15 DECEMBER 2025

# PostgreSQL 18 Asynchronous Disk I/O - Deep Dive Into Implementation

PostgreSQL 17 introduced streaming I/O – grouping multiple page reads into a single system call and using smarter posix_fadvise() hints. That alone gave up to ~30% faster sequential scans in some workloads, but it was still strictly synchronous: each backend process would issue a read and then sit there waiting for the kernel to return data before proceeding. Before PG17, PostgreSQL typically read one 8kB page at a time.

PostgreSQL 18 takes the next logical step: a full asynchronous I/O (AIO) subsystem that can keep multiple reads in flight while backends keep doing useful work. Reads become overlapped instead of only serialized. The AIO subsystem is deliberately targeted at operations that know their future block numbers ahead of time and can issue multiple reads in advance:

- Heap sequential scans, like plain SELECT and COPY operations that stream lots of data
- VACUUM on big tables and indexes
- ANALYZE sampling
- Bitmap heap scans

Autovacuum benefits from this change too, since its workers share the same VACUUM/ANALYZE code paths. Other operations still remain synchronous for now:

- B-tree index scans / index-only scans
- Recovery & replication
- All write operations INSERT, UPDATE, DELETE, WAL writes
- Small OLTP lookups that touch a single heap page

Future work is expected to widen coverage, especially index-only scans and some write-path optimizations.

## Significant improvements for cloud volumes

Community benchmarks show that PostgreSQL 18 AIO significantly improves cold cache data reads in cloud setups with network-attached storage where latency is high. AWS documentation states that average latency of Block Express volumes is “under 500 microseconds for 16 KiB I/O size”, when latency of General Purpose volumes can exceed 800 microseconds. Some articles suggest that under high load each physical block read from disk can cost around 1ms, while page processing in PostgreSQL is much cheaper. By combining many pages into one read, all these pages together now cost around 1ms. And by performing multiple reading requests at the same time in parallel, we effectively pay that 1ms latency just once per the whole batch.

## Asynchronous I/O methods

The new subsystem can run in one of three modes, configured via ***io_method*** parameter with possible values "*worker*" (default), "*io_uring*", and "*sync*". We will cover how each works and then show how to monitor asynchronous I/O in our environment.

## io_method = sync

This mode effectively turns AIO off. Reads are executed through the same AIO API but synchronously, using regular preadv or pwritev methods on the backend process that issued the I/O. This method does not use any extra shared memory and is intended mainly for regression testing or if we suspect AIO is misbehaving. It is also used internally as fall back to the synchronous I/O for operations which cannot use asynchronous I/O. PostgreSQL core functions issue an error, if some extension would try to force asynchronous I/O through AIO API when global io_method is set to "sync". Available benchmarks show that this PostgreSQL 18 mode performs similarly to PostgreSQL 17's streaming I/O.

## io_method = io_uring (Linux only)

On modern Linux (kernel version 5.1 or higher), PostgreSQL can talk directly to the kernel's io_uring interface. Usage requires PostgreSQL to be built with liburing support – we can check it inside PostgreSQL using select from pg_config() function:

```
SELECT pg_config FROM pg_config() where pg_config::text ilike '%liburing%';
```

PostgreSQL asynchronous I/O operations (both io_uring and worker) use shared memory structures for issuing the requests and receiving info about its completion or failure. This way PostgreSQL AIO code can manage batching and concurrency without direct dependency on specific AIO method. PostgreSQL code maintains one separate io_uring instance for each backend, including auxiliary processes. But rings are created in the postmaster, so they can use shared memory and there is no contention or blocking between backends.
Processing scenario is very simple:

1. **Backends write requests via API into a submission ring in shared memory**
2. **The kernel performs I/O asynchronously and writes results into a completion ring**
3. **Completion ring content is consumed by the backend with fewer context switches**

Execution still happens in the same process, like with the "*sync*" method, but it uses kernel worker threads for parallel processing. This typically shines on very fast NVMe SSDs.

However, io_uring Linux feature also has had a rough security history. It bypasses traditional syscall audit paths and therefore has been involved in a large share of Linux kernel exploits. Google reported that 60% of Linux kernel vulnerabilities in 2022 involved io_uring and some security tools were unable to uncover these types of attacks. Therefore some container environments disable io_uring entirely.

## io_method = worker

This is the cross-platform, "safe" implementation and the default in PostgreSQL 18. Mechanism is very similar to existing parallel query processing. The main difference is that background I/O workers are long-lived independent processes created at server start, not short-lived processes spawned per query.

Typical flow:

1. **At server start, the postmaster creates a pool of I/O worker processes. Number is controlled by *io_workers* parameter with a default of *3*. However, benchmarks suggest this number should be higher on many-core machines, typically between ¼ and ½ of available CPU threads. Best value depends on workload and storage latency.**
2. **Backends submit read requests into a shared memory submission queue. This submission queue is generally a ring buffer that multiple backends can write into concurrently. It contains only metadata about the request –**

handle indices, not full request record. There is only one submission queue for the entire cluster, not per database or per backend. The actual details of the request are stored in separate memory structure.

3. Request is checked if it must be executed synchronously or can be handled asynchronously. Synchronous execution can also be chosen if the submission queue is full. This avoids problems with shared memory usage under extreme load. In case of synchronous execution, code uses path for “sync” method described above.
4. Request submission in shared memory wakes up one I/O worker, which pops request and executes traditional blocking read() / pread() calls. If queue is still not empty, woken worker can wake up 2 additional workers to process it in parallel. Note in code mentions that this can be in the future extended to configurable N workers. This limit helps to avoid so called “thundering herd problem”, when single submitter would wake up too many workers causing havoc and locks for other backends.
5. One limitation for asynchronous I/O is the fact, that workers cannot simply reuse file descriptors opened by backends, they must reopen files in their own context. If this is not possible for some types of operations, synchronous I/O path is used for that specific request.
6. When workers finish a request without an error, they write data blocks into share buffers, put result into a completion queue and signal the backend.
7. From the perspective of the backend, I/O becomes “asynchronous”, because the “waiting” happens in worker processes, not in the query process itself.

Advantages of this approach:

- Works on all supported OSes
- Simple error handling: if a worker crashes, requests are marked as failed, worker exits and a new worker is spawned by postmaster
- Avoids the security concerns around Linux io_uring interface
- The downside is extra context switches and possible shared-memory queue contention, but for many workloads the ability to overlap reads easily pays for that
- This method improves performance even in the case when all blocks are just copied from local Linux memory cache, because it is now done in parallel

## Tuning the New I/O Parameters

PostgreSQL 18 adds or updates several parameters related to disk I/O. We already covered ***io_method*** and ***io_workers***; let’s look at the others. Another new parameters are ***io_combine_limit*** and ***io_max_combine_limit***. They control how many data pages PostgreSQL groups into a single AIO request. Larger requests typically yield better throughput, but can also increase latency and memory usage. Values without units are interpreted in 8kB data blocks. With units (kB, MB), they directly represent size – however, should be multiples of 8kB.

Parameter ***io_max_combine_limit*** is a hard server-start cap, ***io_combine_limit*** is the user-tunable value that can be changed at runtime but cannot exceed the max. Default values of both is 128kB (16 data pages). But documentation recommends setting up to 1MB on Unix (128 data pages) and 128kB on Windows (16 data pages – due to limitations in internal Widows buffers). We can experiment with higher values, but based on HW and OS limits AIO benefits plateau after some chunk size; pushing this too high doesn’t help and can even increase latency.

PostgreSQL 18 introduces also ***io_max_concurrency*** setting, which controls max number of IOs that one process can execute simultaneously. Default setting *-1* means value will be selected automatically based on other settings, but it cannot exceed 64.

Other related parameter is ***effective_io_concurrency*** – number of concurrent I/O operations that can be executed simultaneously on storage. Range of values is from 1 to 1000, value 0 disables asynchronous I/O requests. Default value is now 16, some community articles suggest to go up to 200 on modern SSDs. Best setting depends on specific hardware and OS, however, some articles also warn that too high value may significantly increase I/O latency for all queries.

# How to Monitor Asynchronous I/O

## pg_stat_activity

For *io_method = worker* background I/O workers are visible in ***pg_stat_activity*** as *backend_type = 'io worker'*. They show *wait_event_type* / *wait_event* values Activity / IoWorkerMain when they are idle, or typically IO / DataFileRead when they're busy doing work.

```
SELECT pid, backend_start, wait_event_type, wait_event, backend_type
FROM pg_stat_activity
WHERE backend_type = 'io worker';


  pid |        backend_start          | wait_event_type |  wait_event  | backend_type
------+-------------------------------+-----------------+--------------+--------------
   34 | 2025-12-09 11:44:23.852461+00 | Activity        | IoWorkerMain | io worker
   35 | 2025-12-09 11:44:23.852832+00 | Activity        | IoWorkerMain | io worker
   36 | 2025-12-09 11:44:23.853119+00 | IO              | DataFileRead | io worker
   37 | 2025-12-09 11:44:23.8534+00   | IO              | DataFileRead | io worker
```

We can combine ***pg_stat_io*** with ***pg_stat_activity*** to see which backends are issuing AIO requests, which queries they're running and what their current AIO state is:

```
SELECT a.pid, a.usename, a.application_name, a.backend_type, a.state, a.query,
ai.operation, ai.state AS aio_state, ai.length AS aio_bytes, ai.target_desc
FROM pg_aios ai
JOIN pg_stat_activity a ON a.pid = ai.pid
ORDER BY a.backend_type, a.pid, ai.io_id;


-[ RECORD 1 ]----+-----------------------------------------------------------------
             pid | 58
        usename | postgres
application_name | psql
    backend_type | client backend
           state | active
           query | explain analyze SELECT ........
       operation | readv
       aio_state | SUBMITTED
       aio_bytes | 704512
     target_desc | blocks 539820..539905 in file "pg_tblspc/16647/PG_18_202506291/5/16716"
-[ RECORD 2 ]----+-----------------------------------------------------------------
             pid | 159
         usename | postgres
application_name | psql
    backend_type | parallel worker
           state | active
           query | explain analyze SELECT ........
       operation | readv
       aio_state | SUBMITTED
       aio_bytes | 704512
     target_desc | blocks 536326..536411 in file "pg_tblspc/16647/PG_18_202506291/5/16716"
```

## pg_aios: Current AIO handles

PostgreSQL 18 introduces several new observability features to help us to monitor asynchronous I/O in action. New system view ***pg_aios*** is listing currently in-use asynchronous I/O handles – essentially "I/O requests that are being prepared, executed, or finishing".
Key columns are for each handle:

- pid: backend issuing the I/O

- io_id, io_generation: identify a handle across reuse
- state: HANDED_OUT, DEFINED, STAGED, SUBMITTED, COMPLETED_IO, COMPLETED_SHARED, COMPLETED_LOCAL
- operation: invalid, readv (vectored read) or writev (vectored write)
- off, length: offset and size of I/O operation
- target, target_desc: what we're reading/writing (typically relations)
- result: UNKNOWN, OK, PARTIAL, WARNING, ERROR

We can generate some simple stats of all I/Os currently in flight, grouped by state and result:

```
-- Summary of current AIO handles by state and result
SELECT state, result, count(*) AS cnt, pg_size_pretty(sum(length)) AS total_size
FROM pg_aios GROUP BY state, result ORDER BY state, result;

       state      |  result | cnt | total_size
------------------+---------+-----+------------
 COMPLETED_SHARED | OK      |   1 | 688 kB
 SUBMITTED        | UNKNOWN |   6 | 728 kB

-- In-flight async I/O handles
SELECT COUNT(*) AS aio_handles, SUM(length) AS aio_bytes FROM pg_aios;

 aio_handles | aio_bytes
-------------+-----------
           7 |     57344

-- Sessions currently waiting on I/O
SELECT COUNT(*) AS sessions_waiting_on_io FROM pg_stat_activity WHERE wait_event_type = 'IO';

 sessions_waiting_on_io
------------------------
                      9
```

Or we can use it to see details about current AIO requests:

```
SELECT pid, state, operation, pg_size_pretty(length) AS io_size, target_desc, result
FROM pg_aios ORDER BY pid, io_id;

 pid |   state   | operation |   io_size  |                                target_desc
| result
-----+-----------+-----------+------------+-------------------------------------------------------------------------
-+---------
  51 | SUBMITTED | readv     | 688 kB     | blocks 670470..670555 in file "pg_tblspc/16647/PG_18_202506291/5/16716"
| UNKNOWN
  63 | SUBMITTED | readv     | 8192 bytes | block 1347556 in file "pg_tblspc/16647/PG_18_202506291/5/16719"
| UNKNOWN
  65 | SUBMITTED | readv     | 688 kB     | blocks 671236..671321 in file "pg_tblspc/16647/PG_18_202506291/5/16716"
| UNKNOWN
  66 | SUBMITTED | readv     | 8192 bytes | block 1344674 in file "pg_tblspc/16647/PG_18_202506291/5/16719"
| UNKNOWN
  67 | SUBMITTED | readv     | 8192 bytes | block 1337819 in file "pg_tblspc/16647/PG_18_202506291/5/16719"
| UNKNOWN
  68 | SUBMITTED | readv     | 688 kB     | blocks 672002..672087 in file "pg_tblspc/16647/PG_18_202506291/5/16716"
| UNKNOWN
  69 | SUBMITTED | readv     | 688 kB     | blocks 673964..674049 in file "pg_tblspc/16647/PG_18_202506291/5/16716"
| UNKNOWN
```

## pg_stat_io: Cumulative I/O stats

Catalog view ***pg_stat_io*** was introduced in PostgreSQL 16, but PostgreSQL 18 extends it with byte counters (read_bytes, write_bytes, extend_bytes) and better coverage of WAL and bulk I/O contexts. However, timing columns are only populated if we enable the timing parameters – ***track_io_timing*** – default is off.

A handy per-client view of relation I/O:

```
SELECT backend_type, context, sum(reads) AS reads,
pg_size_pretty(sum(read_bytes)) AS read_bytes,
round(sum(read_time)::numeric, 2) AS read_ms, sum(writes) AS writes,
pg_size_pretty(sum(write_bytes)) AS write_bytes,
round(sum(write_time)::numeric, 2) AS write_ms, sum(extends) AS extends,
pg_size_pretty(sum(extend_bytes)) AS extend_bytes
FROM pg_stat_io
WHERE object = 'relation' AND backend_type IN ('client backend')
GROUP BY backend_type, context
ORDER BY backend_type, context;

   backend_type |  context  |  reads  | read_bytes |  read_ms  | writes | write_bytes | write_ms | extends |
extend_bytes
----------------+-----------+---------+------------+-----------+--------+-------------+----------+---------+-------
-------
 client backend | bulkread  |   13833 | 9062 MB    | 124773.28 |      0 | 0 bytes     |     0.00 |         |
 client backend | bulkwrite |       0 | 0 bytes    |      0.00 |      0 | 0 bytes     |     0.00 |       0 | 0
bytes
 client backend | init      |       0 | 0 bytes    |      0.00 |      0 | 0 bytes     |     0.00 |       0 | 0
bytes
 client backend | normal    | 2265214 | 17 GB      | 553940.57 |      0 | 0 bytes     |     0.00 |       0 | 0
bytes
 client backend | vacuum    |       0 | 0 bytes    |      0.00 |      0 | 0 bytes     |     0.00 |       0 | 0
bytes


-- Top tables by heap blocks read and cache hit ratio
SELECT relid::regclass AS table_name, heap_blks_read, heap_blks_hit,
ROUND( CASE WHEN heap_blks_read + heap_blks_hit = 0 THEN 0
ELSE heap_blks_hit::numeric / (heap_blks_read + heap_blks_hit) * 100 END, 2) AS cache_hit_pct
FROM pg_statio_user_tables
ORDER BY heap_blks_read DESC LIMIT 20;

      table_name      | heap_blks_read | heap_blks_hit | cache_hit_pct
----------------------+----------------+---------------+---------------
 table1               |       18551282 |       3676632 |         16.54
 table2               |        1513673 |     102222970 |         98.54
 table3               |          19713 |       1034435 |         98.13
...


-- Top indexes by index blocks read and cache hit ratio
SELECT relid::regclass AS table_name, indexrelid::regclass AS index_name,
idx_blks_read, idx_blks_hit
FROM pg_statio_user_indexes
ORDER BY idx_blks_read DESC LIMIT 20;

 table_name |    index_name   | idx_blks_read | idx_blks_hit
------------+-----------------+---------------+--------------
 table1     | idx_table1_date |        209289 |          141
 table2     | table2_pkey     |         37221 |      1223747
 table3     | table3_pkey     |          9825 |      3143947
...
```

For establishing a baseline before/after a test run, we can reset stats (as superuser):

```
SELECT pg_stat_reset_shared('io');
```

Then run our workload and query pg_stat_io again to see how many bytes were read/written and how much time was spent waiting on I/O.

## Conclusion

PostgreSQL 18's new asynchronous I/O subsystem is a significant step forward in improving I/O performance for large scans and maintenance operations. By overlapping reads and allowing multiple requests to be in flight, it can better utilize modern storage systems and reduce query times for data-intensive workloads. With the new observability features in pg_aios and pg_stat_io, DBAs and developers can monitor AIO activity and tune parameters to optimize performance for their specific workloads. As PostgreSQL continues to evolve, we can expect further enhancements to the AIO subsystem and broader coverage of operations that can benefit from asynchronous I/O.
PostgreSQL is a registered trademark oftThe PostgreSQL Community Association of Canada.

JM

ABOUT THE AUTHOR

Josef Machytka

View posts