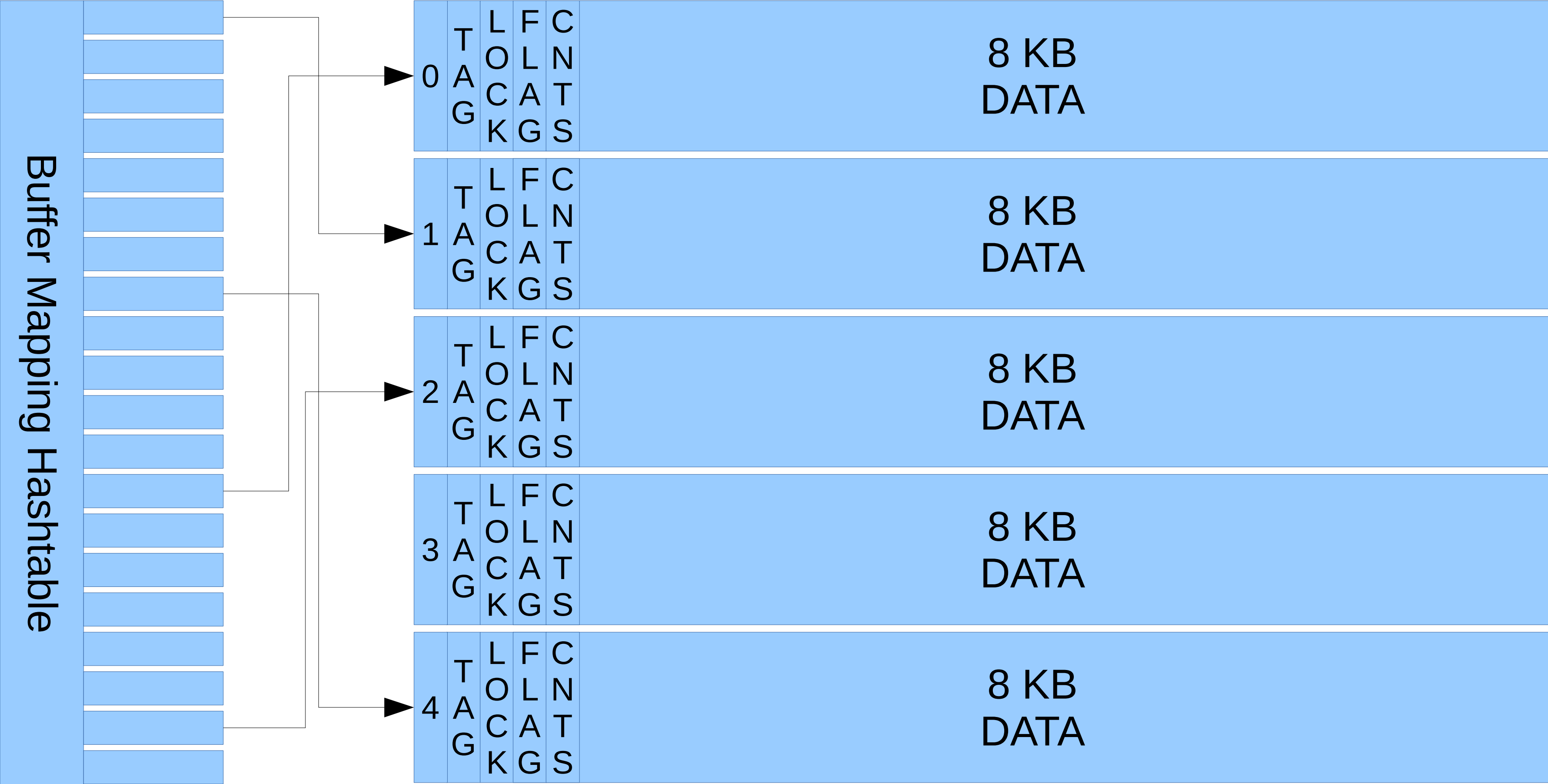
PostgreSQL's Buffer Manager Problems & Improvements

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

http://anarazel.de/talks/pgcon-2016-05-20/io.pdf

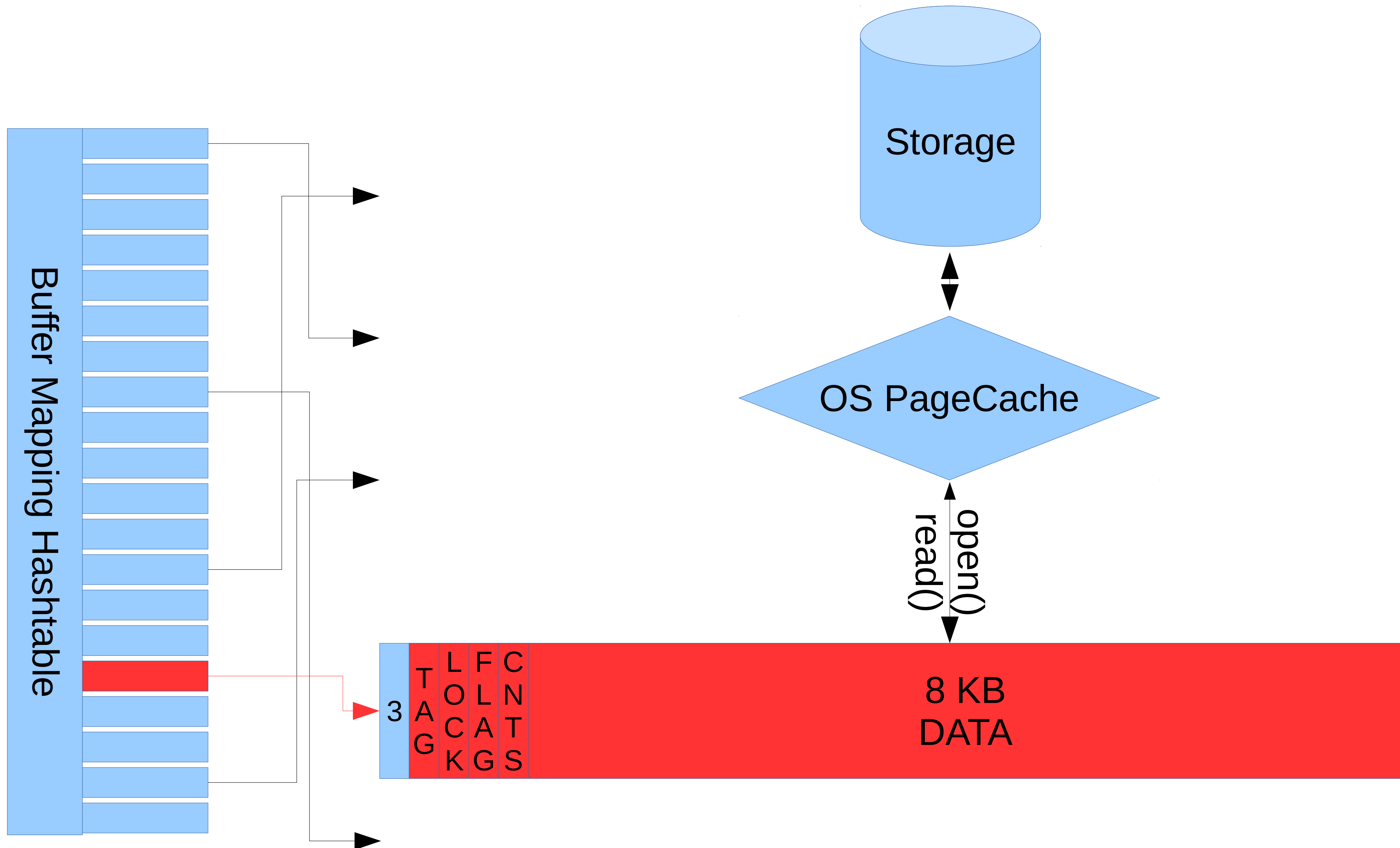
citusdata

Shared Buffers



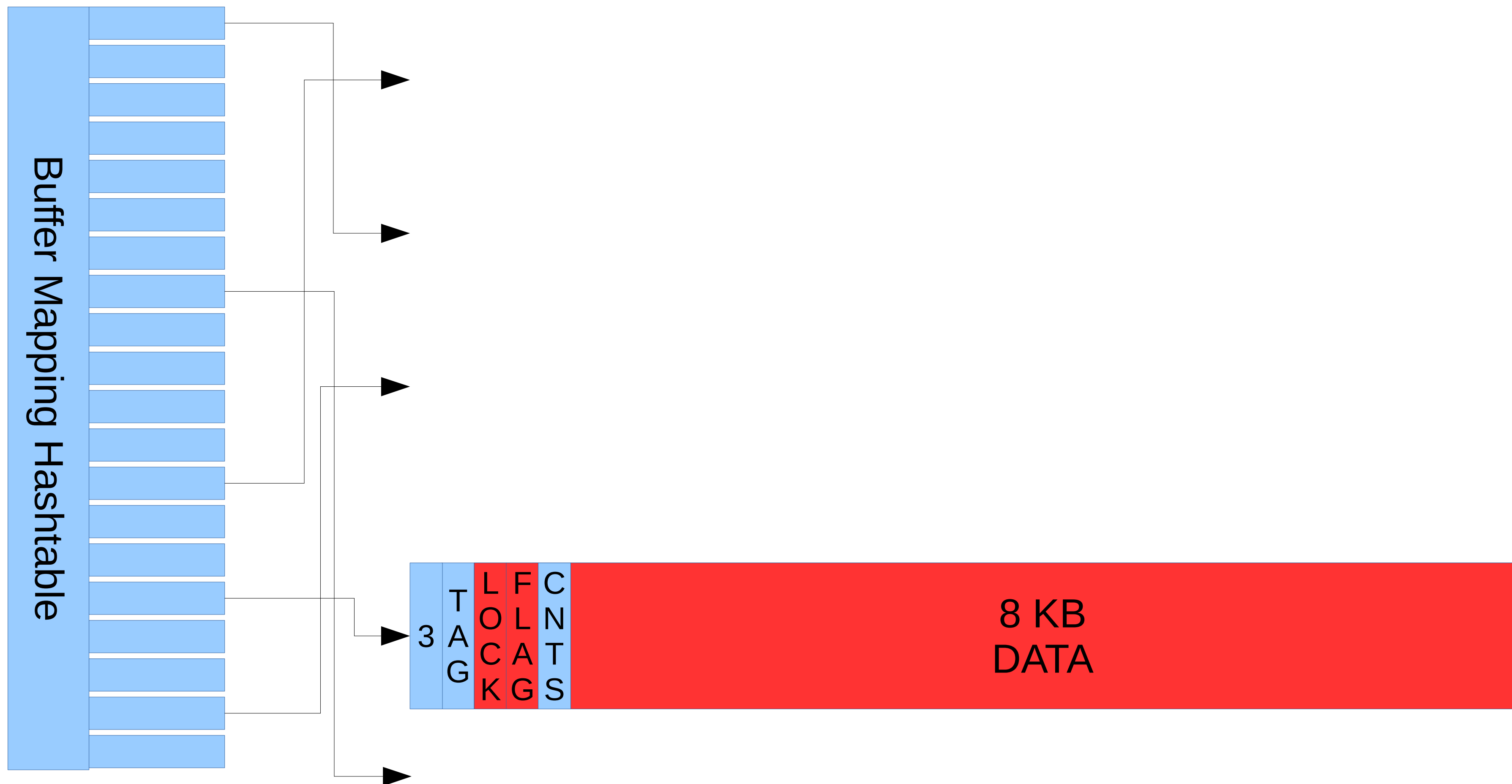
citusdata

Reading Data



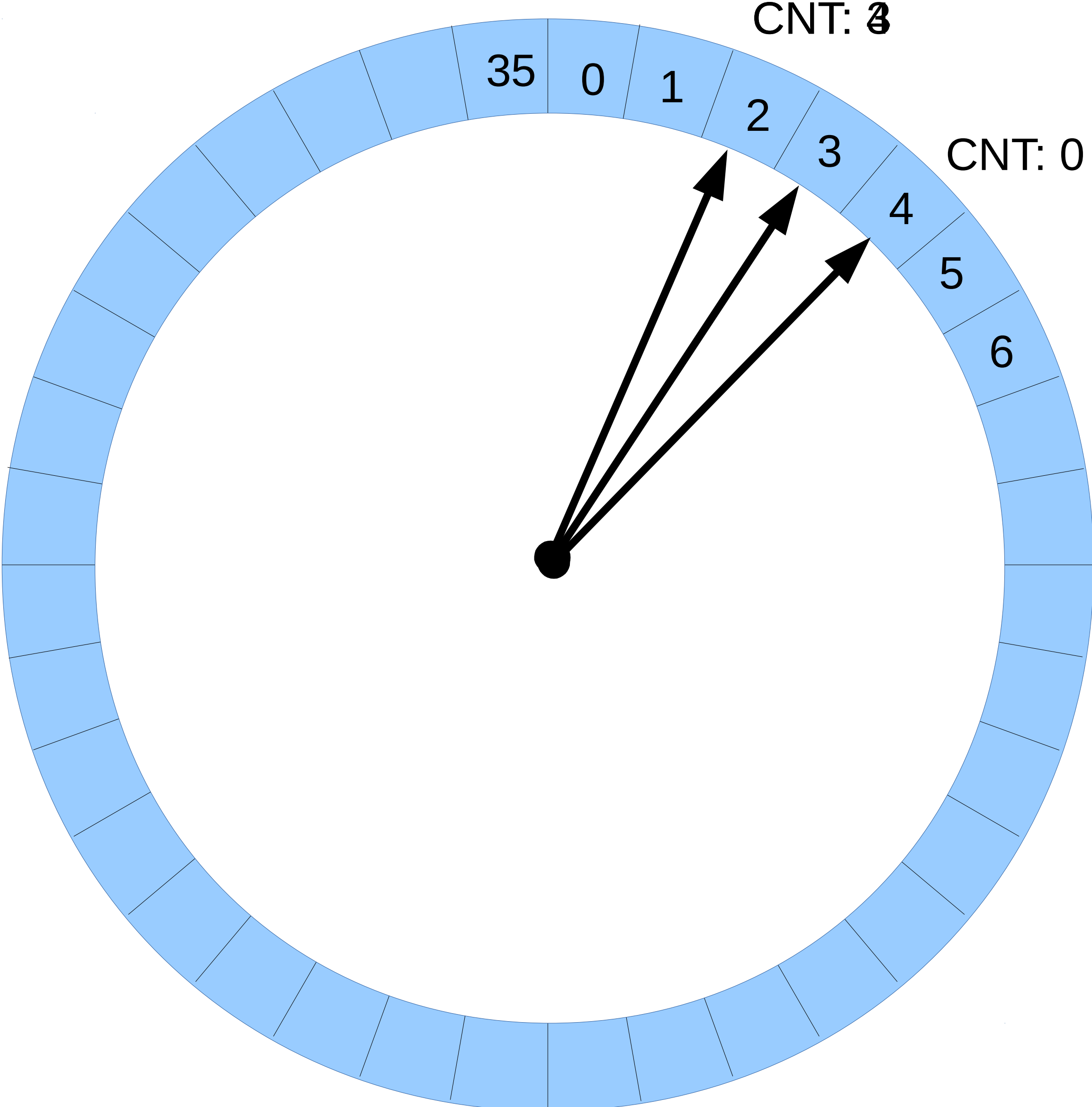
citusdata

Writing Data



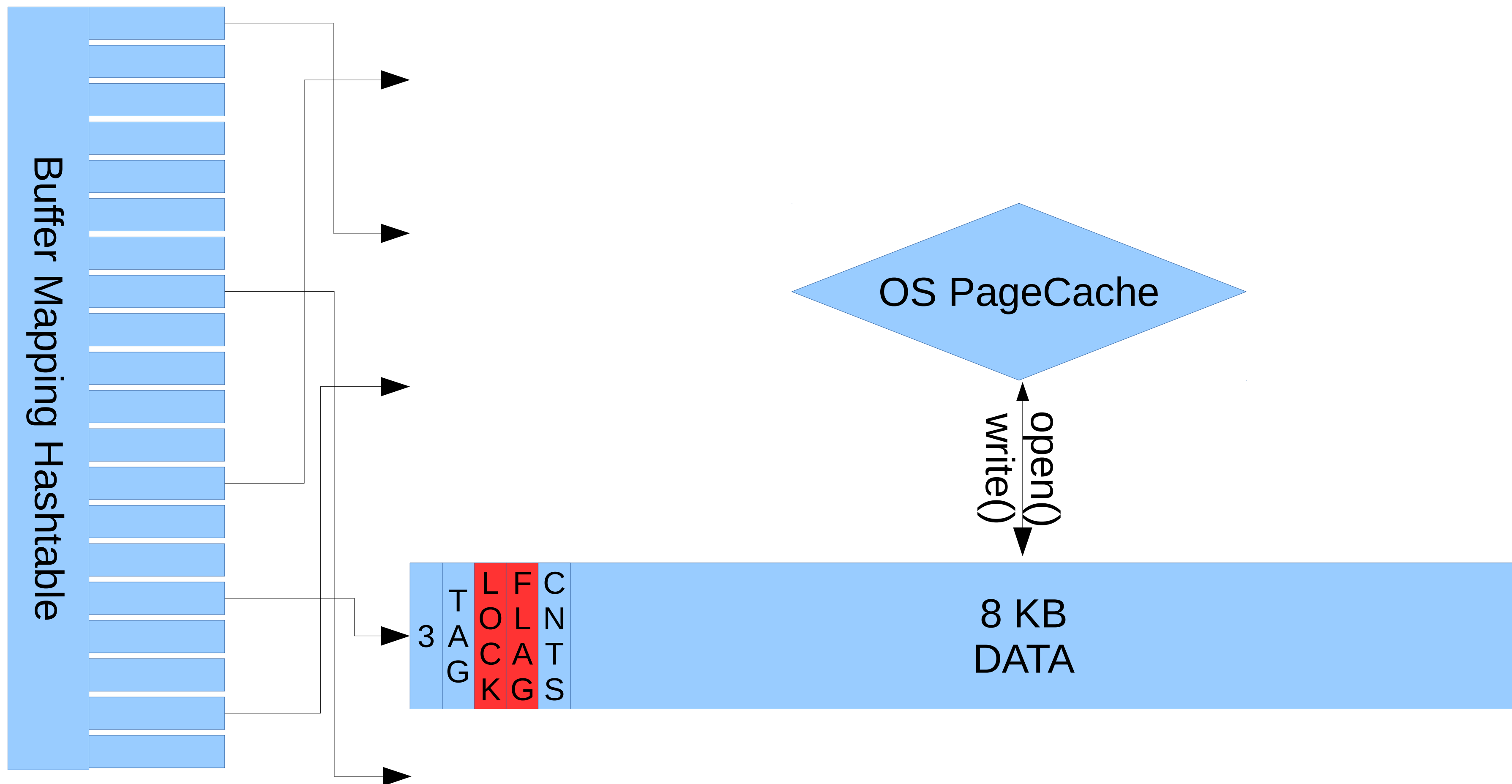
citusdata

Clock-Sweep

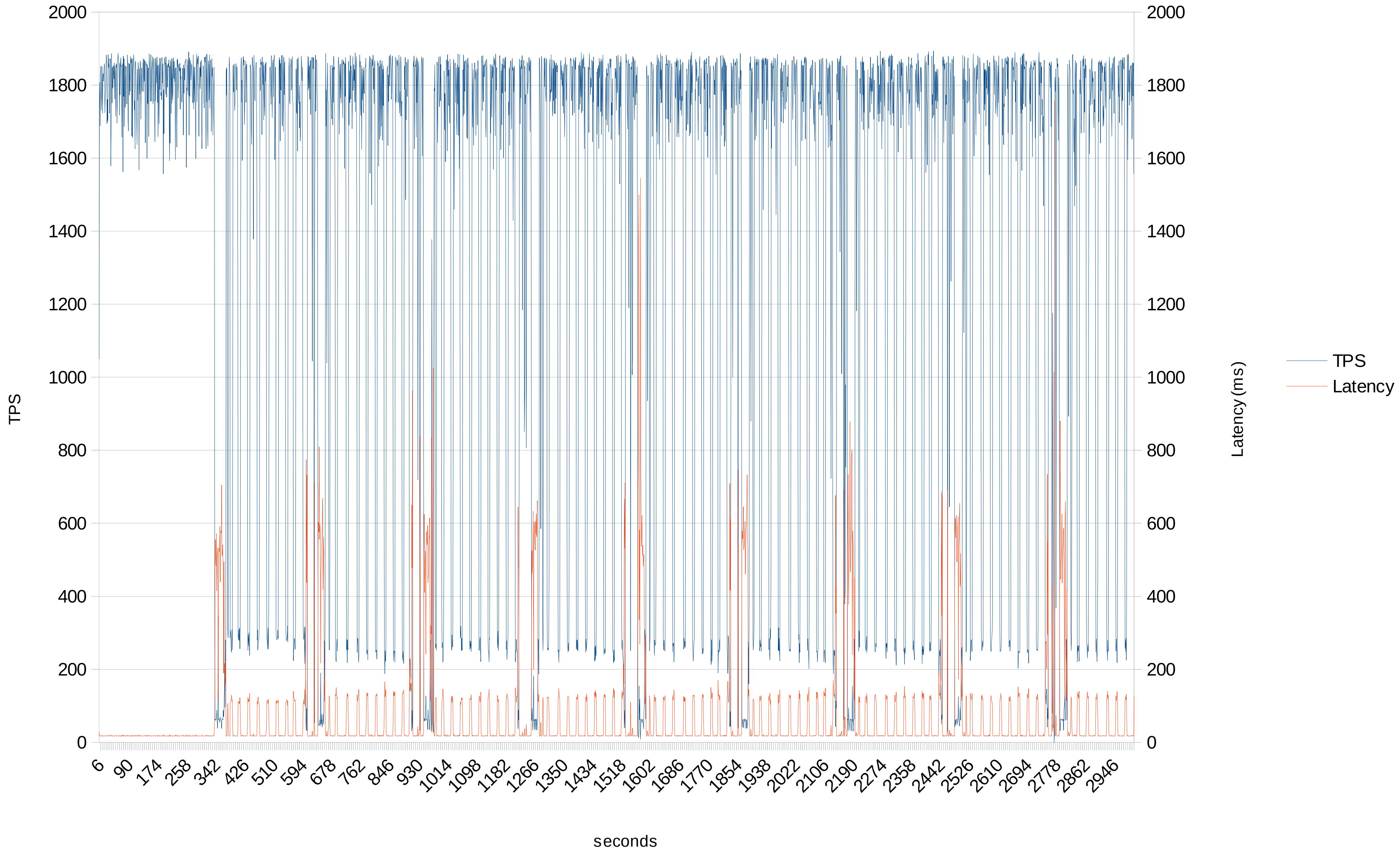


citusdata

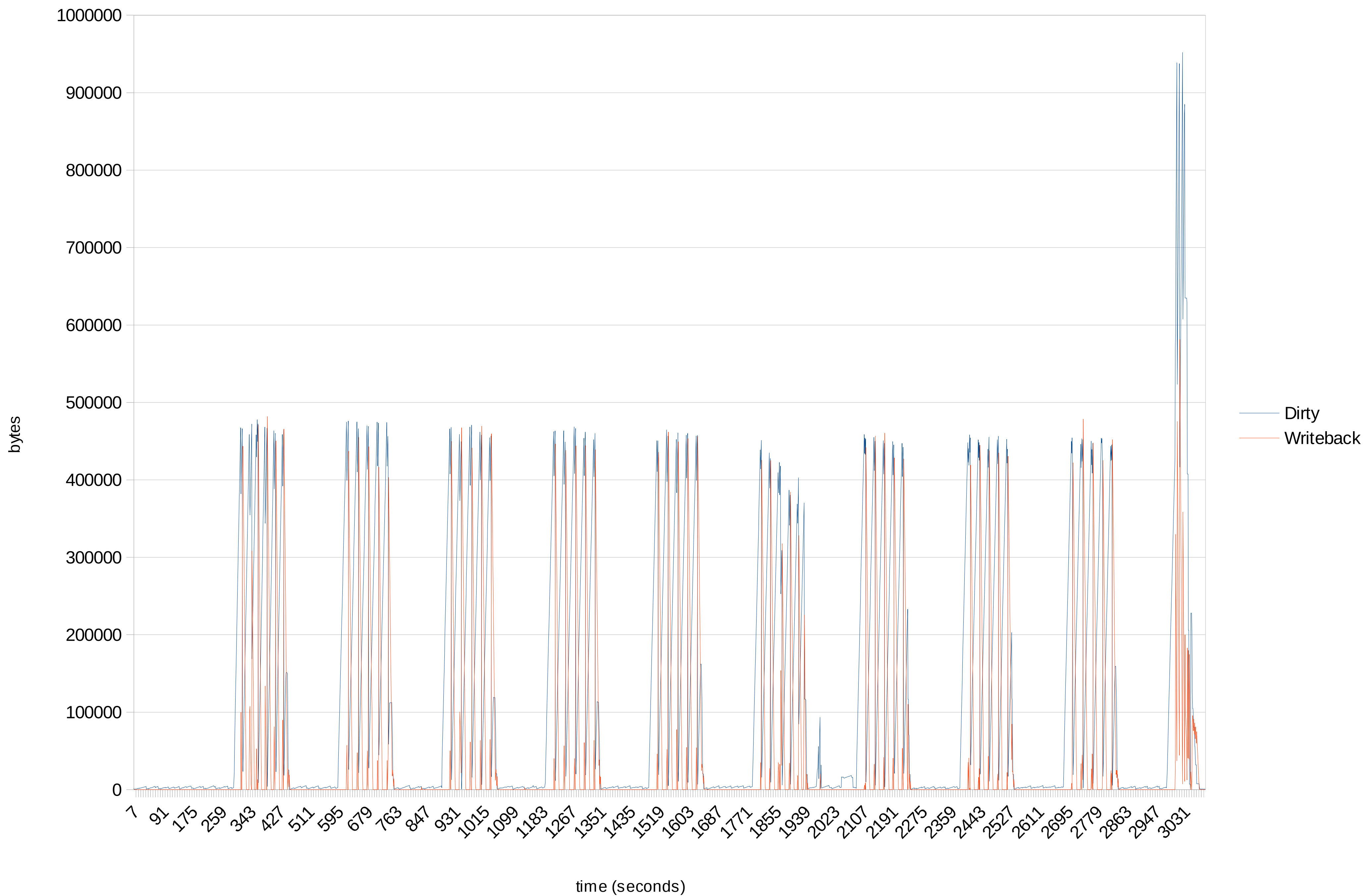
Writing Data Out



citusdata



citusdata



citusdata

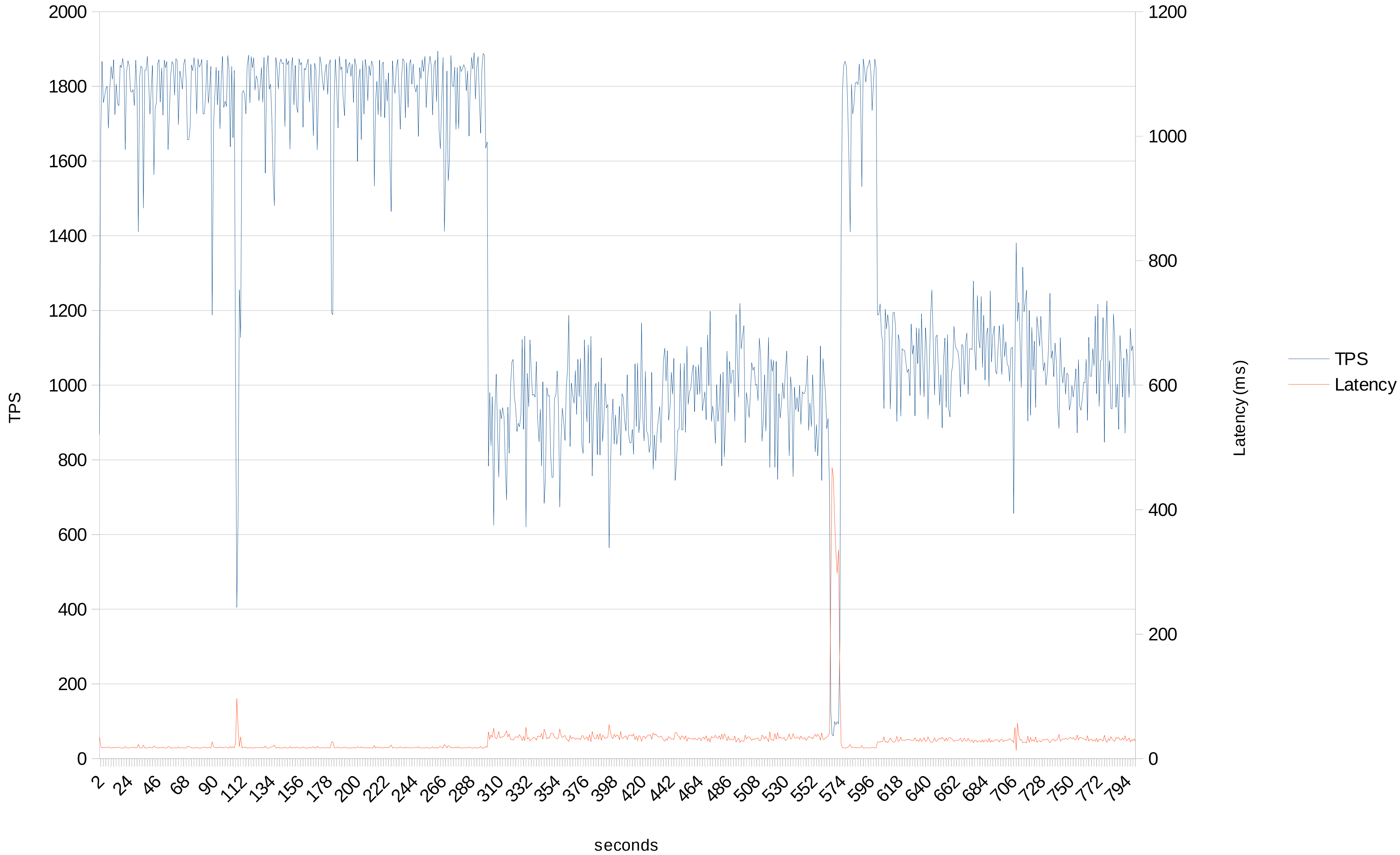
Problem – Dirty Buffers in Kernel

- Massive Latency Spikes, up to hundreds of seconds
- No actually efficient merging of IO requests
- latency spikes every dirty_writeback_centisecs
- spikes when reaching dirty_{background_,}ratio
- latency spikes after checkpoint's fsync()

citusdata

Kernel Dirty Buffer Control

- Use sync_file_range(), mmap/msync() to force OS to write back buffers
- Correctly configured machine → faster
- Unfortunately some workloads with bad config → slower
  - workload bigger than shared buffers, smaller than OS page cache
  - lots of re-dirtying of already dirtied pages
  - fundamental tradeoff


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
citusdata

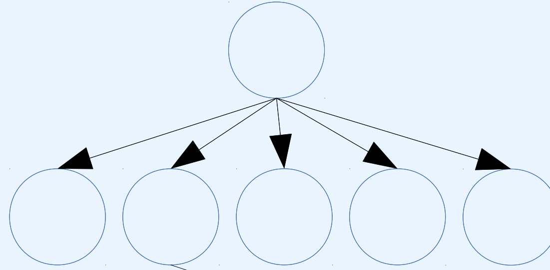
Problem: Hashtable

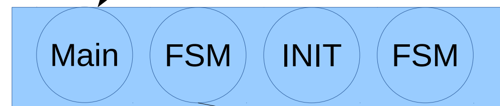
- Expensive Lookups
  - Wide keys need to be compared (20 bytes)
  - Cache inefficient datastructure (spatial locality)
- Can't efficiently search for the next buffer
  - can't scan for all buffers of a relation (DROP/TRUNCATE!)
  - can't write combine to reduce total number of writes
- Possible Solution
  - Open relations table
  - Tree structure for block lookups

```
typedef struct BufferTag
{
    struct RelFileNode
    {
        Oid         spcNode;        /* tablespace */
        Oid         dbNode;         /* database */
        Oid         relNode;        /* relation */
    } rnode;                        /* physical relation identifier */

    ForkNumber  forkNum;

    BlockNumber blockNum;           /* blknum relative to begin of reln */
} BufferTag;
```

Tree of Trees

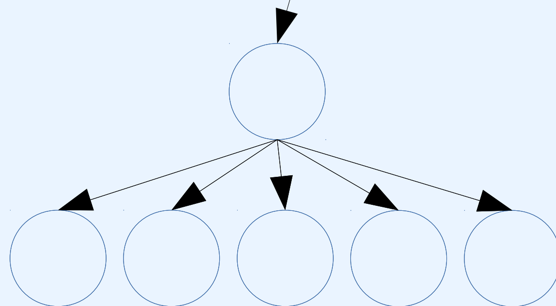
Upper Hash-Table / Tree (tablespace, database, relfilenode)





cached in SmgrRelation

Lower Tree (block number)

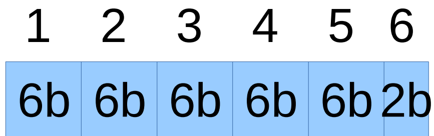


citusdata

Open Relations Table

- Store relation/fork size
  - no lseeks() anymore
  - shrink files without exclusive lock?
- Extend files without exclusive lock
  - track “next unallocated block”, increment atomically

Radix Tree “Linux Style”



0 P1
1 P990
2 P7
3 P9343

0 0x...
1 0x...
...
...
2^6 0x...

0 0x3464
1 0x5440
...
...
2^6 0x9520

0 0x...
1 0x...
...
...
2^6 0x...

....

...

0 0x...
1 0x...
...
...
2^6 0x...

citusdata

Solution: Lock-less / fewer locks

- Hash and radix tree can be made lock-free
- Memory reclamation tricky
  - Hazard-Pointers
  - Epoch-based reclamation
  - RCU
- lock-free reads / locked writes?

Problem: Backends do the writes

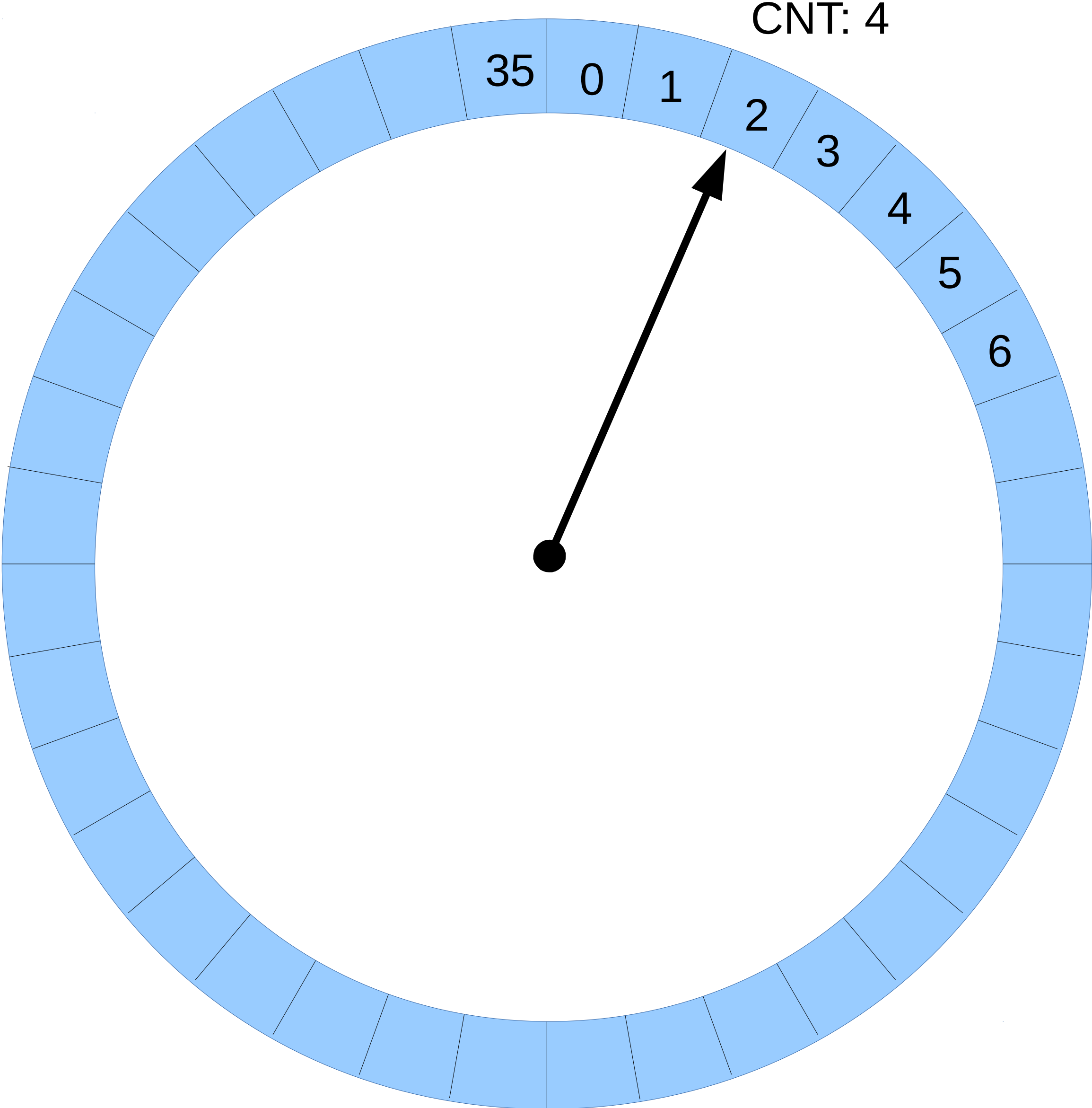
- Only if workload > shared_buffers
- Slows down queries
- Increases randomness of writes
- Limiting kernel dirty buffers hard
- bgwriter inefficient / hard to tune

citusdata

Problem - Cache Replacement Scales Badly

- Single Lock for Clock Sweep!
  - fixed in 9.5
- Every Backend performs Clock Sweep
- Algorithm is fundamentally expensive
  - UH, Oh.
  - Worst case essentially is having to touch `NBuffers * 5` Buffers

Clock-Sweep



citusdata

Solution-ish: sweeper process

- perform ClockSweep in background process(es)
- fill ringbuffer of reusable & clean pages
  - lock-less implementation
- tries to keep at least low_watermark reusable buffers
- stops at high_watermark reusable buffers
- fall back to sweep in backends

citusdata

Problem: too many random/small writes

- random writes are slow, even on SSDs
- throughput scales near linearly with request size on SSDs
- always generate random and small writes on > shared_buffers workloads

citusdata

Solution: write combining

- look for neighbouring dirty pages
- write out neighouring dirty pages in file-order
- or as one big write (using pwrite)
- dirty hack: ~40% write throughput in tpc-b like

Problem - Cache Replacement Replaces Badly

- Usagecount of 5 (max) reached very quickly
  - Often all buffers have 5
  - only works well if replacement rate is higher than average usage rate
  - very expensive form of random replacement
- Increasing max usagecount increases cost, the worst case essentially is

  O(NBuffer * max_usagecount)

Possible Solutions

- Increase usagecount intelligently
  - immediately go from 0→1, 1→2
  - separate counter slowing increment from 2→3, 3→4, 4→5
  - always decrement usagecount by one
- Different data-structure / replacement strategy
  - Segmented list based LRU?
  - random replacement?
- Force clock ticks on buffer access
  - expensive for (mostly) cached workloads

Problem: Kernel Page Cache

- Double buffering decreases effective memory utilization
- memory copying between kernel / shared buffers expensive
- have to work around issues with buffered kernel IO
- Avoiding double-buffering makes restarts more expensive

Solutions: Kernel Page Cache

- Hint aggressively to forget pages
  - forgoes extended read cache
  - allow to gift cache contents??? (yes, throw me out)
- Use O_DIRECT?
  - Requires lots of performance work on our side
    - synchronous writes
  - Considerably faster in some scenarios
  - Less Adaptive (resizable shared_buffers)?
  - Very OS specific (to be fast)

PostgreSQL's Buffer Manager Problems & Improvements

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

http://anarazel.de/talks/pgcon-2016-05-20/io.pdf

citusdata