PostgreSQL's IO subsystem: Problems, Workarounds, Solutions

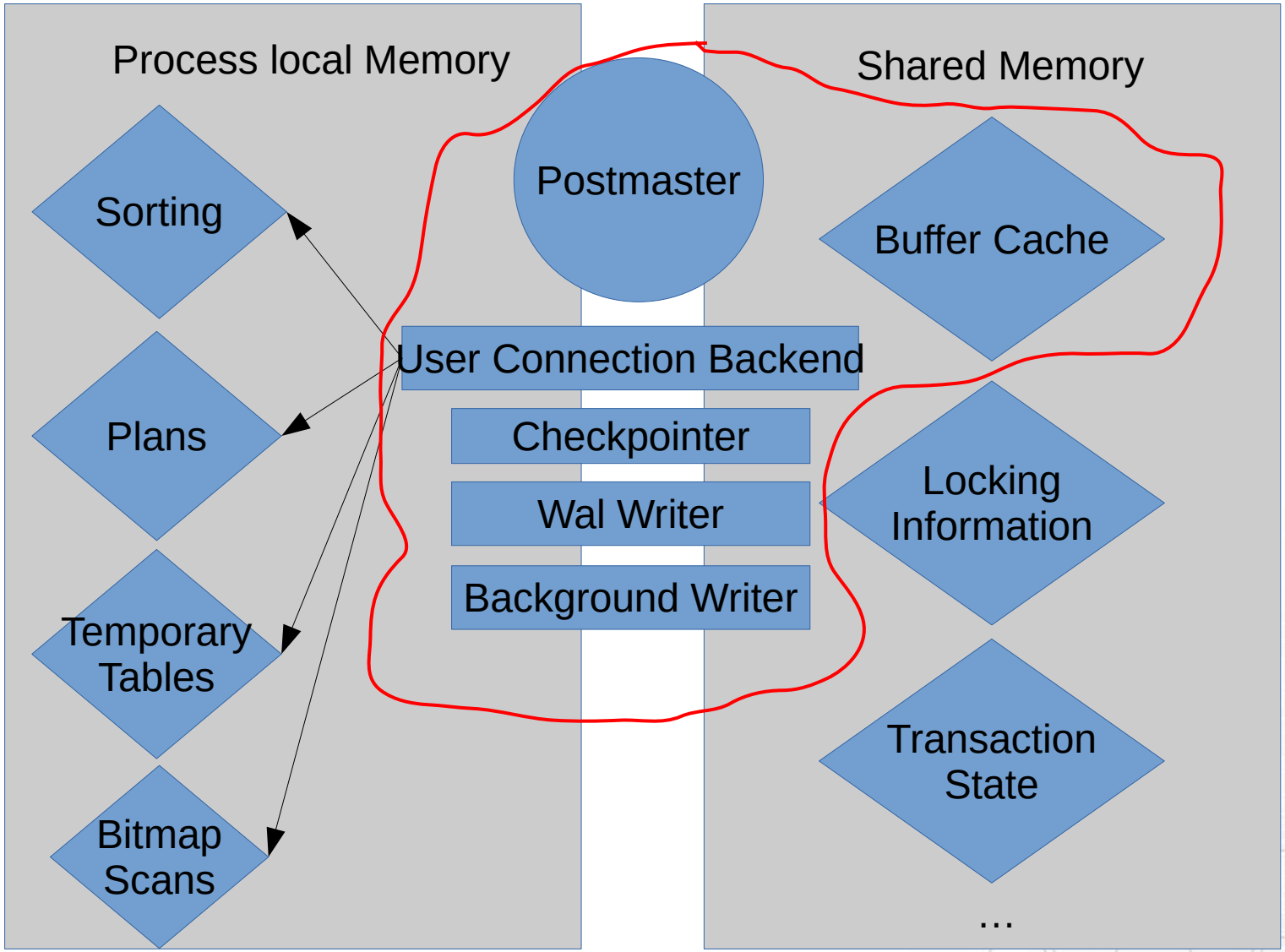
Andres Freund

PostgreSQL Developer & Committer

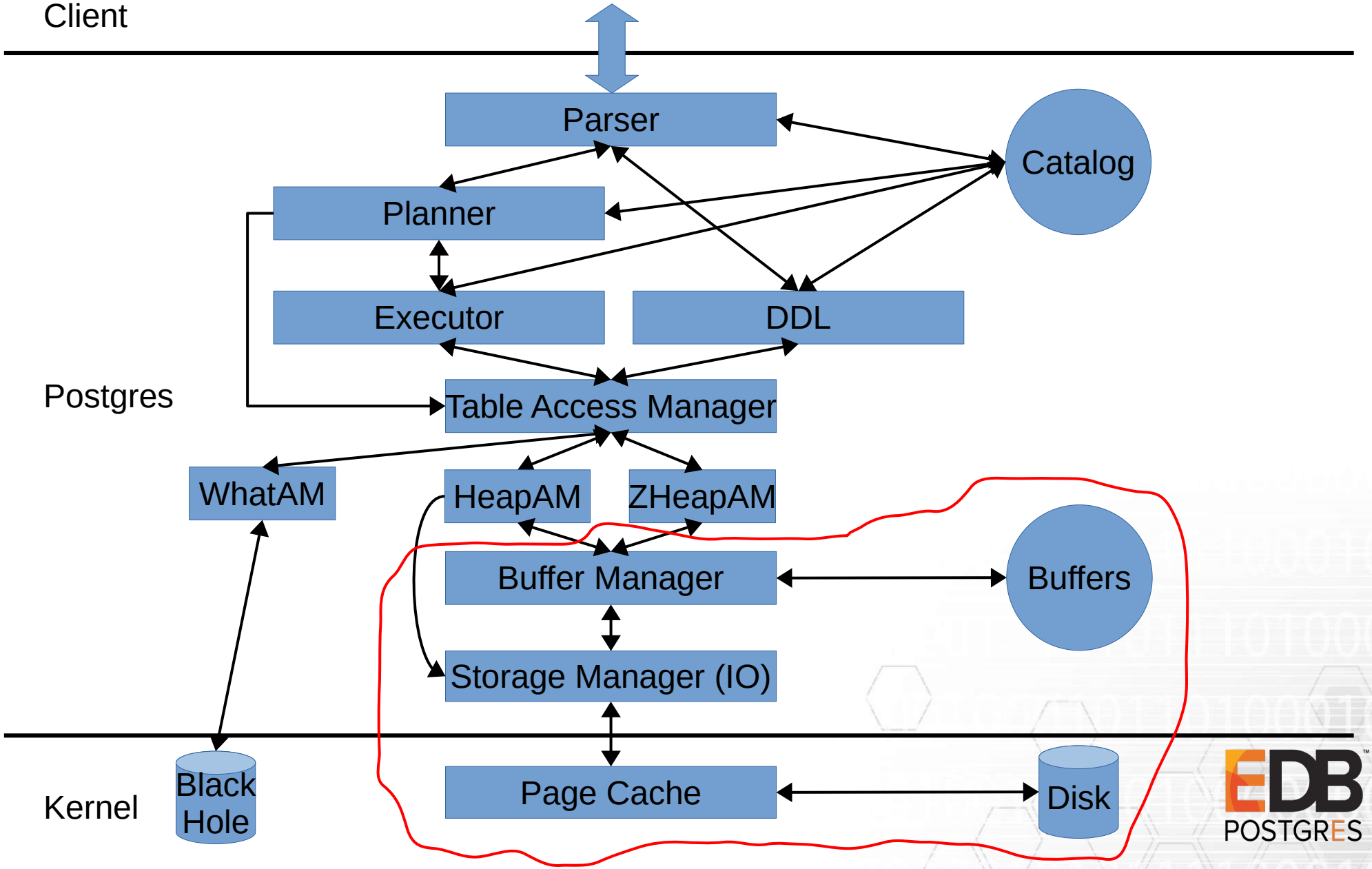
Email: andres@anarazel.de
Email: andres.freund@enterprisedb.com
Twitter: @AndresFreundTec

anarazel.de/talks/2019-10-16-pgconf-milan-io/io.pdf

Memory Architecture



EDB POSTGRES


Client
Parser
Catalog
Planner
Executor
DDL
Postgres
Table Access Manager
WhatAM
HeapAM
ZHeapAM
Buffer Manager
Buffers
Storage Manager (IO)
Black Hole
Page Cache
Disk
Kernel

IO Performance

- Time till IO has finished
- CPU Overhead
  - polling IO, lots of threads, … can be faster, but eat a lot of CPU
- Synchronous Blocking Operation
  - buffered writes: often non-blocking
  - buffered reads: commonly blocking
  - non-buffered writes: blocking & asynchronously
  - non-buffered read: blocking & asynchronously
- Efficiency of IO internally to the drive
  - sequential writes faster than random writes
  - operations covering larger “blocks” of data usually faster
  - deeper queues → higher throughput, higher latency

What Is What

- Backends
  - client connection, or “worker” for parallel query processing
- Checkpointer
  - writes out dirty data once every checkpoint_timeout
  - sorts data before writeout
  - allows to remove / recycle WAL
- Background Writer
  - tries to write out dirty buffers if needed by backends, i.e. working set bigger than memory
- WAL Writer
  - tries to write out WAL generated by backends
  - does most WAL writing when synchronous_commit = off
  - may do a fair bit of WAL writing when most transactions are longer

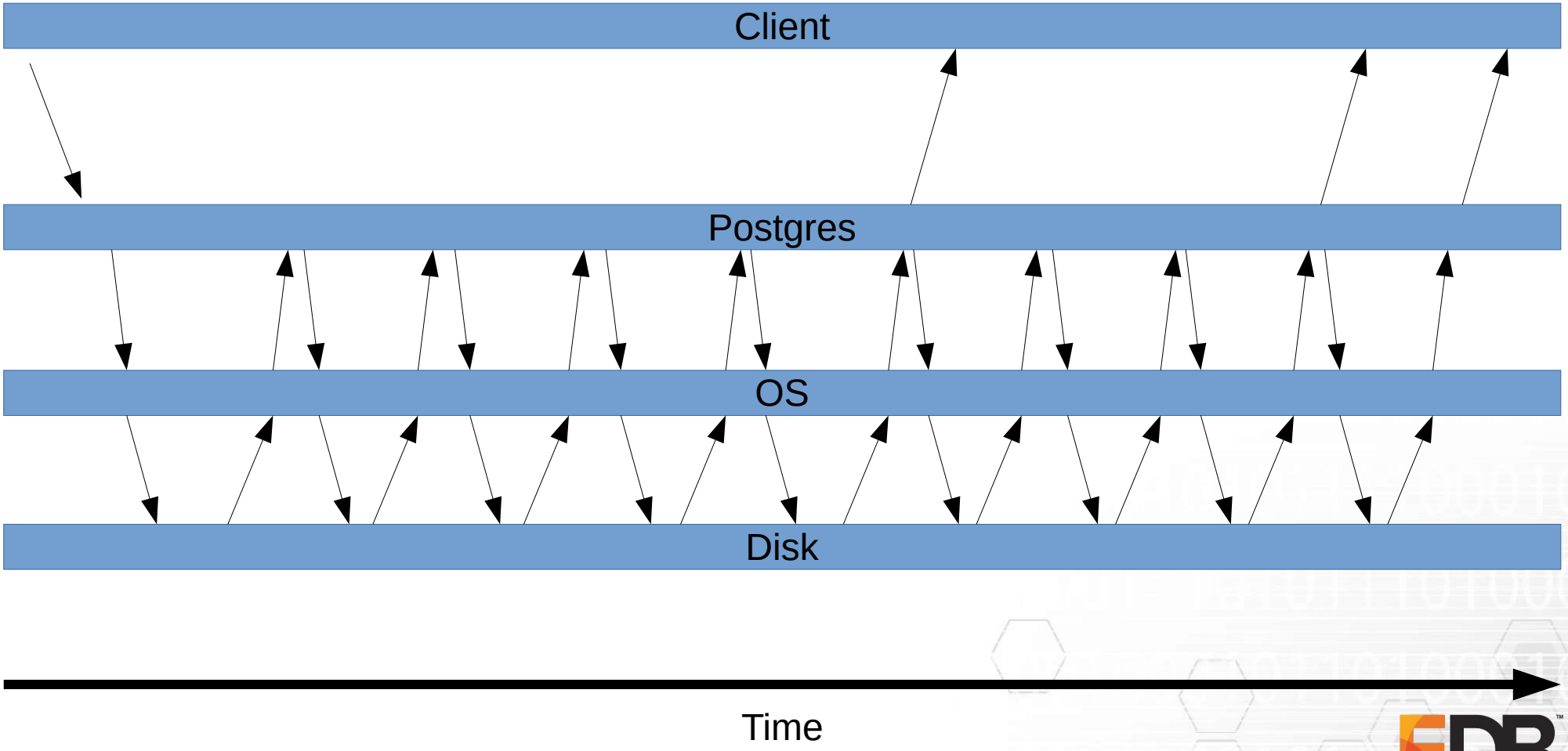
IO Properties

- Backends
  - Data:
    - synchronous random reads
    - triggers read prefetches
    - sequential journal writes
    - under pressure: writeback
  - WAL
    - async append to pre-allocated journal
    - fdatasync on commit
- Checkpointer
  - Data:
    - paced ordered writes (in file order, potentially with lots of gaps)
    - fsyncs all modified files
- Background Writer
  - Data:
    - “writeback”, allows cheap reuse of buffers
    - random writes
- WAL Writer
  - WAL
    - pre flushes WAL
    - commonly purely sequential (potential gaps)

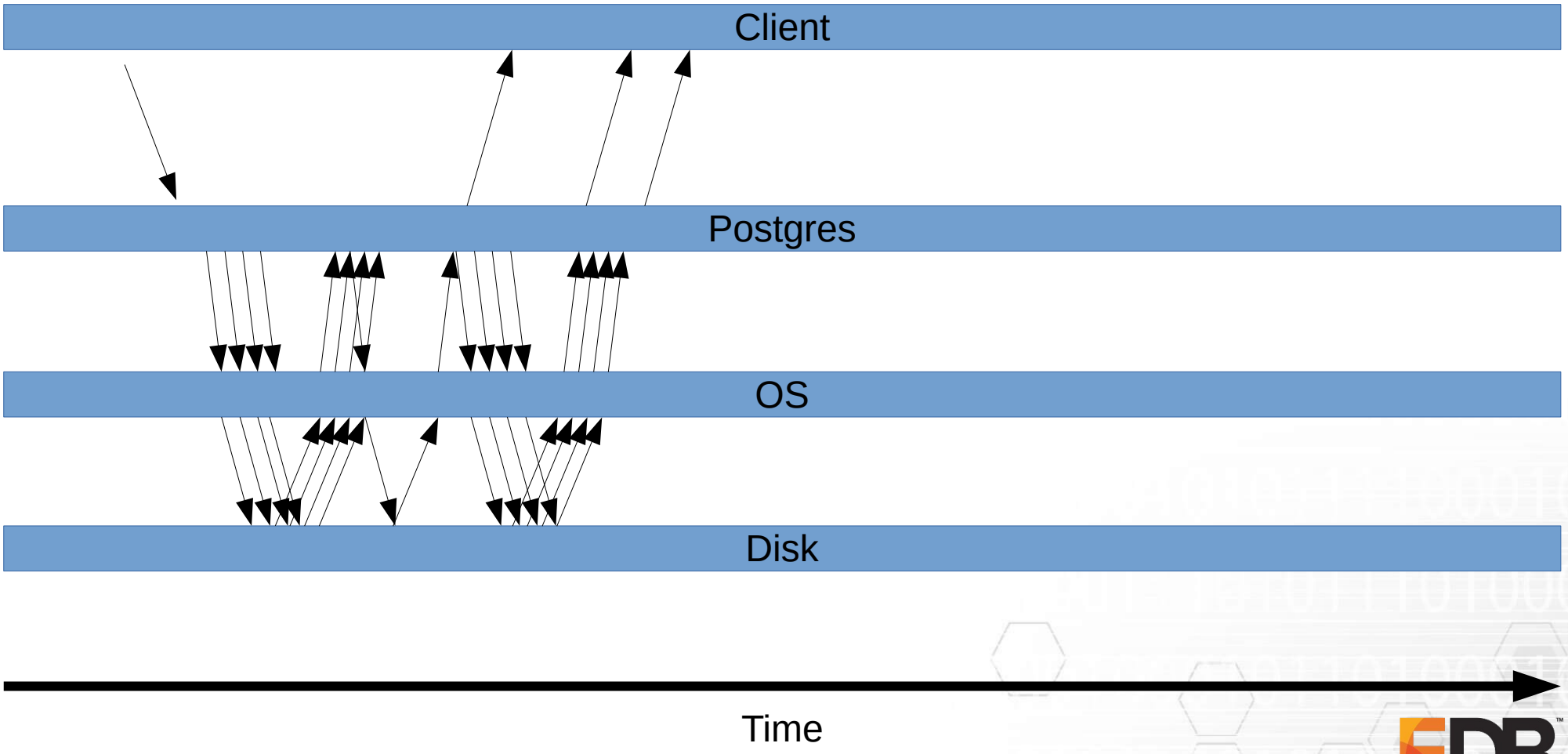
Problem: Postgres Reads

- Very little prefetching
  - partially a problem of the executor
  - partially a problem of the available interfaces
- No concurrent IO
  - especially bad on good SSDs, which can process many many requests in IOs in parallel
- All reads are synchronous
  - the less SQL level concurrency, the worse this is
  - not that bad for nearly entirely cached or very concurrent workloads with just a read or two per “statement”
  - kernel/device to postgres copies are expensive, and not done in parallel
- Workarounds:
  - NVMe SSDs (low latency)

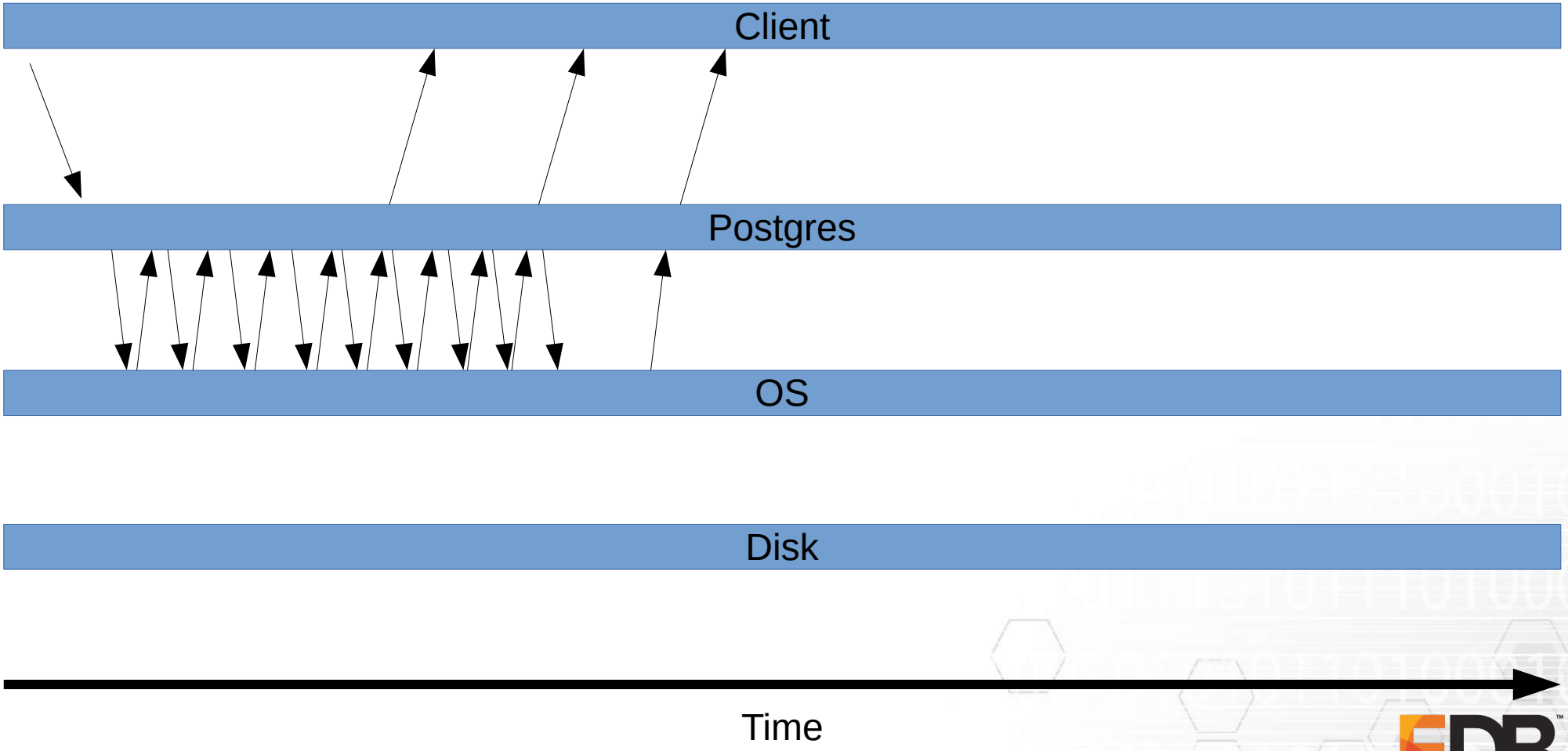
Reads: synchronous, not cached



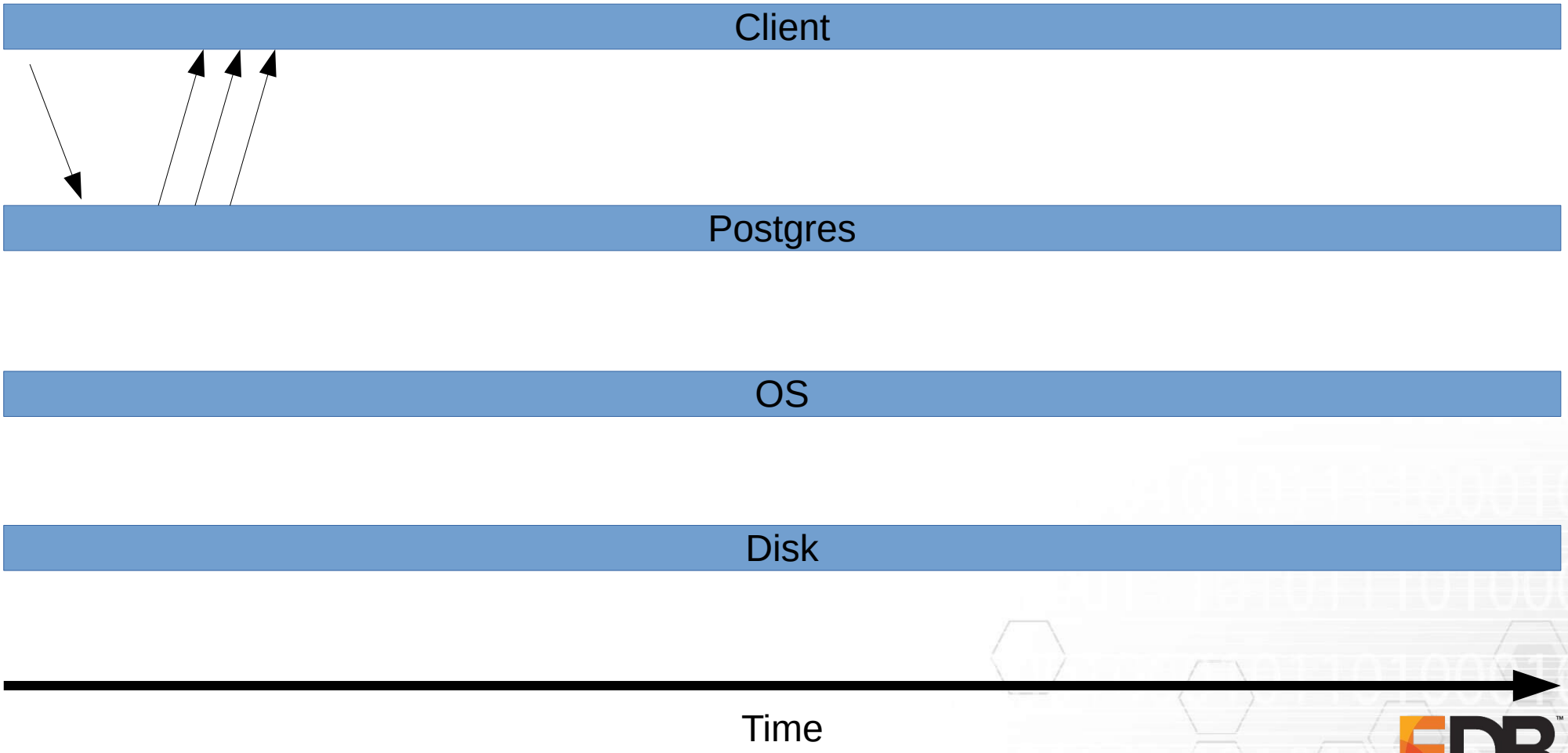
Reads: asynchronous, not cached



Reads: synchronous, OS cached



Reads: synchronous, postgres cached



EDB POSTGRES

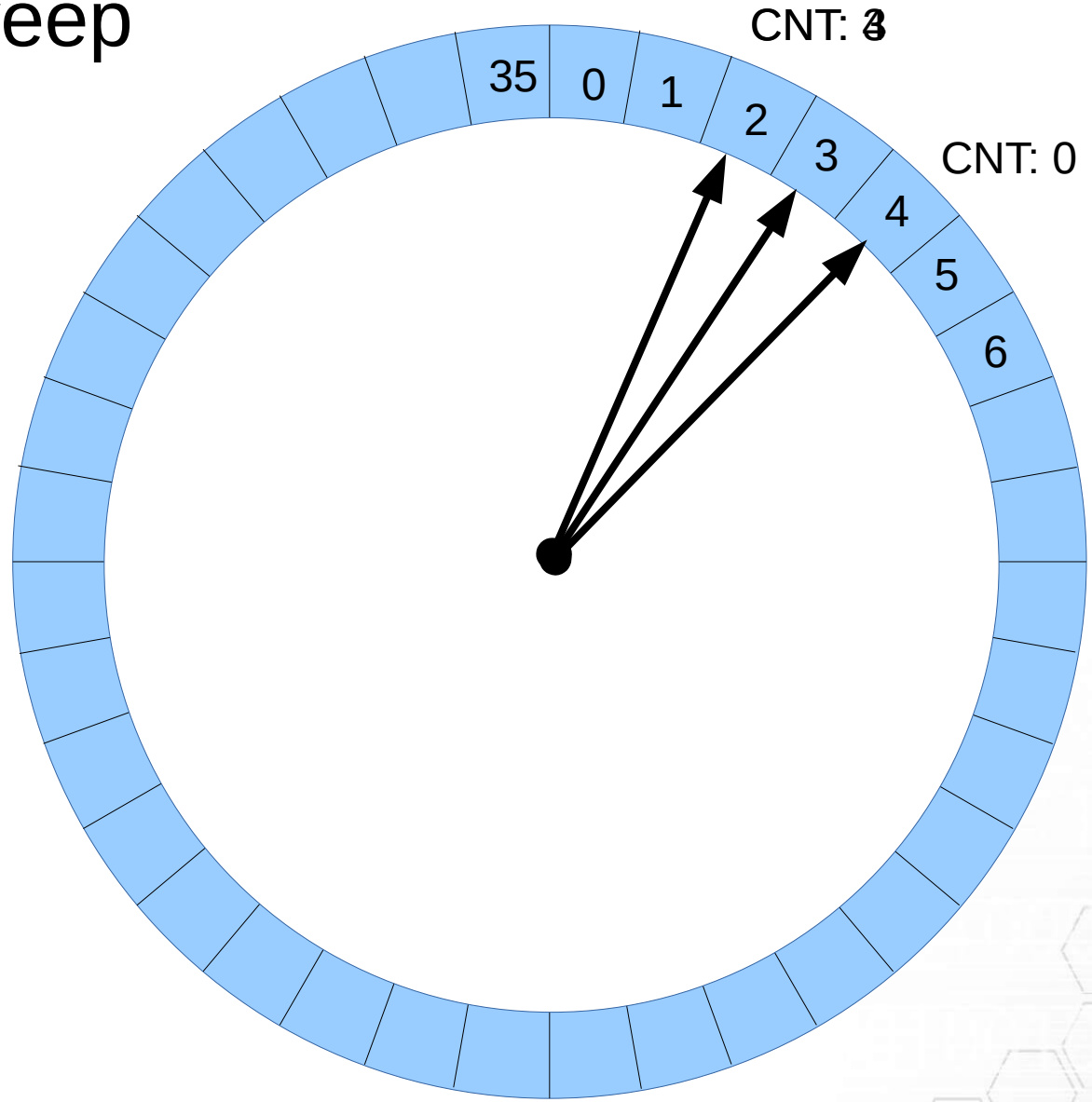
Solution: Postgres Reads

- Add support for asynchronous reads
  - Highly platform dependent
  - typically only supports “direct IO”, i.e. IO bypassing the kernel page cache
    - emulation via fadvise() currently done, but still does kernel→userspace copy
  - linux has new interface, io_uring, that is a lot more flexible
    - including fewer syscalls (important after intel security “fixes”)
  - lots of work
    - including executor architecture
- Emit better prefetching requests
  - not that hard in individual cases, but a lot of places

Problem: Background Writer

- Refresher for bgwriter:
  - writes data back to OS when working set doesn't fit in shared buffers
  - reduces writes needing to be done by backends
- Background writer does not change recency information (perform clock sweep)
  - when all blocks "recently" used → can't do anything
  - configuration complicated & not meaningful
- All IO buffered synchronous
  - throughput / IO utilization too low, and thus falling behind
  - flushes can be disabled, but often causes massive latency issues
- A lot of random IO
  - victim buffer selection usage and buffer pool position dependent
  - hard to efficiently combine writes for neighboring blocks currently (hash mapping)

Clock-Sweep



EDB POSTGRES

Problem: Background Writer

- Consequences:
  - backends to a lot of IO, a lot of it random (slow)
  - high jitter, depending on bgwriter temporarily doing things or not
- Partial Workarounds
  - reduce bgwriter_delay significantly
  - increase shared_buffers and/or decrease checkpoint_timout (all sequential writes)
  - sometimes: set backend_flush_after (for jitter reduction)

Solution: Background Writer

- Perform Clock Sweep
  - avoids inability to find work
  - can actually **improve** recency accuracy (less saturation)
- Queue of clean buffers
  - removes pacing requirements
  - reduces average cost of getting clean buffer
- Asynchronous Writes / Writeback
  - improves IO throughput / utilization, especially with random IO
- Write Combining
  - reduces random IO
  - requires shared_buffer mapping datastructure with ordering support

- Prototype seems to work well

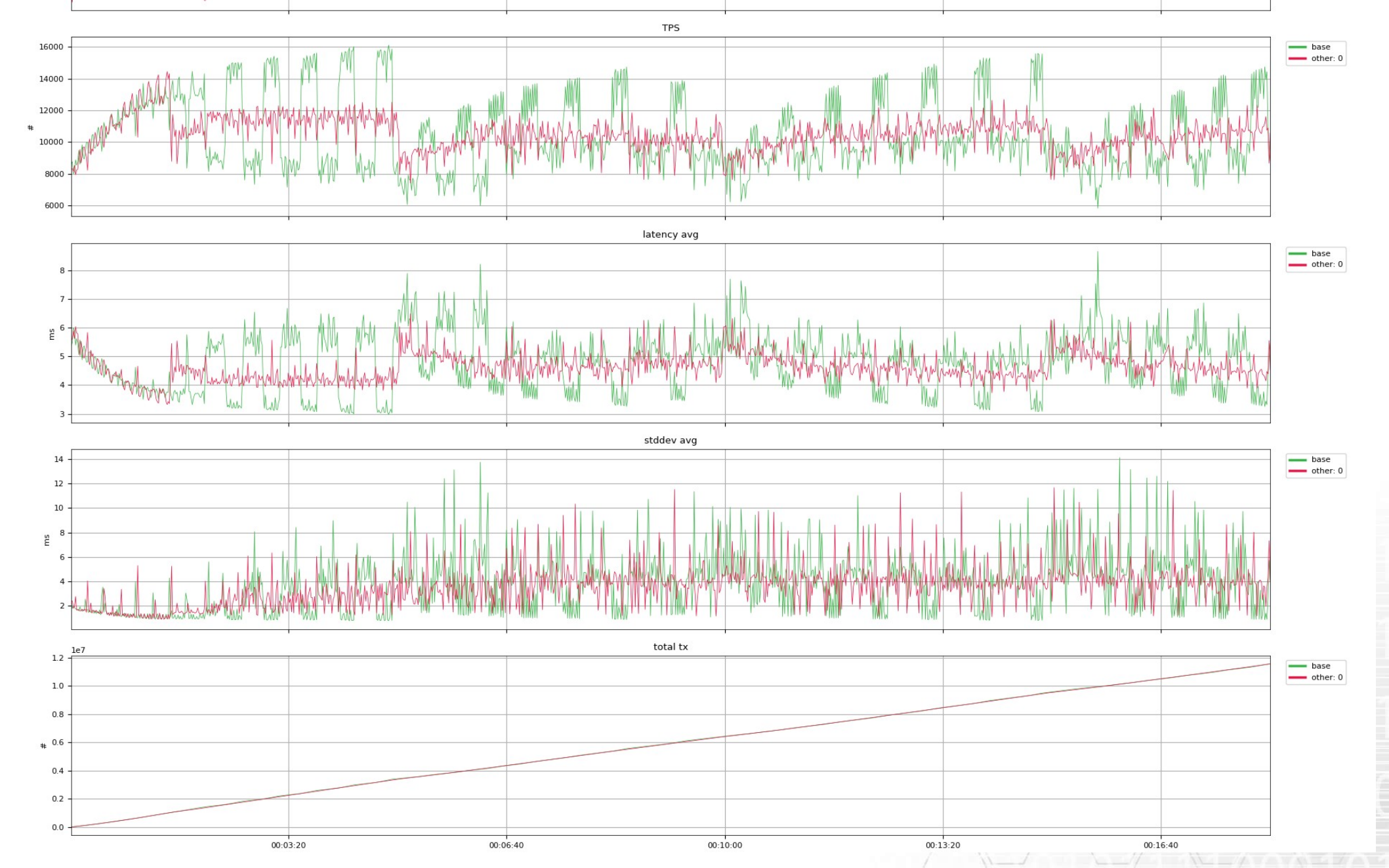

TPS
base
other: 0
#
latency avg
ms
stddev avg
total tx
00:01:40
00:03:20
00:05:00
00:06:40
00:08:20
00:10:00
00:11:40
00:13:20

Problem: Backend Writeback

- takes time away from query execution
- unpredictable latency
  - query - due to having to write
  - write - due to kernel cache
- Diagnose:
  - pg_stat_statements.blk_write_time etc, for readonly queries
  - EXPLAIN (ANALYZE, BUFFERS)
- Workarounds:
  - tune background writer to be aggressive
  - set backend_flush_after
- Solutions:
  - new bgwriter
  - asynchronous direct-IO write submissions

Problem: Jitter

- query performance can be unpredictable
- Causes:
  - kernel has a lot of dirty buffers → decides to write back
  - postgres issues IOs at an unpredictable rate
  - kernel readahead randomly makes reads take longer
- Workarounds:
  - set backend_flush_after, reduce other *_flush_after settings
  - disable kernel readahead (can be bad for sequential scans)
  - make bgwriter more aggressive
- Solutions:
  - disable kernel readahead, perform our own readahead / prefetching
  - prioritize / throttle different IO causes differently
  - improve cache hit ratio


TPS
16000
14000
12000
10000
8000
6000
#
base
other: 0
latency avg
8
7
6
5
4
3
ms
base
other: 0
stddev avg
14
12
10
8
6
4
2
ms
base
other: 0
total tx
1e7
1.2
1.0
0.8
0.6
0.4
0.2
0.0
#
base
other: 0
00:03:20
00:06:40
00:10:00
00:13:20
00:16:40

Why Buffered IO?

- Parts of Postgres' IO stack have, uh, baggage
- Portability
- Needs far less tuning
  - PG buffer cache size less critical, extends to kernel page cache
  - IO issue rate to drive doesn't need to be controlled
- Why is having less tuning crucial:
  - DBAs / sysadmins don't exist for vast majority of systems (if they exist, they don't know hardware that well)
  - workloads continuously change
  - machines / OS instances are heavily over-committed and shared
  - adapting shared memory after start is hard (PG architecture, OS)
- Consequence
  - PG defaults to 128MB shared buffers ("page cache")
  - works OK for low-medium heavy load

Why Direct IO?

- **Much** higher IO throughput, especially for writes
- locking for buffered writes limits concurrency
- no AIO without DIO for most platforms (except now io_uring)
- No Double Buffering
- Writeback behavior of various OS kernels leads to hard to predict performance
- kernel page cache scales badly for large amounts of memory
- kernel page cache lookups are not cheap, so need to be avoided anyway (copy_to_user + radix tree lookup, syscalls after security fixes)

Further Problems

- Available postgres level monitoring incomplete and confusing
  - pg_stat_bgwriter has stats not about bgwriter
  - buffers_backend includes relation extension, which cannot be done by any other process
  - write times of different processes not recorded (checkpoint_write_time useless, includes sleeps)
- Ring Buffers for sequential reads, vacuum, COPY can cause very significant slowdowns
  - data never cached
  - writes quickly trigger blocking
- Double Buffering
  - `trigger posix_fadvise(POSIX_FADV_DONTNEED)` when dirtying?