

AIO performance

(now and in the future)

Tomas Vondra
<tomas@vondra.me>



Agenda

do expect

- a rough overview of AIO performance
- comparisons to previous releases (charts!)
- general findings and a bit of tuning advice
- some future ideas / plans

do not expect

- deepdive into AIO internals
- code examples

Agenda

- introduction to AIO
 - what's the point of AIO?
 - it's not (just) about performance
- AIO timeline
- some comparisons to non-AIO behavior
 - compared to Postgres 16+ (last non-async)
 - compared to `io_method=sync`
 - some direct I/O numbers (but it's a "debug" feature)
- AIO future
 - index prefetching (yay!), writes

What's the point of AIO?

- synchronous I/O
 - issue I/O when a block of data is needed
 - may result in a lot of waiting
 - problem on high-latency storage (cloud), or storage requiring deep queues (NVMe)
- async I/O
 - separate when an I/O is issued and consumed
 - issue the I/O earlier, before it's needed
 - gives the system time to handle it
 - ideally, no wait when the block is actually needed

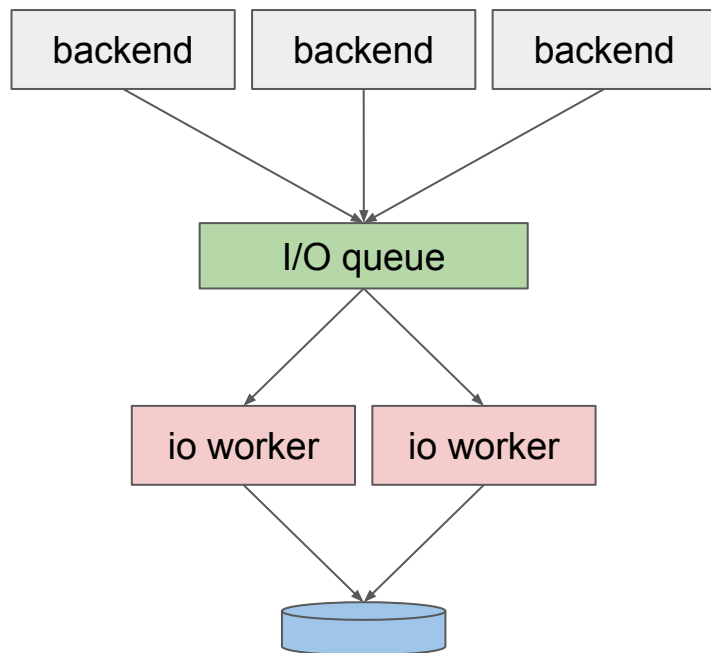
What's the point of AIO?

- concept vs. architecture
 - "asynchronous I/O" is a general concept, can be implemented in many ways
 - AIO is how Postgres implements it (specific architecture)
- `posix_fadvise`
 - a way to do asynchronous I/O without AIO
 - we had that since "forever", but comes with severe limitations
 - doesn't work for writes, architectural issues (e.g. index prefetching), direct I/O
- primary goal: give more control to the database
 - allow features that'd be impossible otherwise
 - explicit separation of queueing and processing a request
 - improved performance is second-order effect

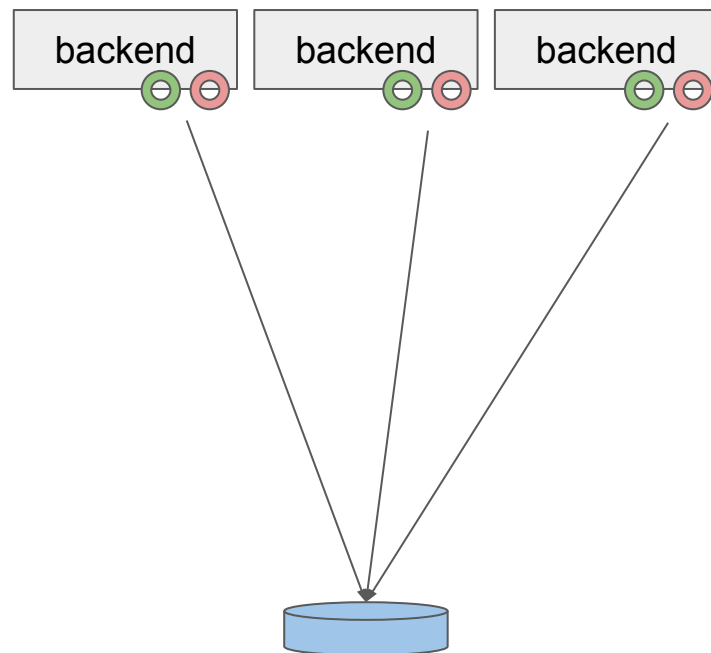
Timeline

- 17 - first pieces of infrastructure
 - but mostly just API work, no behavior change
 - still just `posix_fadvise`, just wrapped differently
- 18 - the "actual" AIO
 - `sync`, `worker`, `io_uring`
 - more places using the new API (some switch from `posix_fadvise`)
- 19 - various improvements
 - scalability, better heuristics, ...
- 20dev - index prefetching (?)
 - ...
- future

io_method = worker



io_method = io_uring



Questions

- What performance should you expect with AIO?
 - compared to non-AIO releases
- How do the I/O methods compare?
 - sync, worker, io_uring
- Various observations
 - number of workers
 - when it can't help
 - ...

Benchmarks

- I'll talk about 3 basic scan operations
 - seqscan, bitmap scan, index scan
- easier to isolate the I/O effect
 - complex queries often benefit from other improvements
 - simple queries are not 100% immune to that either
- hardware-specific behavior
 - bandwidth, latency, concurrency, I/O depth
 - will try to present "typical" behavior
 - local NVMe and network (EBS-like) storage

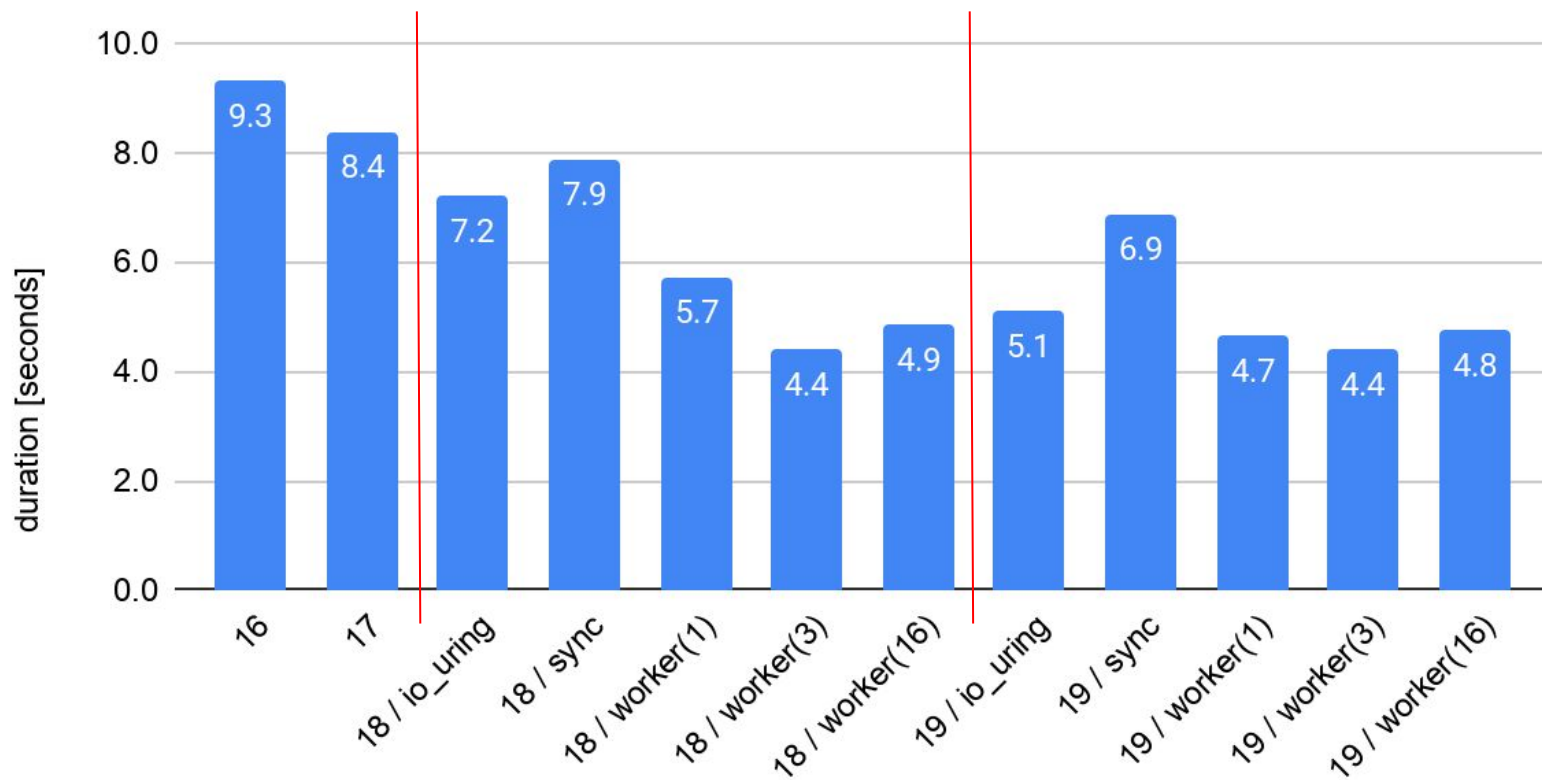
Systems

- azure / NVMe
 - cloud instance
 - local (ephemeral) NVMe devices
 - 2x RAID0
 - 2.2GB/s bandwidth
- ryzen / NVMe
 - my "home" test system
 - 4x NVMe RAID0
 - 25GB/s bandwidth

sequential scans

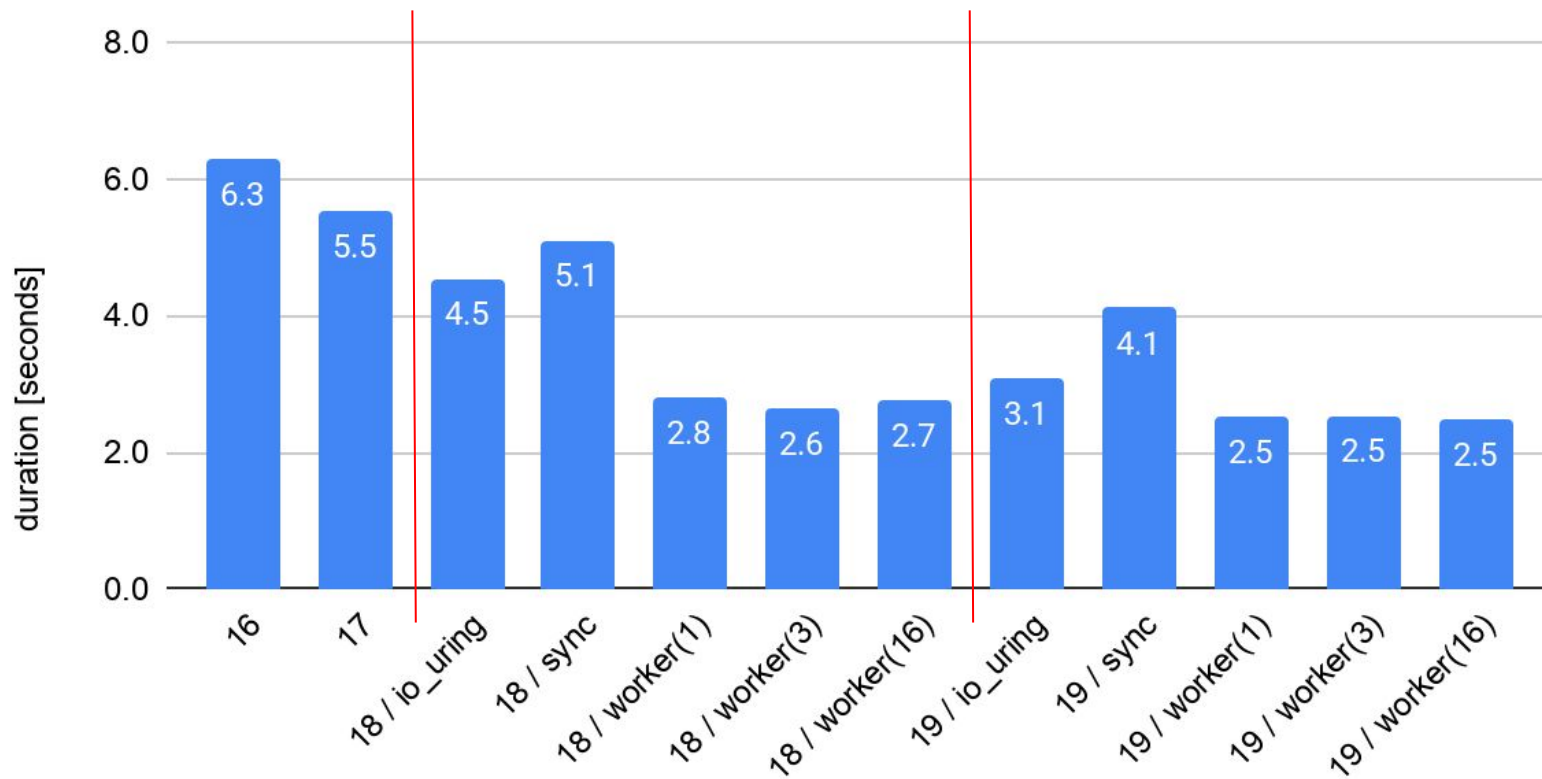
seqscan, 32GB, buffered I/O [cold cache]

Ryzen 9980X, 64GB RAM, 4x NVMe (RAID0), kernel 6.12



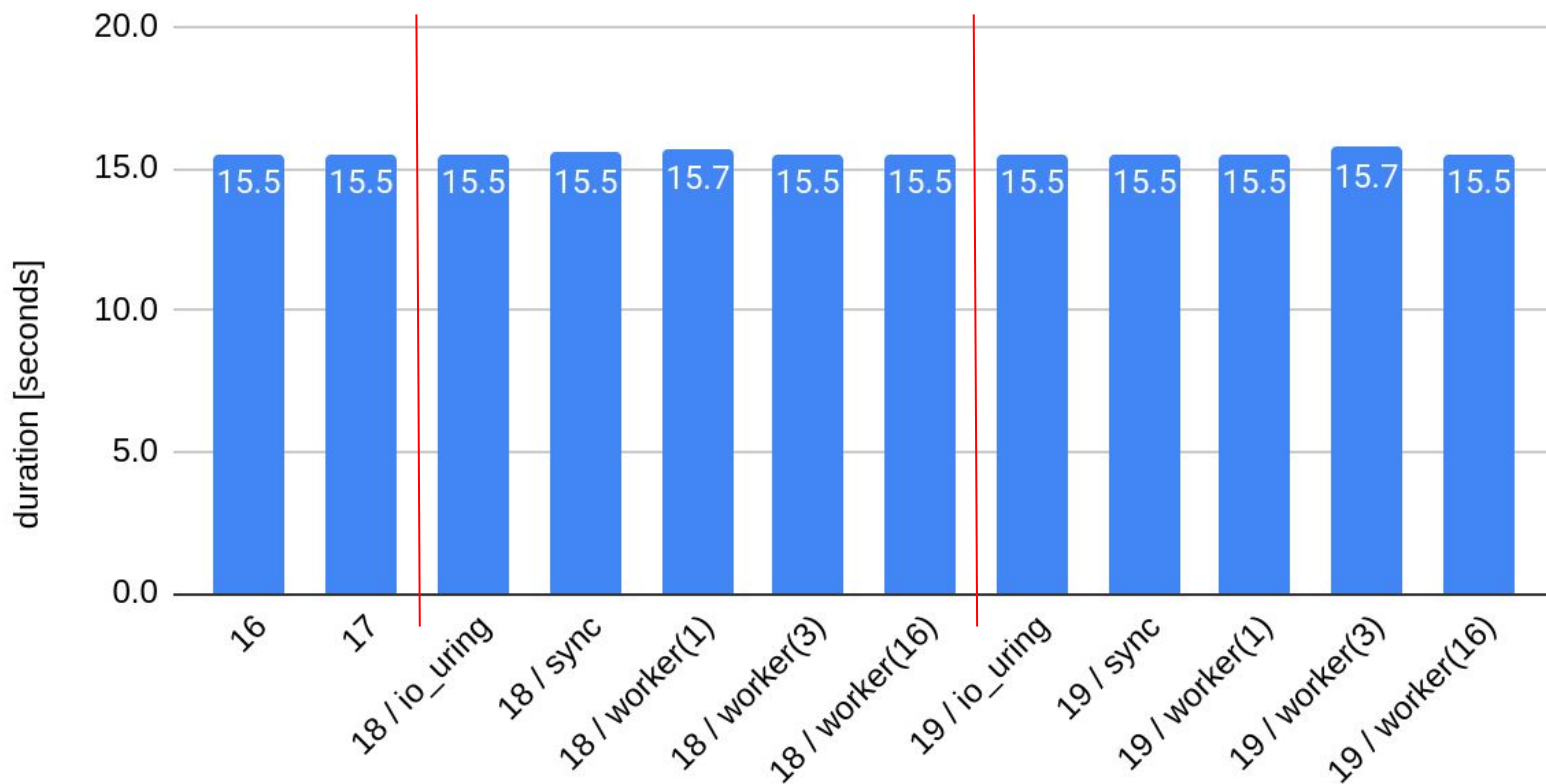
seqscan, 32GB, buffered I/O [warm cache]

Ryzen 9980X, 64GB RAM, 4x NVMe (RAID0), kernel 6.12



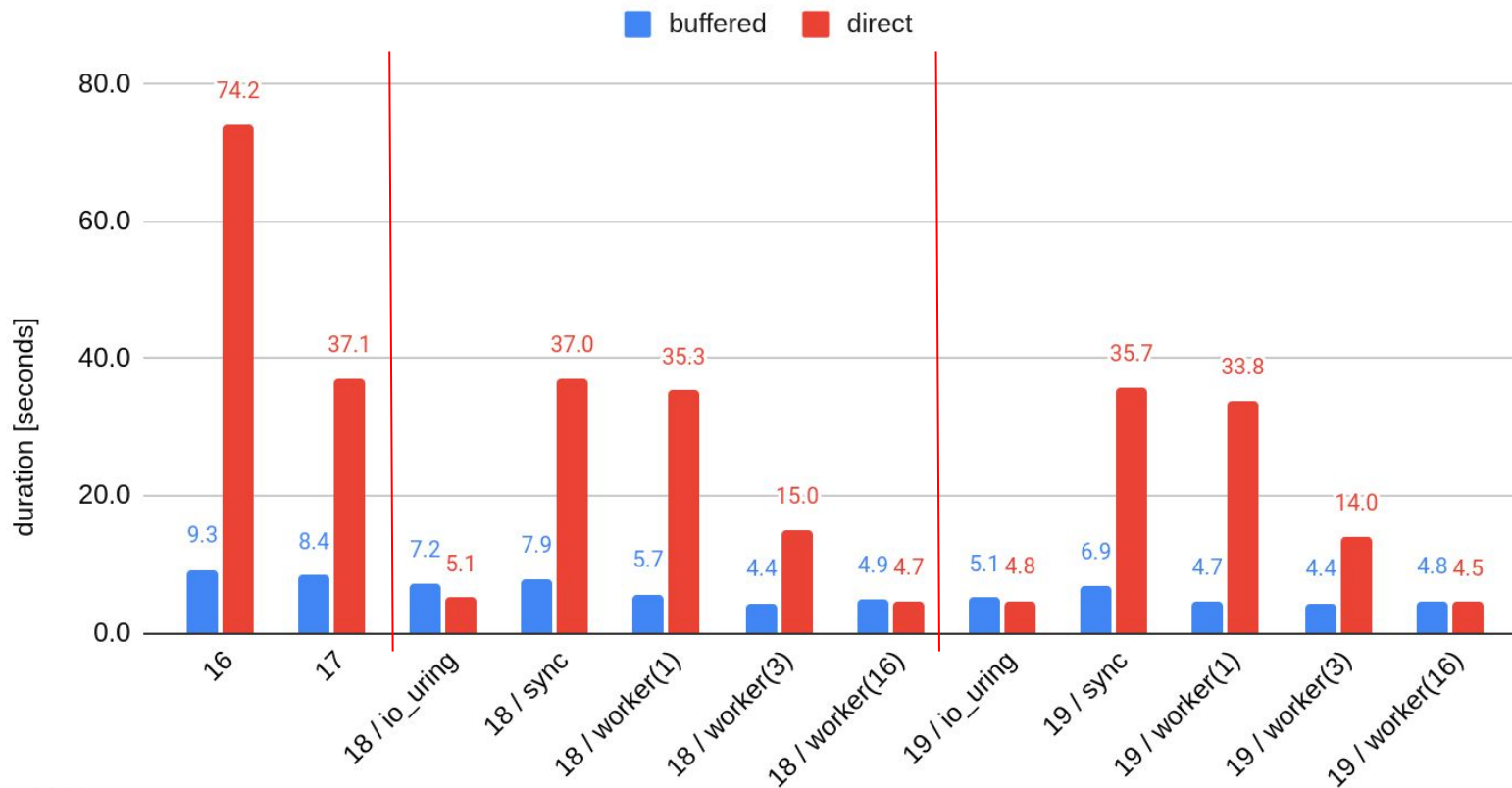
seqscan, 32GB, buffered I/O [cold cache]

AMD EPYC 9V45, 16 VCPUs, 64GB RAM, 2x NVMe (RAID0), kernel 6.18



seqscan, 32GB, buffered vs. direct I/O [cold cache]

Ryzen 9980X, 64GB RAM, 4x NVMe (RAID0), kernel 6.12



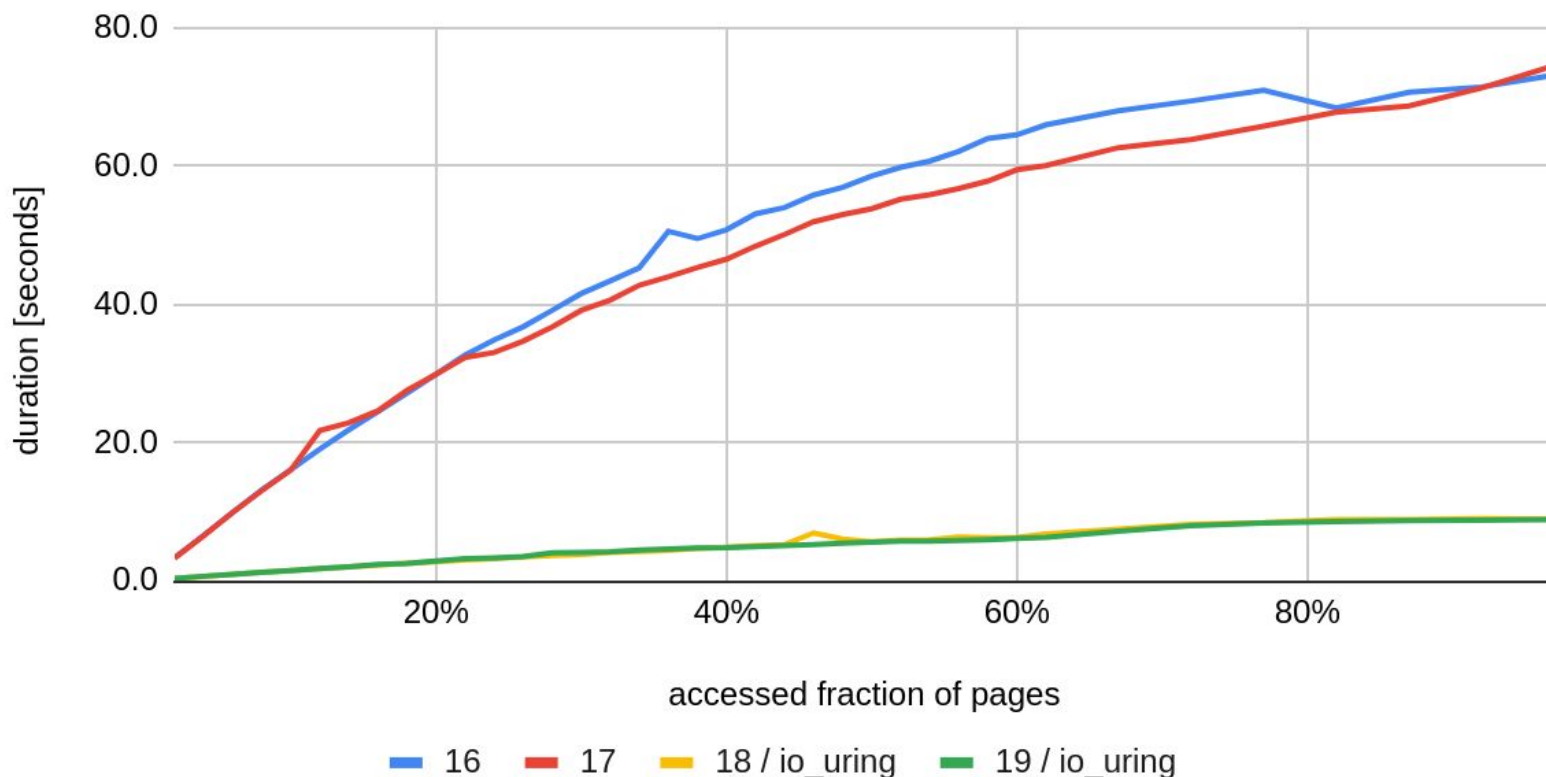
Sequential scans

- big improvement 16 -> 17
 - read stream introduction (not "proper AIO")
 - likely thanks to I/O combining
- stable behavior since 18
 - "sync" slow, other methods about the same
 - cached shows advantage of `io_method=worker` (`memcpy/checksums`)
- direct I/O
 - often substantially slower, but `io_uring` / `worker` are on par with buffered
 - on Azure the differences are much smaller (hitting I/O bandwidth)

bitmap scans

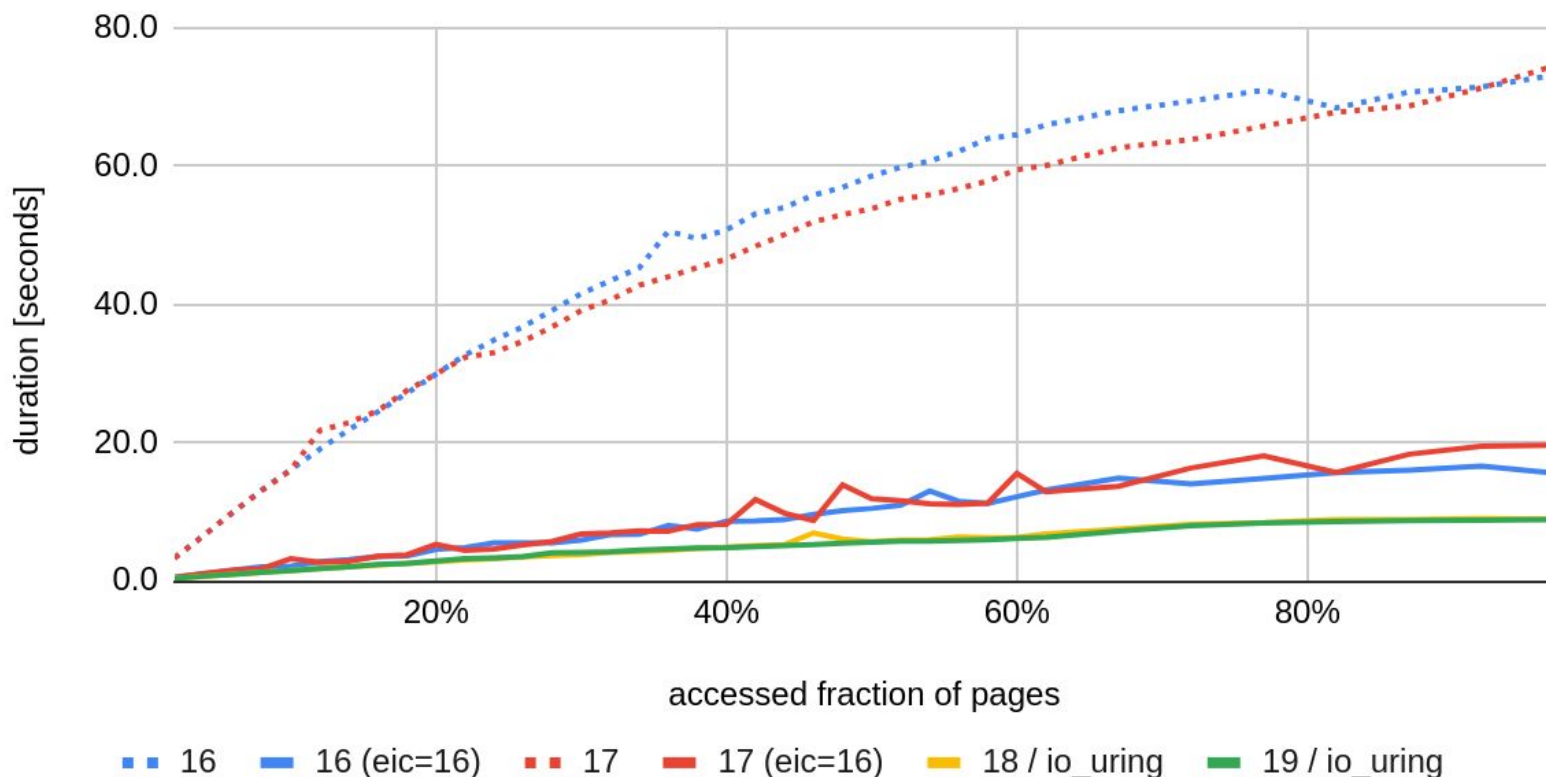
bitmap scans, 14GB, buffered I/O [cold cache]

AMD EPYC 9V45, 16 VCPUs, 64GB RAM, 2x NVMe (RAID0), kernel 6.18



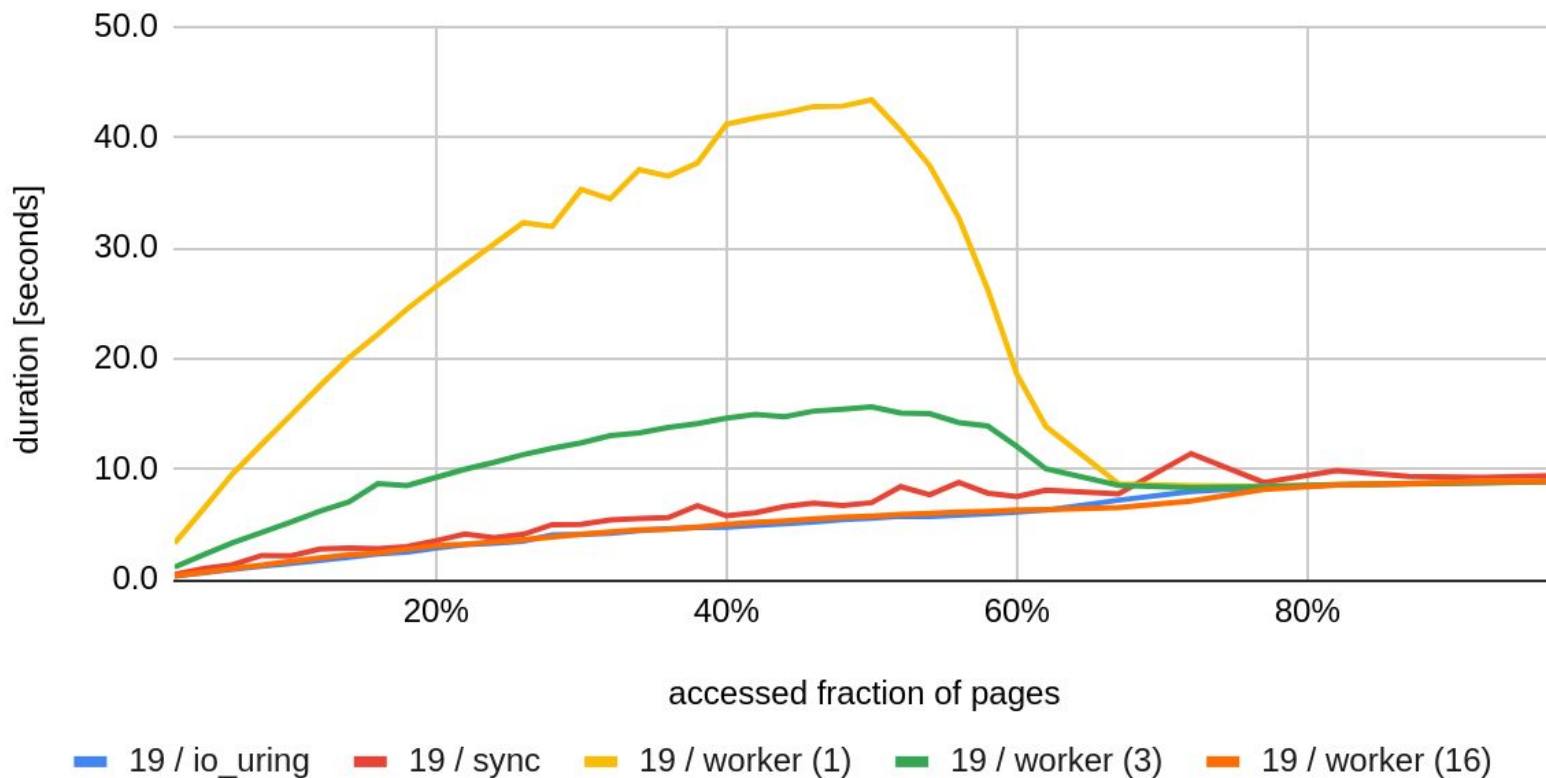
bitmap scans, 14GB, buffered I/O [cold cache]

AMD EPYC 9V45, 16 VCPUs, 64GB RAM, 2x NVMe (RAID0), kernel 6.18



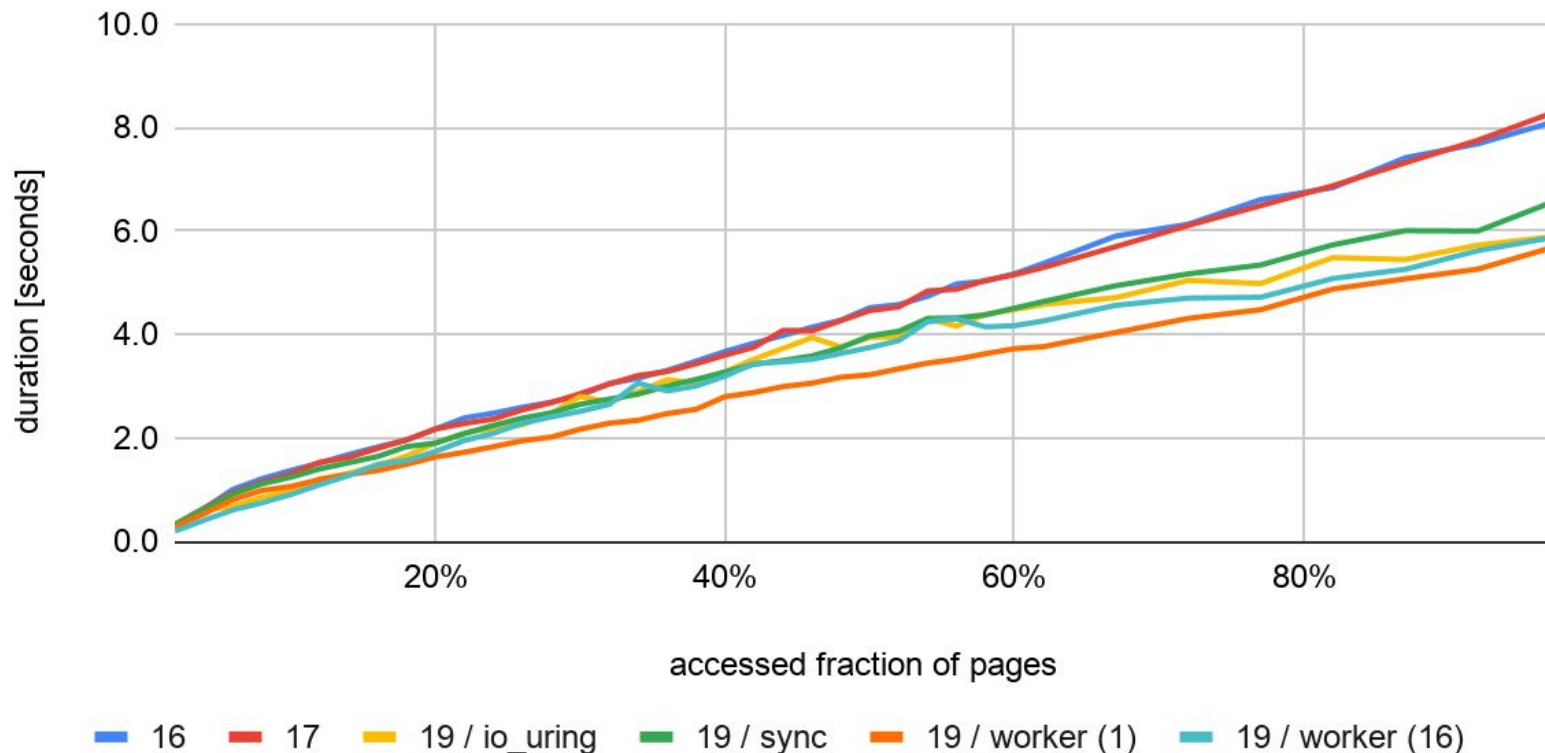
bitmap scans, 14GB, buffered I/O [cold cache]

AMD EPYC 9V45, 16 VCPUs, 64GB RAM, 2x NVMe (RAID0), kernel 6.18



bitmap scans, 14GB, buffered I/O [warm cache]

AMD EPYC 9V45, 16 VCPUs, 64GB RAM, 2x NVMe (RAID0), kernel 6.18



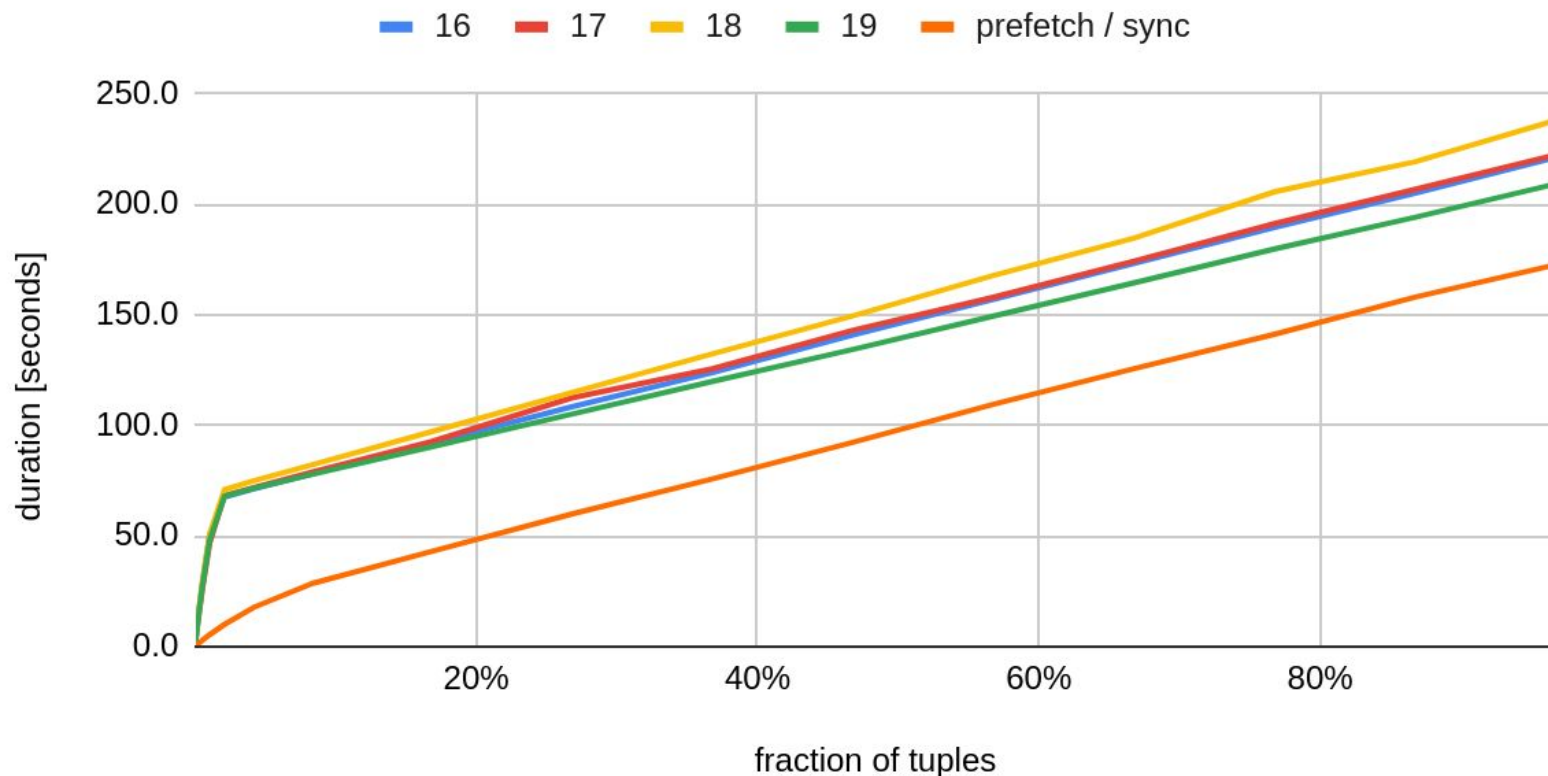
Bitmap scans

- `effective_io_concurrency = 16`
 - gets you fairly close to AIO (10x -> 2x slower)
- `worker`
 - number of workers matters a lot (default 3 is not enough)
- `cached`
 - AIO helps a lot (~20%) compared to 16/17 (surprising)
 - small differences between AIO methods
 - "worker" wins, thanks to offloading memcpy/checksums

index scans

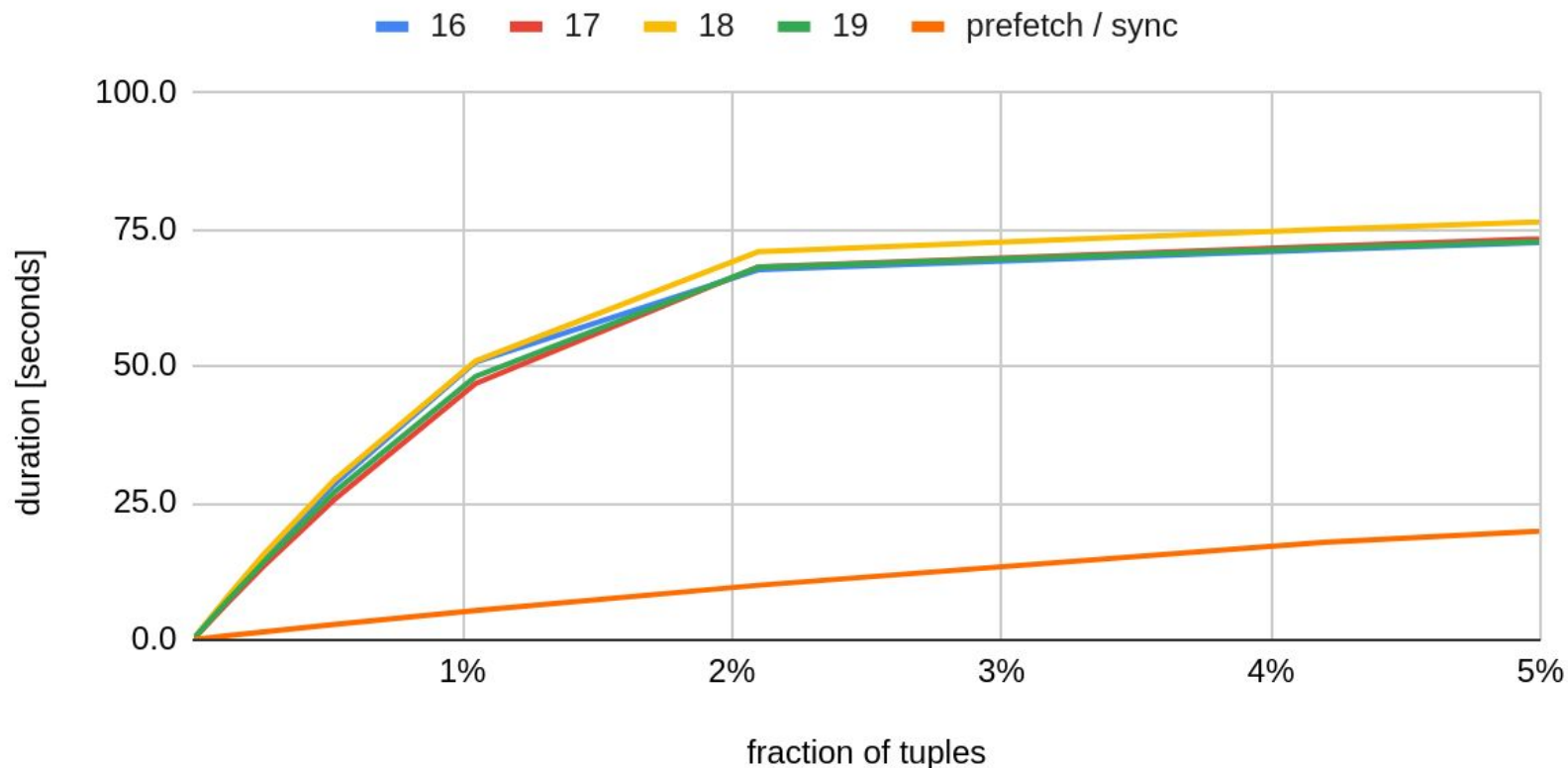
index scan, 14GB [cold cache]

Ryzen 9980X, 64GB RAM, 4x NVMe (RAID0), kernel 6.12



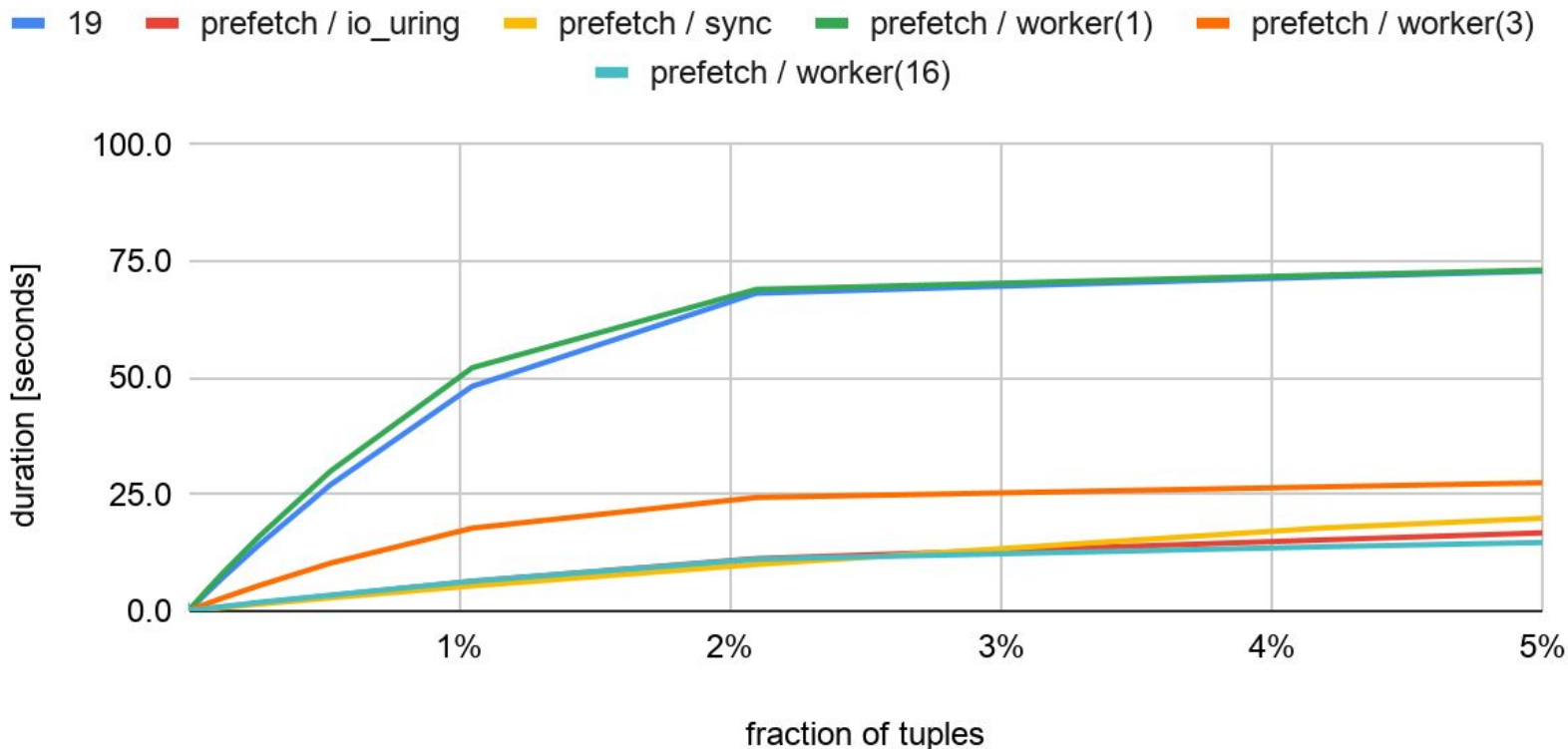
index scan, 14GB [cold cache]

Ryzen 9980X, 64GB RAM, 4x NVMe (RAID0), kernel 6.12



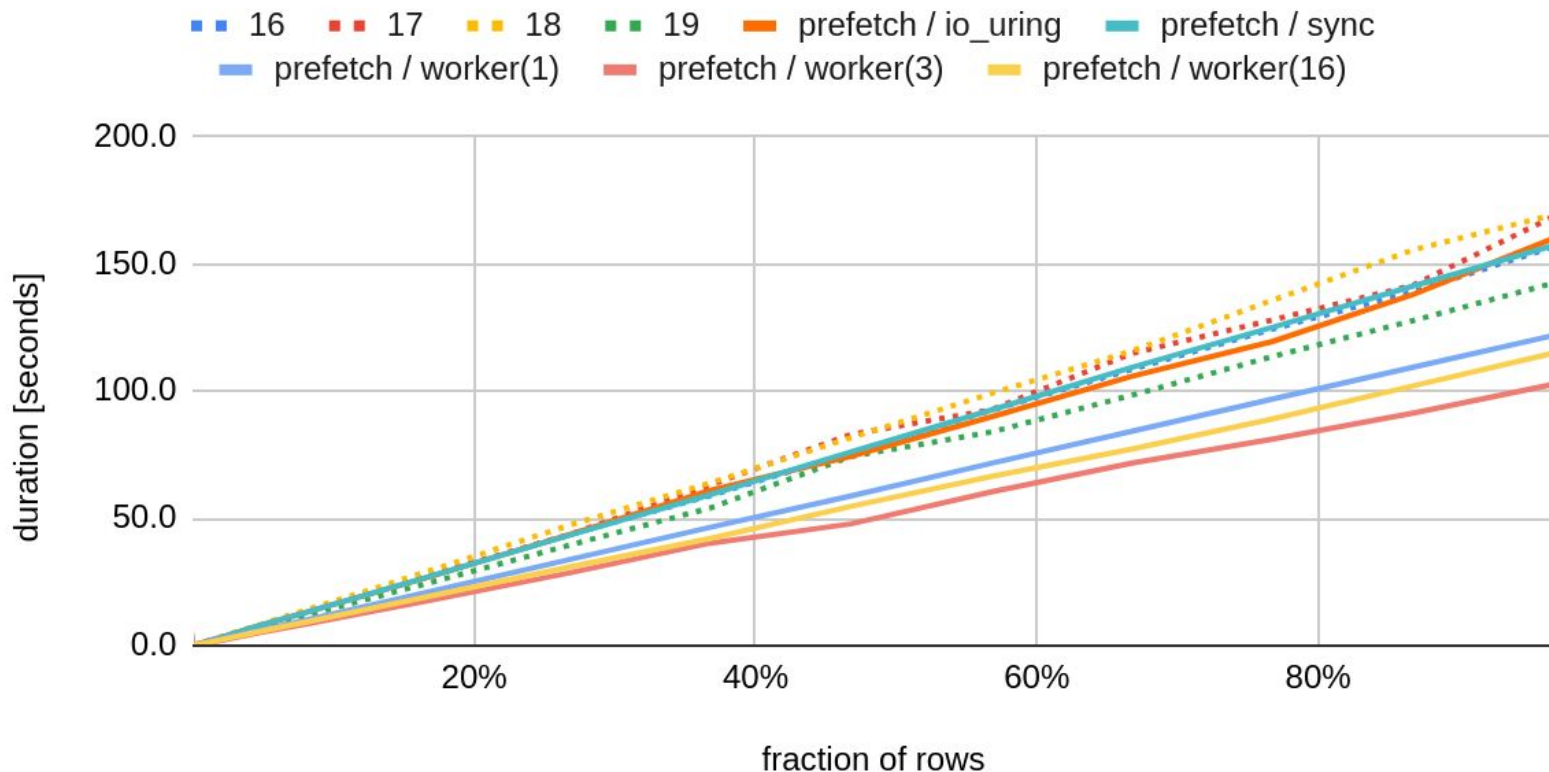
index scan, 14GB [cold cache]

Ryzen 9980X, 64GB RAM, 4x NVMe (RAID0), kernel 6.12



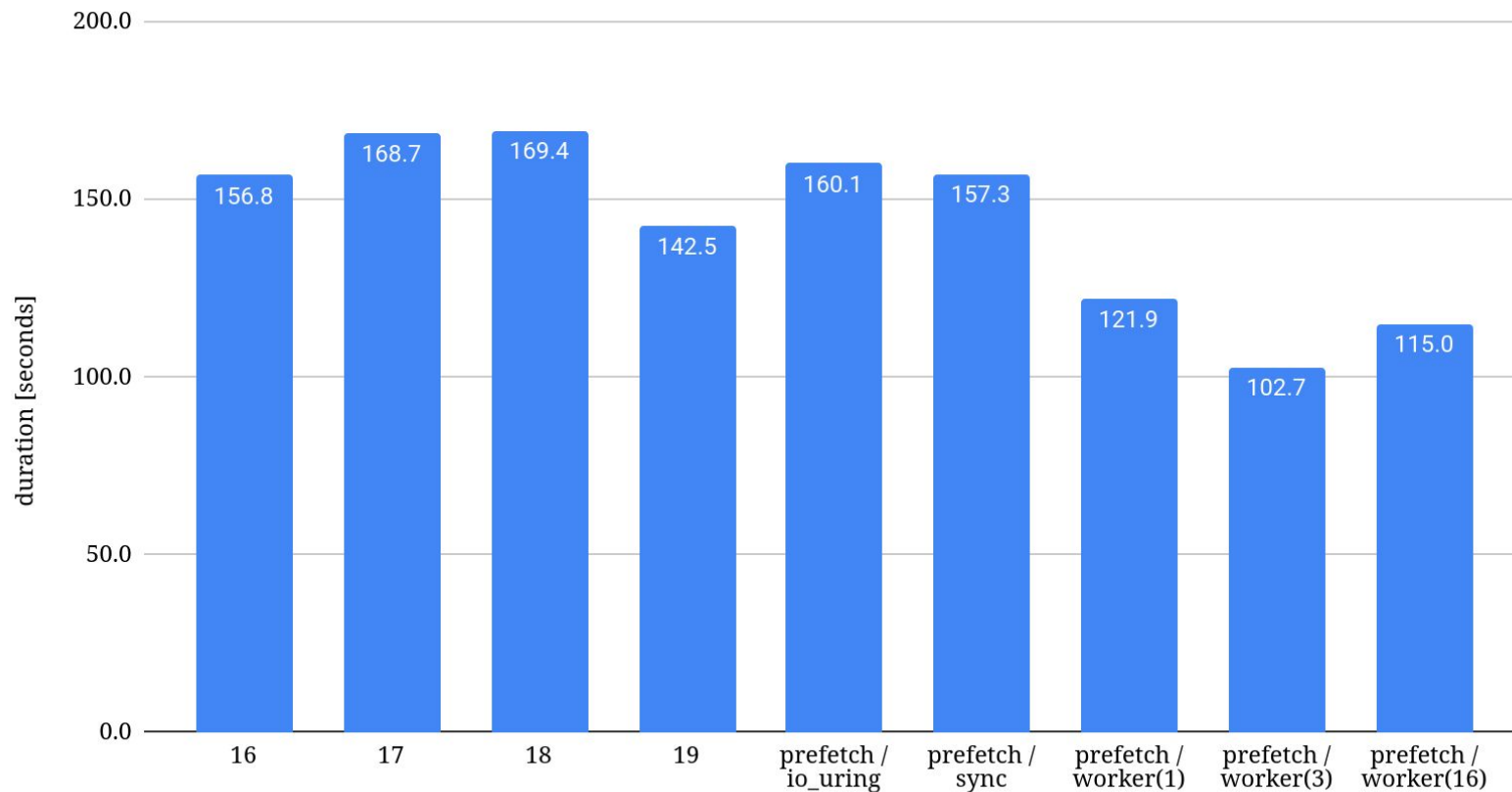
index scan, 14GB [warm cache]

Ryzen 9980X, 64GB RAM, 4x NVMe (RAID0), kernel 6.12



index scan, 14GB [warm cache]

Ryzen 9980X, 64GB RAM, 4x NVMe (RAID0), kernel 6.12



Index scans

- AIO brings massive speedup
 - charts show ~10x improvement
 - we've seen ~30-100x plausible/practical examples
- cold runs
 - modest differences between methods
 - assuming there's enough workers
- warm runs
 - worker method have advantage (memcpy/checksums offload)

Conclusion

Conclusions

- `io_uring` is great
 - very deep I/O queues (important for NVMe)
 - worker still wins in cached cases (which is very common)
 - also, restricted in containers because of security issues :-)
- number of I/O workers matters
 - the default "3" is way too low, fallback to backend may not always apply
 - I'd just bump it to 16 or 32 (especially with the adaptive behavior in 19)
 - parallel worker ~ I/O worker
- AIO can help even with cached workloads
 - may seem a bit surprising (no "real I/O")
 - but depends on how you define "I/O" (if it includes the memcpy, checksums, ...)

Conclusions

- `effective_io_concurrency = 16`
- `io_min_workers = 2`
`io_max_workers = 8`

Future

Future

- index prefetching (still)
 - hopefully PG 20 (fingers crossed)
 - https://www.youtube.com/watch?v=7hbxw0W_9gQ
 - <https://www.postgresql.eu/events/pgconfeu2026/schedule/session/8163-an-overview-of-index-prefetching/>
- maybe improved worker method?
 - limited to 32 workers, with 1 IO per worker (so 32 IOs is not enough)
 - trivial to trigger fallback to "sync" (backends doing their I/O)
 - can't add more workers - contention for the queue (locking, ...)
 - likely queue partitioning (multiple queues + separate worker pools)
 - maybe some worker/io_uring hybrid (worker per queue), or threading?

Writes

- AIO writes in bgwriter & checkpointer
 - 2-3x checkpoint speed for sequential data
 - bigger for large amounts of random data
- AIO for WAL writes
 - bulk load, VACUUM, high-concurrency OLTP
 - hard
- more future stuff in Andres' talk
 - <https://anarazel.de/talks/2025-10-23-pgconf-eu-aio-in-PG-18-and-beyond/aio-in-PG-18-and-beyond.pdf>
 - https://www.youtube.com/watch?v=rw_8HaAOQnY
- Melanie mentioned write combining
 - <https://www.postgresql.eu/events/pgconfeu2026/schedule/session/8341-exercising-postgresql-performance-enhancing-patches/>

Resources

- aio in PG 18 and beyond [Andres Freund, pgconf.eu 2025]
 - <https://anarazel.de/talks/2025-10-23-pgconf-eu-aio-in-PG-18-and-beyond/aio-in-PG-18-and-beyond.pdf>
 - https://www.youtube.com/watch?v=rw_8HaAQnY
- index prefetching [Peter Geoghegan, Tomas Vondra, pgconf.dev 2025]
 - <https://www.pgevents.ca/events/pgconfdev2026/schedule/session/659-update-on-index-prefetching/>
 - https://www.youtube.com/watch?v=7hbxw0W_9gQ
- io_uring in Oracle Database: A Hybrid Storage I/O Architecture at Production Scale
 - <https://arxiv.org/abs/2609.22781>

[feedback](#)

