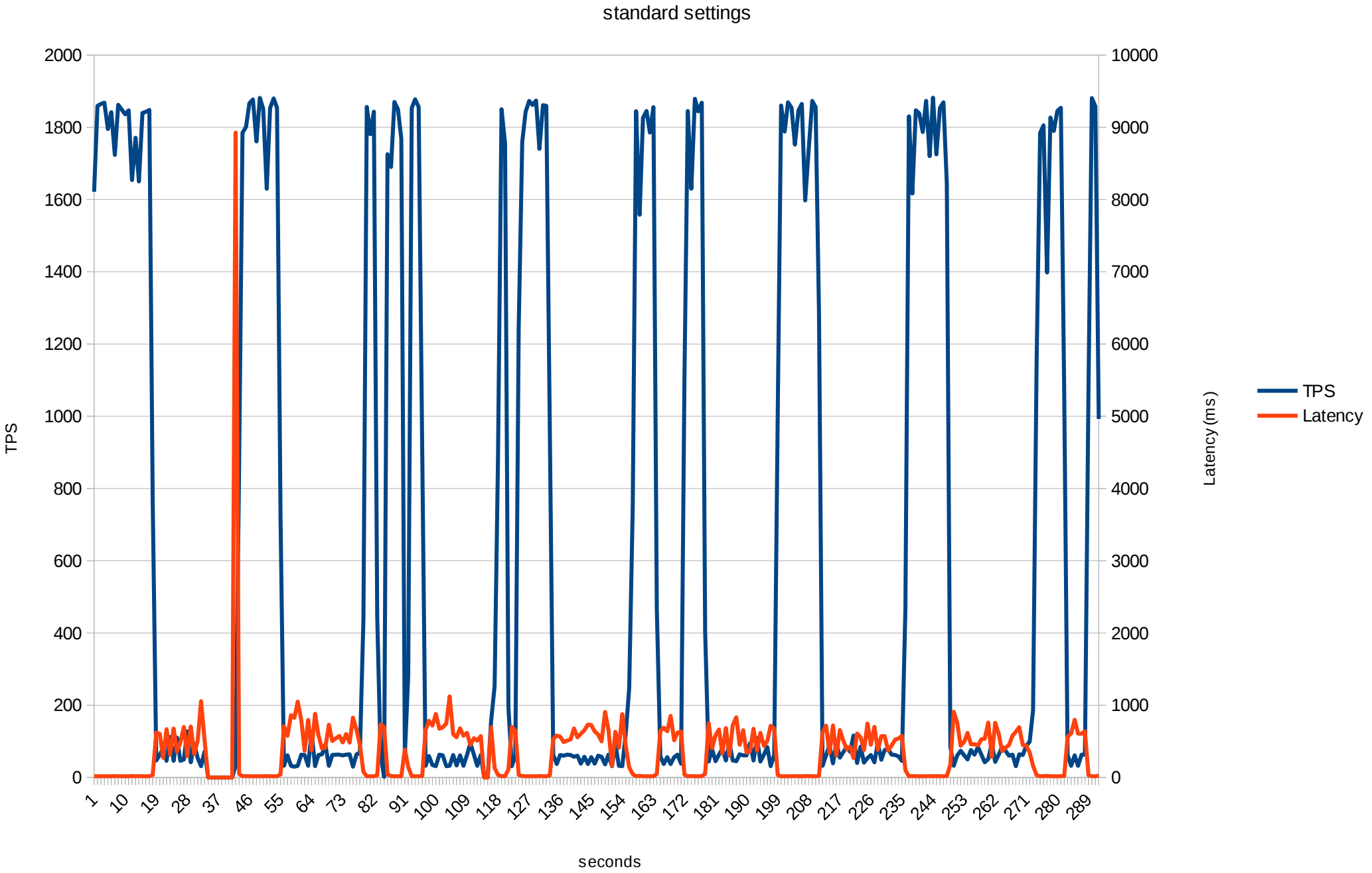
Postgres' IO
Architecture, Tuning, Problems

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

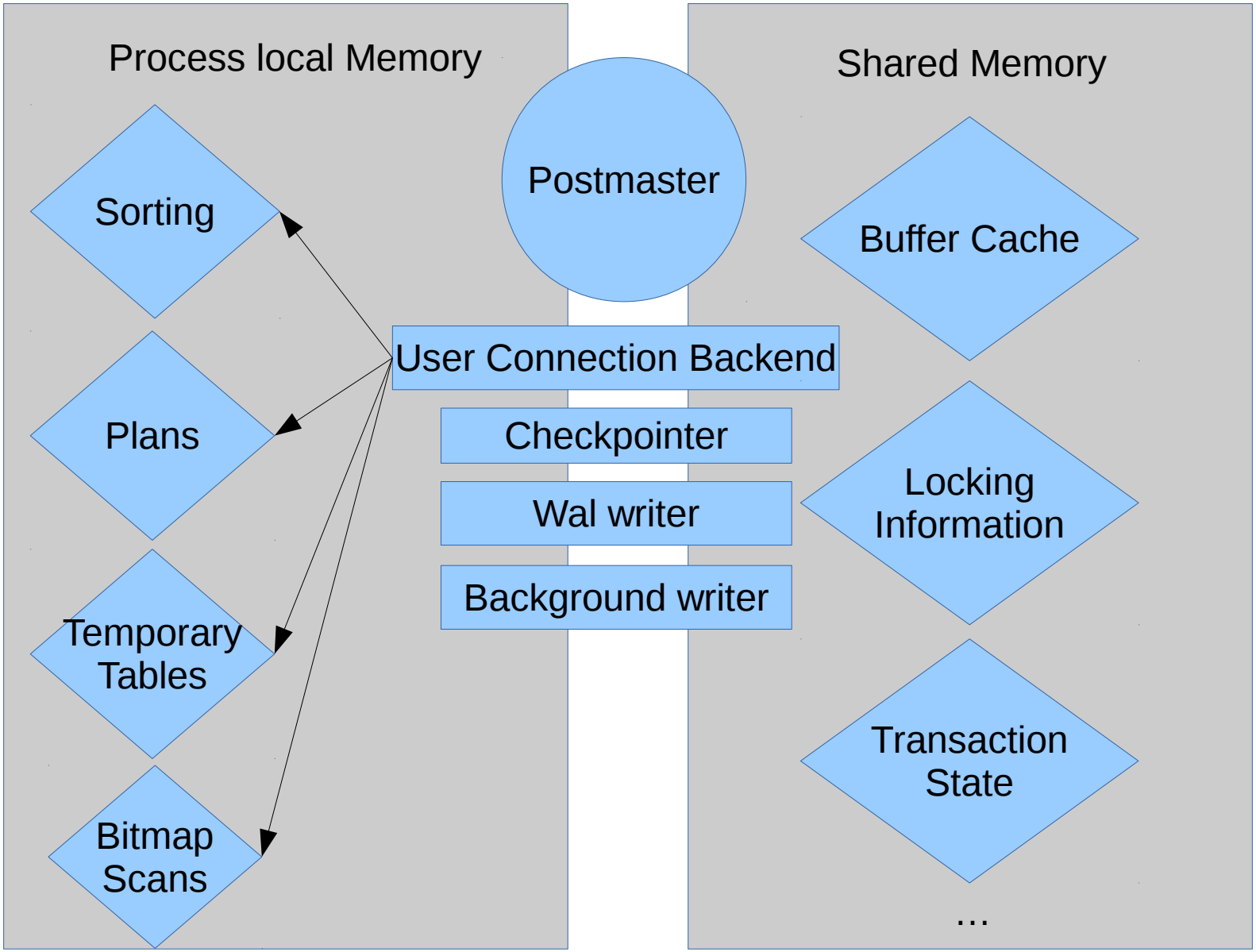
http://anarazel.de/talks/pgdevday-prague-2016-02-18/io.pdf

citusdata

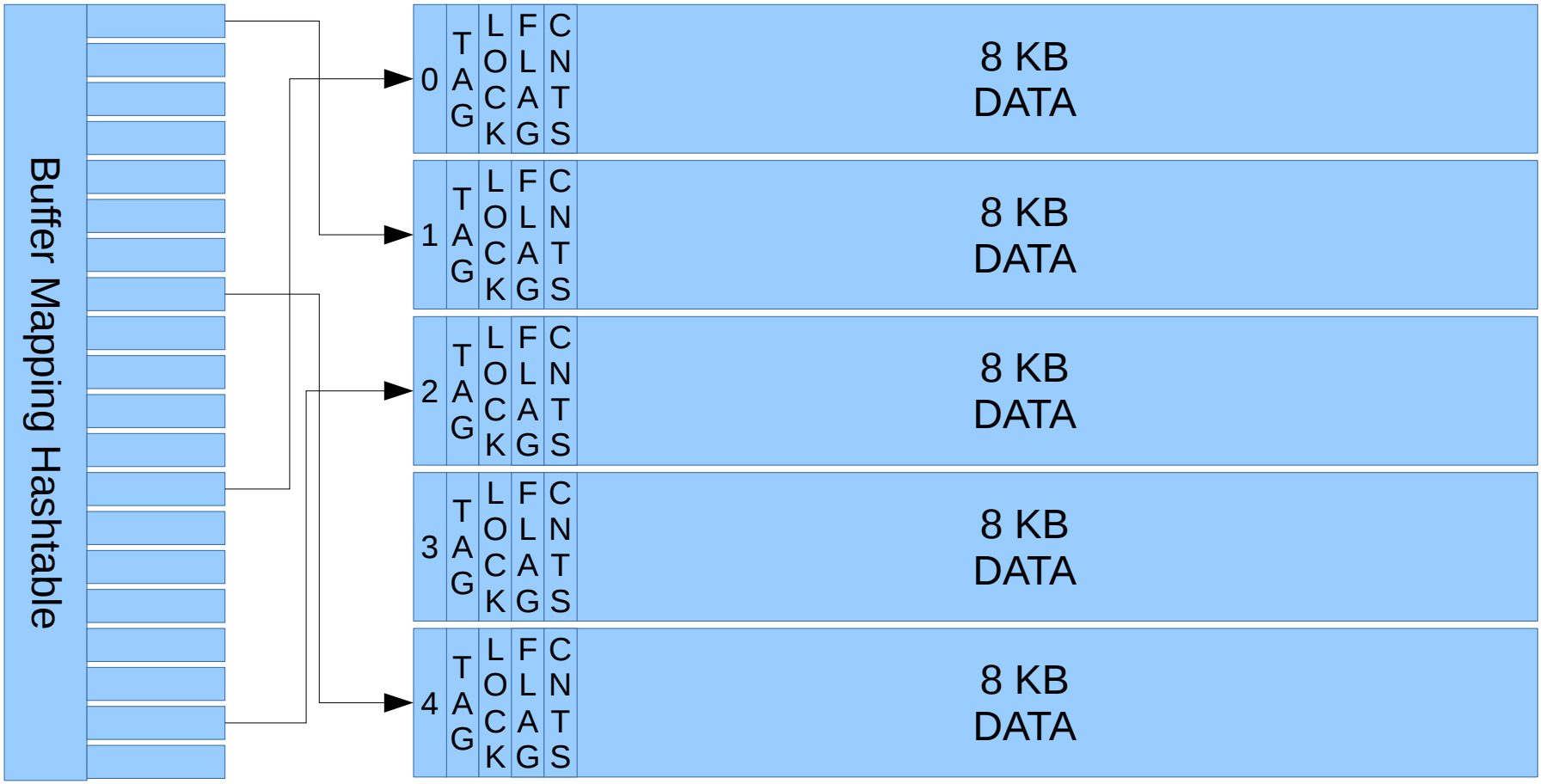

pgbench -M prepared -c 32 -j 32
standard settings
TPS
Latency (ms)
seconds
TPS
Latency
2000
1800
1600
1400
1200
1000
800
600
400
200
0
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
1 10 19 28 37 46 55 64 73 82 91 100 109 118 127 136 145 154 163 172 181 190 199 208 217 226 235 244 253 262 271 280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

citusdata

Memory Architecture

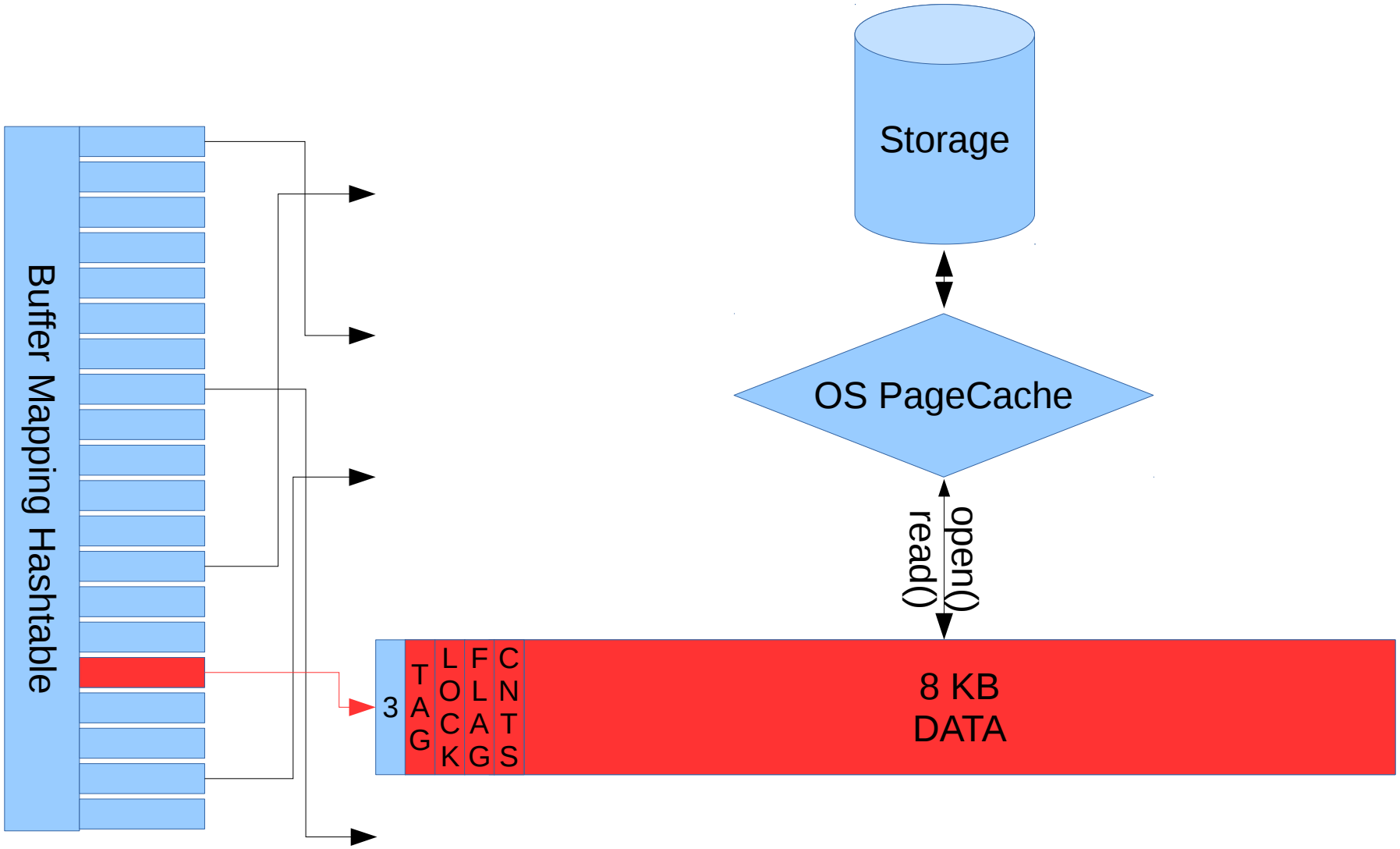


Shared Buffers

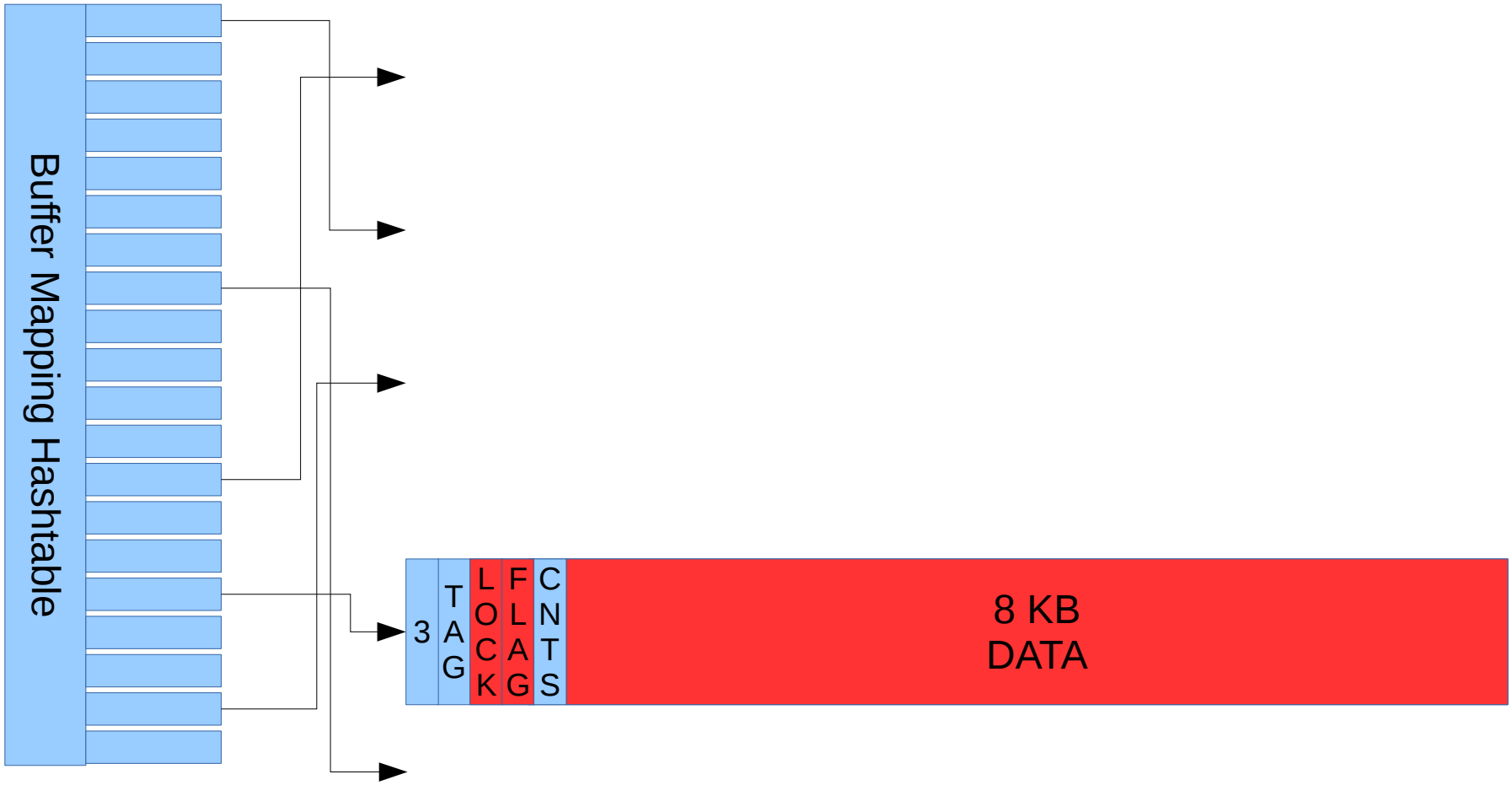


citusdata

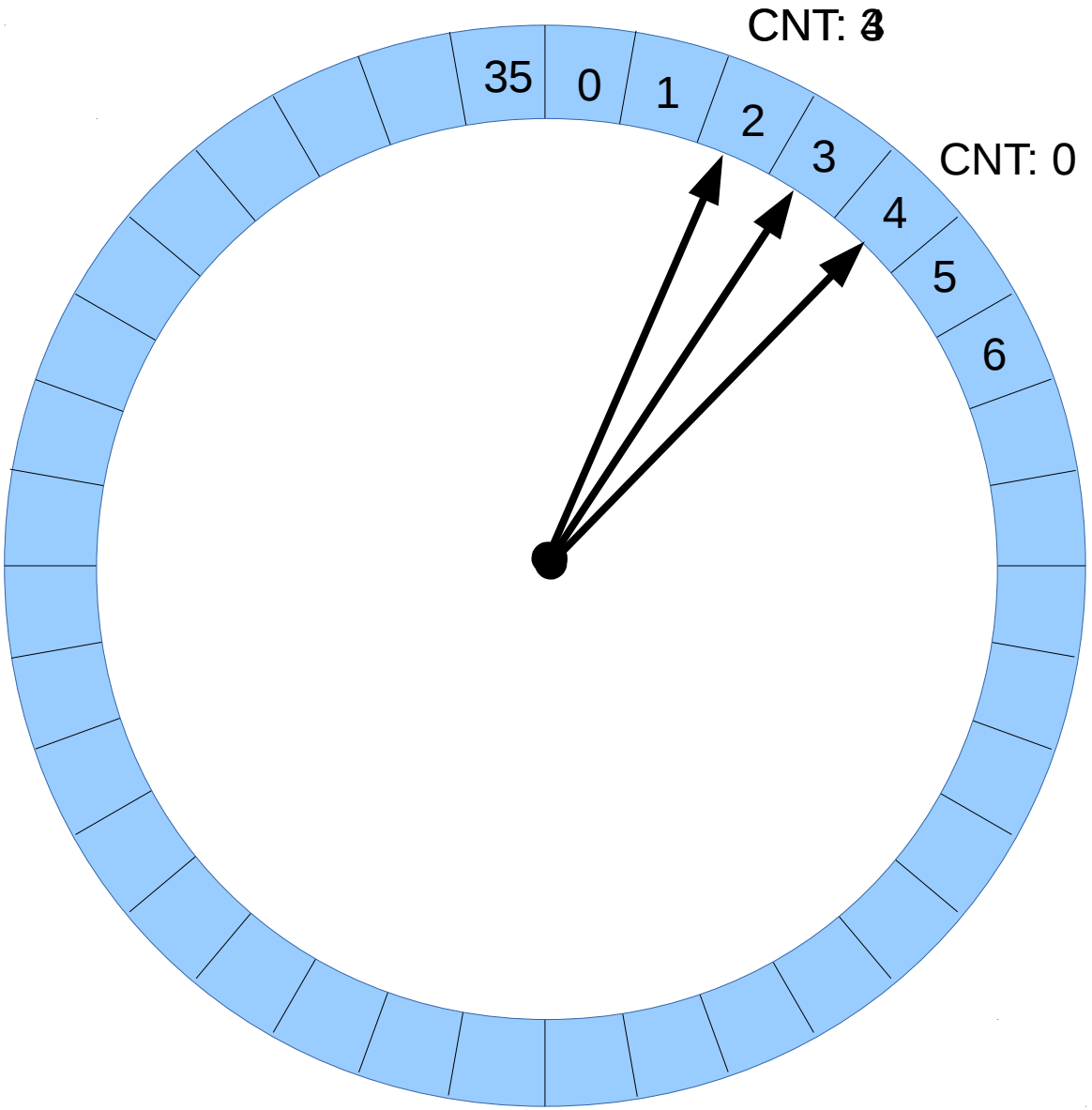
Reading Data



Writing Data

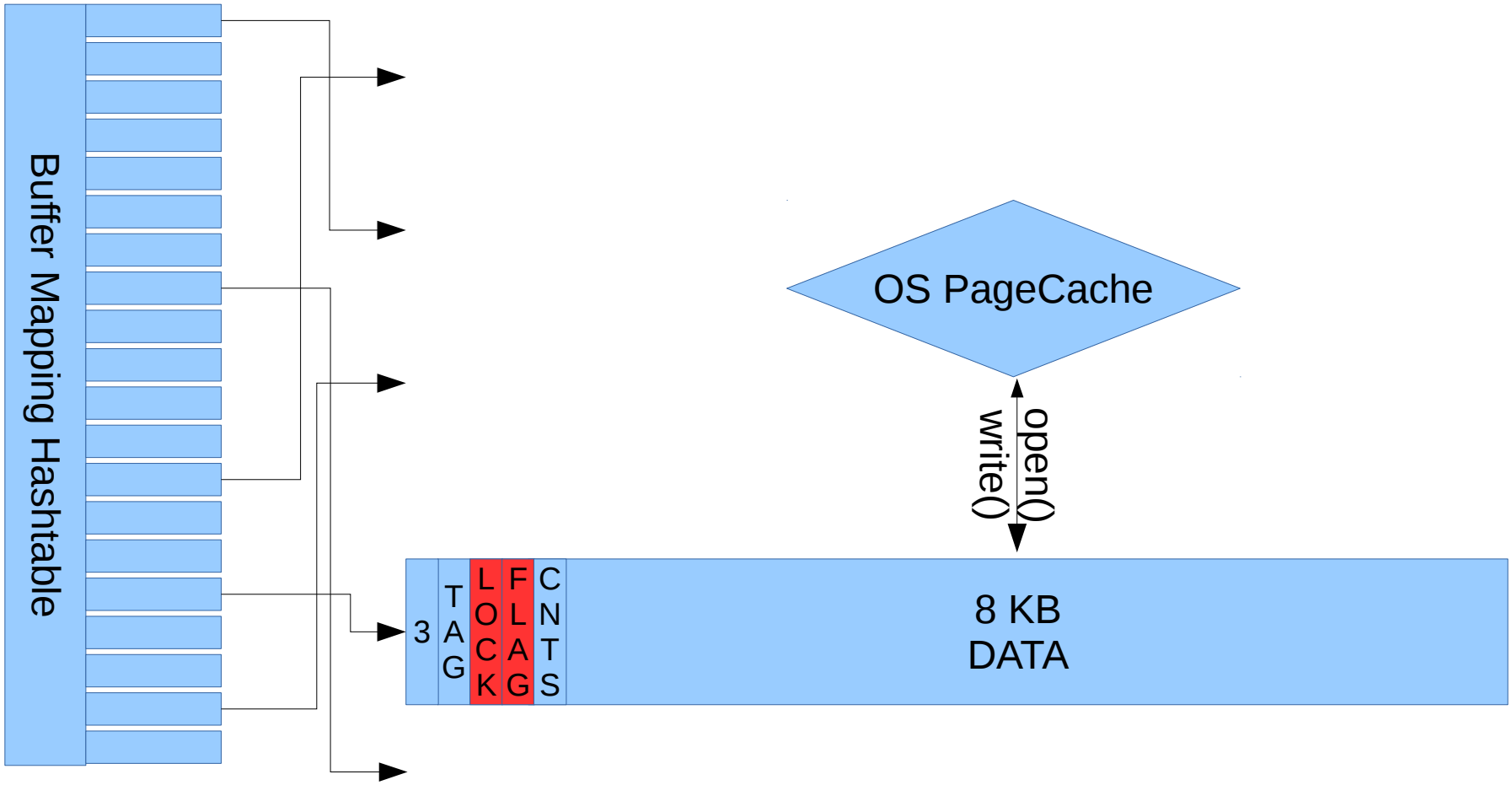


Clock-Sweep



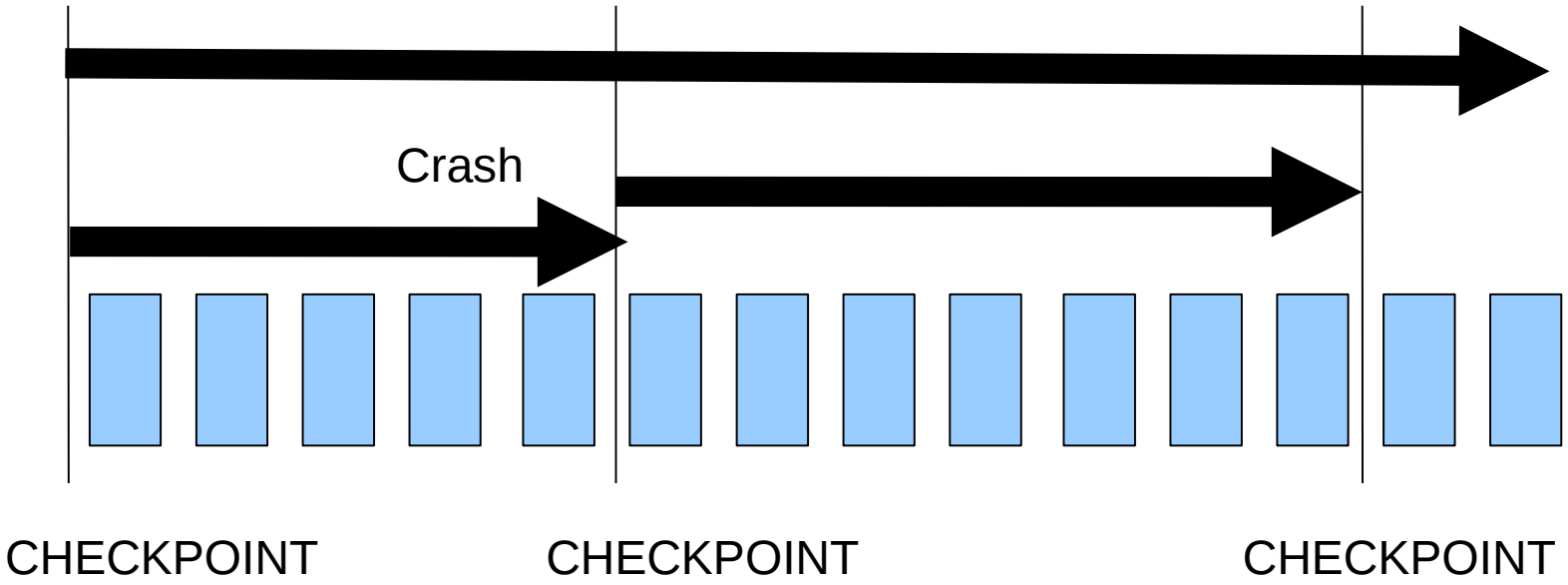
citusdata

Writing Data Out



citusdata

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.

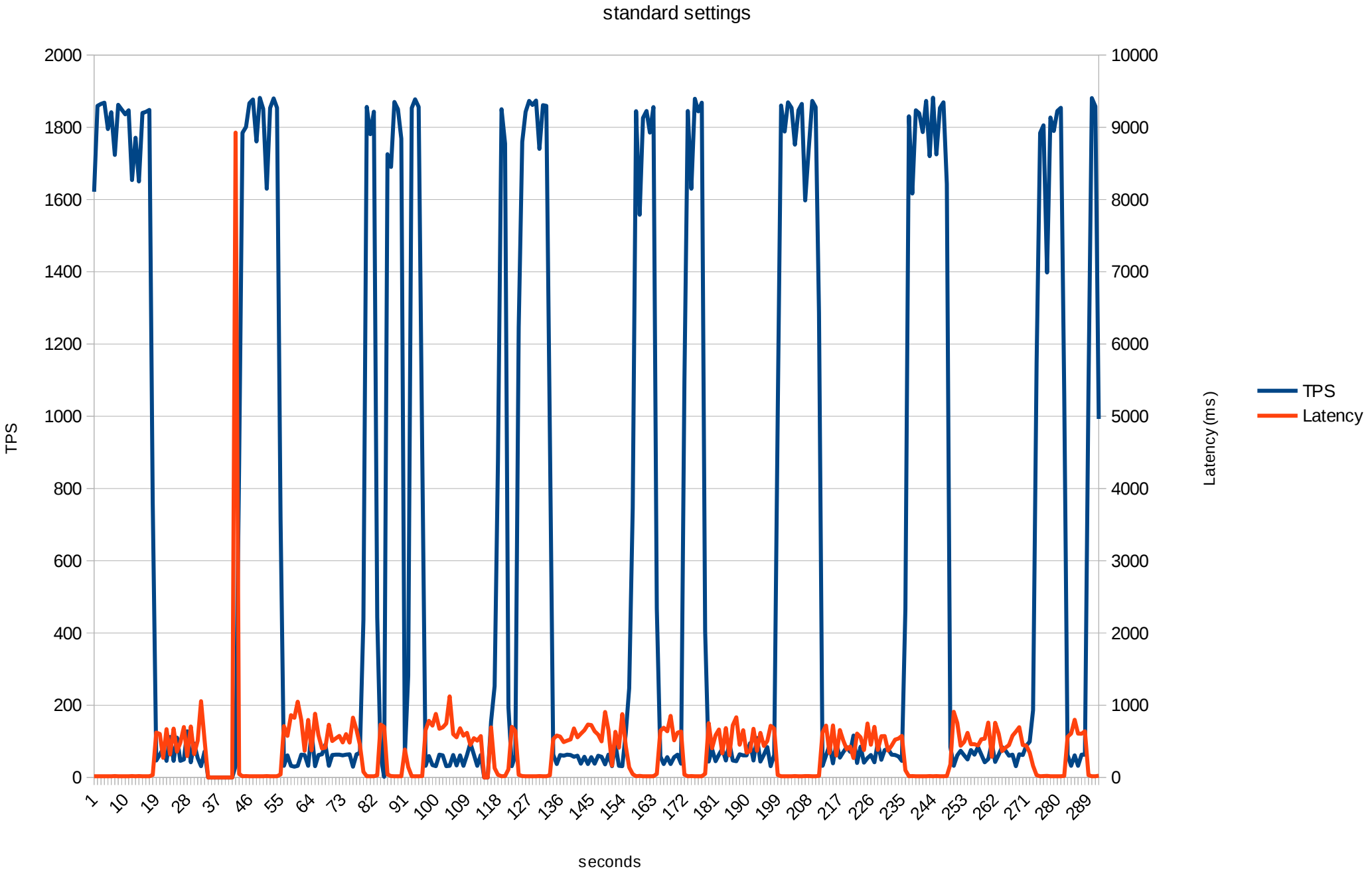
6)Remove old WAL

Triggering Checkpoints

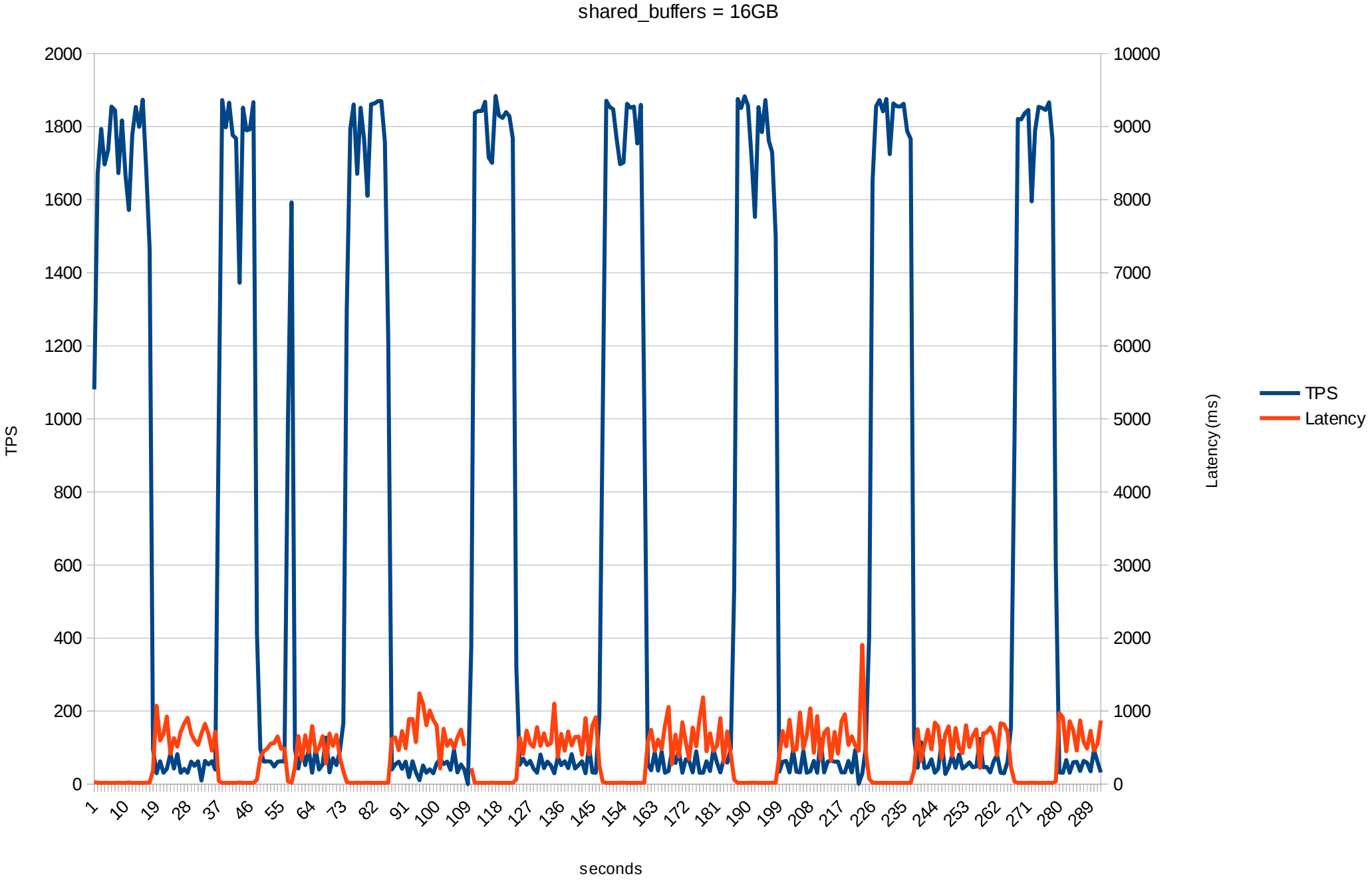
- checkpoint_timeout = 5min
  - LOG: checkpoint starting: time
- checkpoint_segments = 3 / max_wal_size = 1GB
  - LOG: checkpoint starting: xlog
  - LOG: checkpoints are occurring too frequently (2 seconds apart)
- shutdown
  - LOG: checkpoint starting: shutdown immediate
- manually (CHECKPOINT;)

Spreading Checkpoints

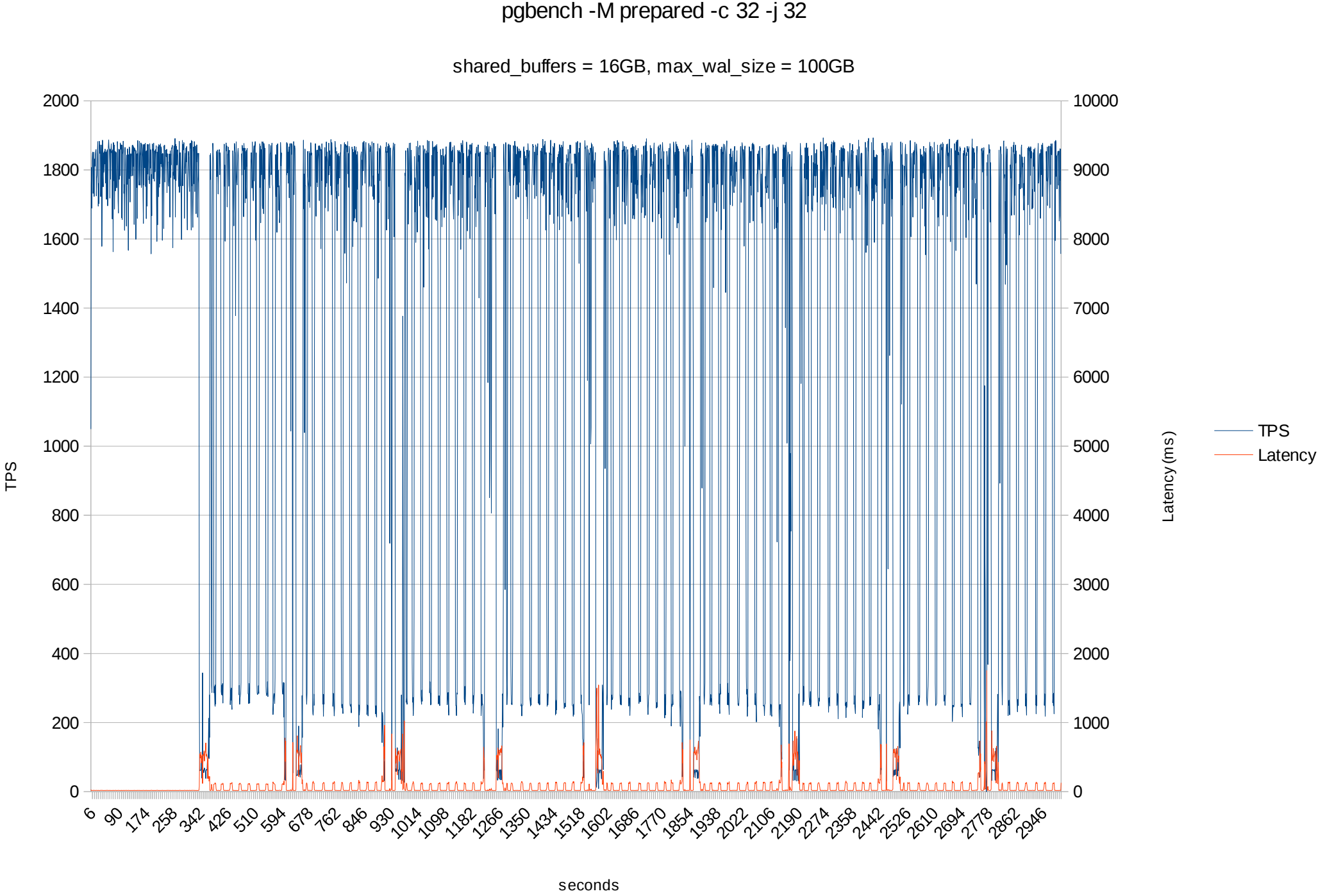
- checkpoint_completion_target = 0.5
- estimation based on
  - checkpoint_timeout
  - checkpoint_segments/max_wal_size
- Spread checkpoints over completion_target * timeout/segments till next checkpoint
- Try to keep pace


pgbench -M prepared -c 32 -j 32
standard settings
TPS
Latency (ms)
seconds
TPS
Latency
2000
1800
1600
1400
1200
1000
800
600
400
200
0
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
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

citusdata


pgbench -M prepared -c 32 -j 32
shared_buffers = 16GB
TPS
Latency (ms)
seconds
TPS
Latency
2000
1800
1600
1400
1200
1000
800
600
400
200
0
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
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

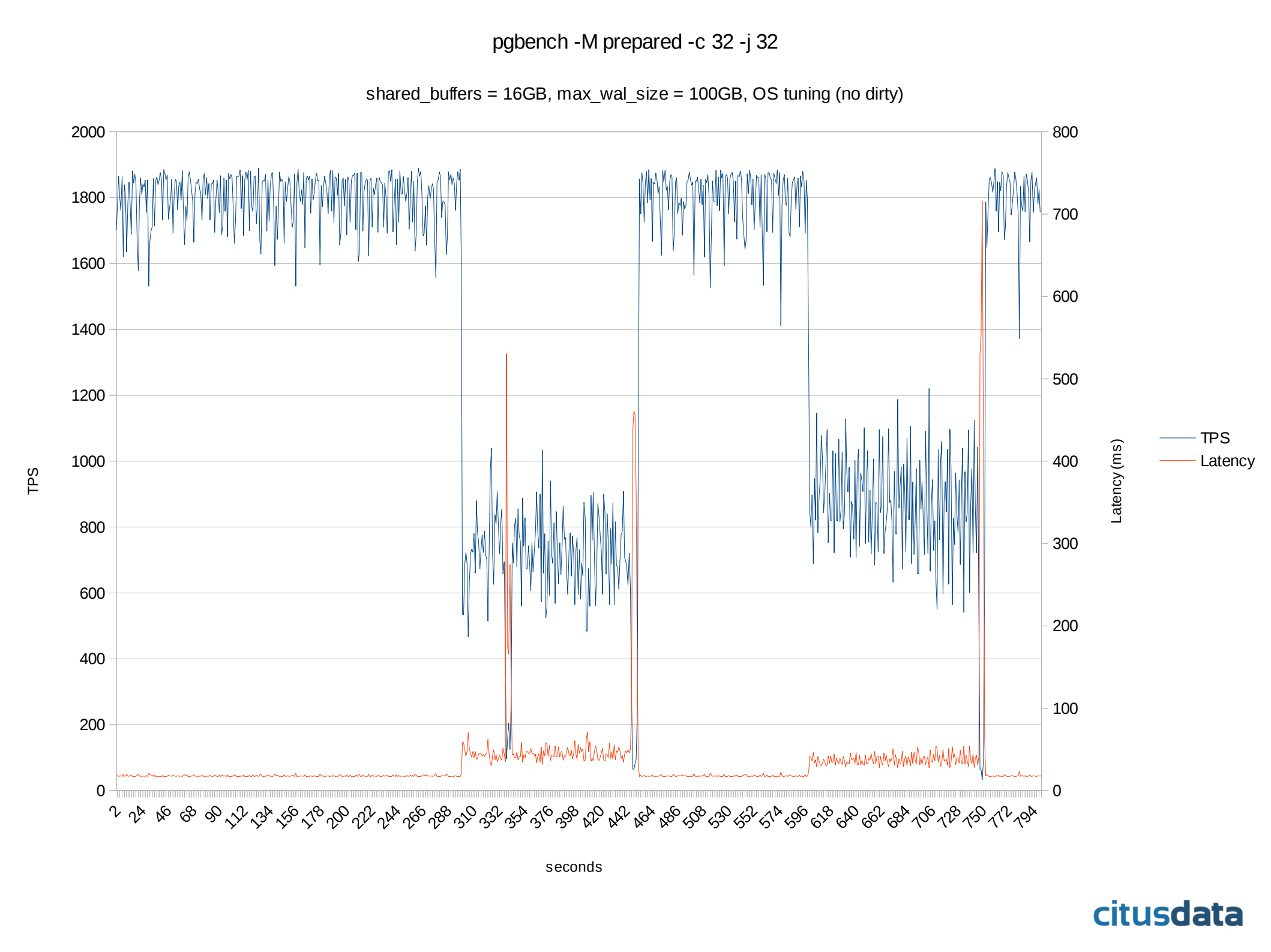
citusdata

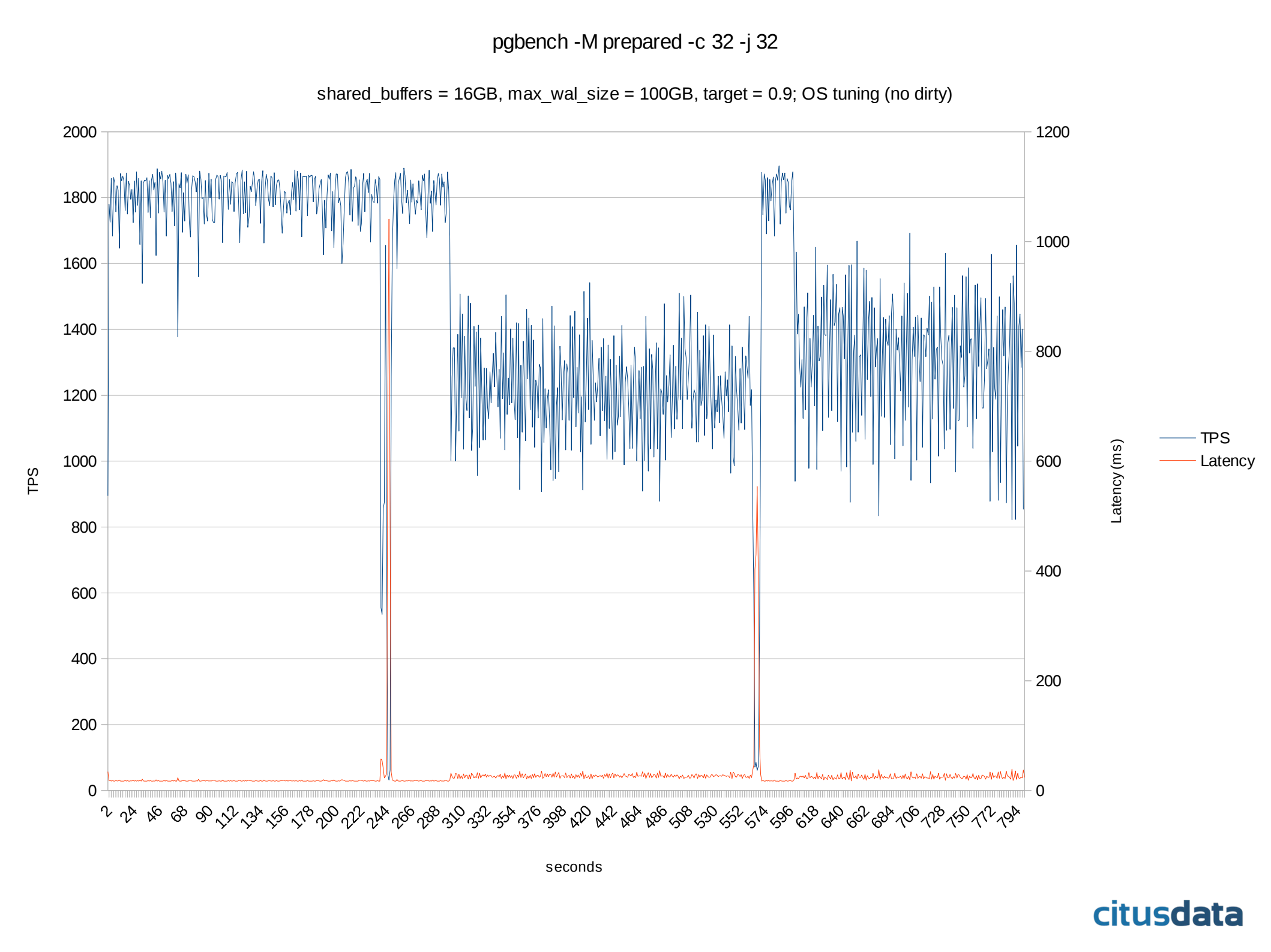

pgbench -M prepared -c 32 -j 32
shared_buffers = 16GB, max_wal_size = 100GB
TPS
Latency (ms)
seconds
TPS
Latency
0
200
400
600
800
1000
1200
1400
1600
1800
2000
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
6
90
174
258
342
426
510
594
678
762
846
930
1014
1098
1182
1266
1350
1434
1518
1602
1686
1770
1854
1938
2022
2106
2190
2274
2358
2442
2526
2610
2694
2778
2862
2946

citusdata

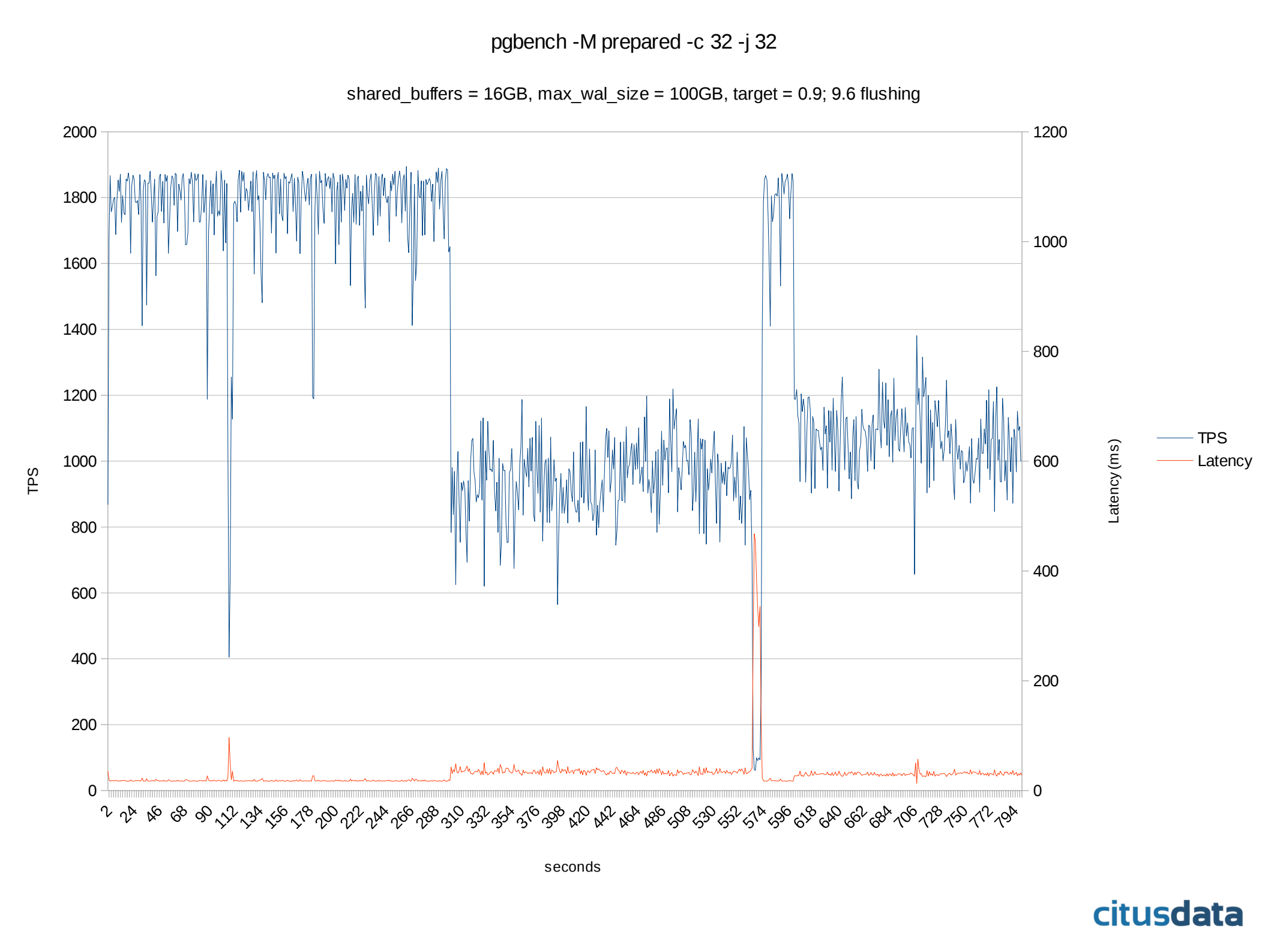

Dirty Data
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
bytes
7
91
175
259
343
427
511
595
679
763
847
931
1015
1099
1183
1267
1351
1435
1519
1603
1687
1771
1855
1939
2023
2107
2191
2275
2359
2443
2527
2611
2695
2779
2863
2947
3031
time (seconds)
Dirty
Writeback

citusdata


pgbench -M prepared -c 32 -j 32
shared_buffers = 16GB, max_wal_size = 100GB, OS tuning (no dirty)
TPS
Latency (ms)
seconds
TPS
Latency
2000
1800
1600
1400
1200
1000
800
600
400
200
0
800
700
600
500
400
300
200
100
0
2
24
46
68
90
112
134
156
178
200
222
244
266
288
310
332
354
376
398
420
442
464
486
508
530
552
574
596
618
640
662
684
706
728
750
772
794
citusdata


pgbench -M prepared -c 32 -j 32
shared_buffers = 16GB, max_wal_size = 100GB, target = 0.9; OS tuning (no dirty)
TPS
Latency (ms)
seconds
TPS
Latency
2000
1800
1600
1400
1200
1000
800
600
400
200
0
1200
1000
800
600
400
200
0
2
24
46
68
90
112
134
156
178
200
222
244
266
288
310
332
354
376
398
420
442
464
486
508
530
552
574
596
618
640
662
684
706
728
750
772
794

citusdata


pgbench -M prepared -c 32 -j 32
shared_buffers = 16GB, max_wal_size = 100GB, target = 0.9; 9.6 flushing
TPS
Latency (ms)
seconds
TPS
Latency
2000
1800
1600
1400
1200
1000
800
600
400
200
0
1200
1000
800
600
400
200
0
2
24
46
68
90
112
134
156
178
200
222
244
266
288
310
332
354
376
398
420
442
464
486
508
530
552
574
596
618
640
662
684
706
728
750
772
794
citusdata

Shared Buffers Tuning

- Hot data fits into shared_buffers => increase s_b
- Bulk-Writes in a bigger than shared_buffers workload => measure decreasing s_b
- Large Shared Buffers => enable huge pages
- Frequent Relation DROP/REINDEX => decrease s_b

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