Asynchronous IO for PostgreSQL

Andres Freund
PostgreSQL Developer & Committer
Microsoft

andres@anarazol.de
andres.freund@microsoft.com
@AndresFreundTec
Why AIO?

Buffered IO is a major limitation.
Why AIO?

tpch_100[1575595][1]=# EXPLAIN (ANALYZE, BUFFERS) SELECT sum(l_quantity) FROM lineitem ;

<table>
<thead>
<tr>
<th>QUERY PLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finalize Aggregate (cost=11439442.89..11439442.90 rows=1 width=8) (actual time=46264.524..46264.524 rows=1 loops=1)</td>
</tr>
<tr>
<td>Buffers: shared hit=2503 read=10602553</td>
</tr>
<tr>
<td>I/O Timings: read=294514.747</td>
</tr>
<tr>
<td>Workers Planned: 9</td>
</tr>
<tr>
<td>Workers Launched: 8</td>
</tr>
<tr>
<td>Buffers: shared hit=2503 read=10602553</td>
</tr>
<tr>
<td>I/O Timings: read=294514.747</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Gather (cost=11439441.95..11439442.86 rows=9 width=8) (actual time=46250.927..46278.690 rows=9 loops=1)</td>
</tr>
<tr>
<td>Buffers: shared hit=2503 read=10602553</td>
</tr>
<tr>
<td>I/O Timings: read=294514.747</td>
</tr>
<tr>
<td>Workers Planned: 9</td>
</tr>
<tr>
<td>Workers Launched: 8</td>
</tr>
<tr>
<td>Buffers: shared hit=2503 read=10602553</td>
</tr>
<tr>
<td>I/O Timings: read=294514.747</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Partial Aggregate (cost=11439441.95..11439441.96 rows=1 width=8) (actual time=46201.154..46201.154 rows=1 loops=9)</td>
</tr>
<tr>
<td>Buffers: shared hit=2503 read=10602553</td>
</tr>
<tr>
<td>I/O Timings: read=294514.747</td>
</tr>
<tr>
<td>Workers Planned: 9</td>
</tr>
<tr>
<td>Workers Launched: 8</td>
</tr>
<tr>
<td>Buffers: shared hit=2503 read=10602553</td>
</tr>
<tr>
<td>I/O Timings: read=294514.747</td>
</tr>
</tbody>
</table>

Planning Time: 0.139 ms
JIT:
Functions: 29
Options: Inlining true, Optimization true, Expressions true, Deforming true
Timing: Generation 5.209 ms, Inlining 559.852 ms, Optimization 266.074 ms, Emission 163.010 ms, Total 985.145 ms
Execution Time: 46279.595 ms

(20 rows)
Why AIO?

$ perf stat -a -e cycles:u,cycles:k,ref-cycles:u,ref-cycles:k sleep 5
Performance counter stats for 'system wide':

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Command</th>
<th>Command</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.95%</td>
<td>postgres</td>
<td>elf</td>
<td>[k] copy_user_enhanced_fast_string</td>
</tr>
<tr>
<td>29.65%</td>
<td>postgres</td>
<td>jittd-1216148-2.so</td>
<td>[.] evalexpr_0_1</td>
</tr>
<tr>
<td>7.43%</td>
<td>postgres</td>
<td>postgres</td>
<td>[.] heapgetpagemode</td>
</tr>
<tr>
<td>4.46%</td>
<td>postgres</td>
<td>postgres</td>
<td>[.] heap getnextslot</td>
</tr>
<tr>
<td>4.83%</td>
<td>postgres</td>
<td>postgres</td>
<td>[.] heapgetpage</td>
</tr>
<tr>
<td>3.98%</td>
<td>postgres</td>
<td>postgres</td>
<td>[.] ExecStoreBufferHeapTuple</td>
</tr>
<tr>
<td>3.88%</td>
<td>postgres</td>
<td>postgres</td>
<td>[.] SeqNext</td>
</tr>
<tr>
<td>3.46%</td>
<td>postgres</td>
<td>postgres</td>
<td>[.] ExecAgg</td>
</tr>
<tr>
<td>3.45%</td>
<td>postgres</td>
<td>postgres</td>
<td>[.] ExecScan</td>
</tr>
<tr>
<td>3.13%</td>
<td>postgres</td>
<td>elf</td>
<td>[k] get_page_from_freelist</td>
</tr>
<tr>
<td>3.08%</td>
<td>postgres</td>
<td>postgres</td>
<td>[.] hash_search with hash value</td>
</tr>
<tr>
<td>3.02%</td>
<td>postgres</td>
<td>postgres</td>
<td>[.] MemoryContextReset</td>
</tr>
<tr>
<td>2.37%</td>
<td>postgres</td>
<td>postgres</td>
<td>[.] fetch_input_tuple</td>
</tr>
<tr>
<td>1.17%</td>
<td>postgres</td>
<td>postgres</td>
<td>[.] LWLockRelease</td>
</tr>
<tr>
<td>1.15%</td>
<td>postgres</td>
<td>elf</td>
<td>[k] xas_load</td>
</tr>
<tr>
<td>1.02%</td>
<td>postgres</td>
<td>postgres</td>
<td>[.] LWLockAcquire</td>
</tr>
<tr>
<td>1.01%</td>
<td>postgres</td>
<td>postgres</td>
<td>[.] ReadBuffer_common</td>
</tr>
<tr>
<td>1.01%</td>
<td>postgres</td>
<td>elf</td>
<td>[k] __pagenvec_lru_add_fn</td>
</tr>
<tr>
<td>0.78%</td>
<td>postgres</td>
<td>elf</td>
<td>[k] generic_file_read_iter</td>
</tr>
<tr>
<td>0.69%</td>
<td>postgres</td>
<td>postgres</td>
<td>[.] CheckForSerializableConflictOutNeeded</td>
</tr>
<tr>
<td>0.58%</td>
<td>postgres</td>
<td>[vds0]</td>
<td>[.] _vds0_clock_gettime</td>
</tr>
<tr>
<td>0.55%</td>
<td>postgres</td>
<td>elf</td>
<td>[k] _add_to_page_cache_locked</td>
</tr>
<tr>
<td>0.48%</td>
<td>postgres</td>
<td>elf</td>
<td>[k] Find get entry</td>
</tr>
<tr>
<td>0.43%</td>
<td>postgres</td>
<td>elf</td>
<td>[k] entry_SYSCALL_64</td>
</tr>
</tbody>
</table>

Tip: To record every process run by a user: perf record -u <user>

$ 52,886,623,568 cycles:u (49.99%)
$ 50,676,736,054 cycles:k (74.99%)
$ 47,563,244,024 ref-cycles:u (75.00%)
$ 46,187,922,930 ref-cycles:k (25.00%)
5.002662309 seconds time elapsed
Why AIO?

<table>
<thead>
<tr>
<th>Type</th>
<th>Workers</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>from disk</td>
<td>0</td>
<td>59.94s</td>
</tr>
<tr>
<td>from disk</td>
<td>3</td>
<td>48.56s</td>
</tr>
<tr>
<td>from disk</td>
<td>9</td>
<td>37.84s</td>
</tr>
<tr>
<td>from os cache</td>
<td>0</td>
<td>47.28s</td>
</tr>
<tr>
<td>from os cache</td>
<td>9</td>
<td>8.13s</td>
</tr>
<tr>
<td>from PG cache</td>
<td>0</td>
<td>34 s</td>
</tr>
<tr>
<td>from PG cache</td>
<td>9</td>
<td>5.37s</td>
</tr>
</tbody>
</table>

- With no amount of concurrency the disk bandwith for streaming reads (~3.2GB/s) can be reached.
- Kernel pagecache is not much faster than doing the IO for single process
Buffered uncached read()
Direct IO read()
“Direct IO”

- Kernel <-> Userspace buffer transfer, without a separate pagecache
  - Often using DMA, i.e. not using CPU cycles
  - Very little buffering in kernel
- Userspace has much much more control / responsibility when using DIO
- No readahead, no buffered writes => read(), write() are synchronous
- Synchronous use unusably slow for most purposes
Why AIO?

• Throughput problems:
  - background writer quickly saturated (leaving algorithmic issues aside)
  - checkpoint can’t keep up
  - WAL write flushes too slow / latency too high

• CPU Overhead
  - memory copy for each IO (CPU time & cache effects)
  - pagecache management
  - overhead of filesystem + pagecache lookup for small IOs

• Lack of good control
  - Frequent large latency spikes due to kernel dirty writeback management
  - Kernel readahead fails often (segment boundary, concurrent accesses, low QD for too fast / too high latency
    drives)
  - Our “manual” readahead comes at high costs (double shared_buffers lookup, double OS pagecache lookup,
    unwanted blocking when queue depth is reached, …)

• ...
## Buffered vs Direct

<table>
<thead>
<tr>
<th>Query</th>
<th>Branch</th>
<th>Time (s)</th>
<th>Avg CPU %</th>
<th>Avg MB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>select pg_prewarm('lineitem', 'read');</code></td>
<td>master</td>
<td>34.6</td>
<td>~78</td>
<td>~2550</td>
</tr>
<tr>
<td><code>select pg_prewarm('lineitem', 'read_aio');</code></td>
<td>aio</td>
<td>27.0</td>
<td>~51</td>
<td>~3100</td>
</tr>
<tr>
<td><code>select pg_prewarm('lineitem', 'buffer');</code></td>
<td>master</td>
<td>56.6</td>
<td>~95</td>
<td>~1520</td>
</tr>
<tr>
<td><code>select pg_prewarm('lineitem', 'buffer_aio');</code></td>
<td>aio</td>
<td>29.3</td>
<td>~75</td>
<td>~2900</td>
</tr>
</tbody>
</table>
Why not yet?

- Linux AIO didn’t use to support buffered IO
- Not everyone can use DIO
- Synchronous DIO very slow (latency)
- It’s a large project / most people are sane
- Adds / Requires complexity
postgres[1583389][1]=# SHOW io_data_direct ;

<table>
<thead>
<tr>
<th>io_data_direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
</tr>
</tbody>
</table>

tpch_100[1583290][1]=# select pg_prewarm('lineitem', 'read');

<table>
<thead>
<tr>
<th>pg_prewarm</th>
</tr>
</thead>
<tbody>
<tr>
<td>10605056</td>
</tr>
</tbody>
</table>

(1 row)

Time: 160227.904 ms (02:40.228)

<table>
<thead>
<tr>
<th>Device</th>
<th>r/s</th>
<th>rMB/s</th>
<th>rrqm/s</th>
<th>%rrqm</th>
<th>r-await</th>
<th>rareq-sz</th>
<th>aqu-sz</th>
<th>%util</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvme1n1</td>
<td>71070.00</td>
<td>555.23</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>8.00</td>
<td>0.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>
io_uring

• New linux AIO interface, added in 5.1
• Generic, quite a few operations supported
  – open / close / readv / writev / fsync, statx, …
  – send/recv/accept/connect/…, including polling
• One single-reader / single writer ring for IO submission, one SPSC ring for completion
  – allows batched “syscalls”
• Operations that aren’t fully asynchronous are made asynchronous via kernel threads
io_uring basics

- **Application**
  - Submission Queue
  - Completion Queue
- **io_uring**
- **Buffered IO**
- **Buffered IO**
- **Network Subsystem**
- **...**

**Userspace**

**Kernel**
io_uring operations

/*
 * IO submission data structure (Submission Queue Entry)
 */

struct io_uring_sqe {
    __u8    opcode;         /* type of operation for this sqe */
    __u8    flags;          /* IO_SQE_ flags */
    __u16   ioprio;         /* ioprio for the request */
    __s32   fd;             /* file descriptor to do IO on */
    union {
        __u64   off;    /* offset into file */
        __u64   addr2;
    };
    union {
        __u64   addr;   /* pointer to buffer or iovecs */
        __u64   splice_off_in;
    };
    __u32   len;            /* buffer size or number of iovecs */
    union {
        __kernel_rwf_t  rw_flags;
        __u32           fsync_flags;
        __u16           poll_events;
        __u32           sync_range_flags;
        ...
    };
    __u64   user_data;      /* data to be passed back at completion */
};//

enum {
    IORING_OP_NOP,
    IORING_OP_READV,
    IORING_OP_WRITEV,
    IORING_OP_FSYNC,
    IORING_OP_READ_FIXED,
    IORING_OP_WRITE_FIXED,
    IORING_OP_POLL_ADD,
    IORING_OP_POLL_REMOVE,
    IORING_OP_SYNC_FILE_RANGE,
    IORING_OP_SENDMSG,
    IORING_OP_RECVMSG,
    IORING_OP_TIMEOUT,
    IORING_OP_TIMEOUT_REMOVE,
    IORING_OP_ACCEPT,
    IORING_OP_ASYNC_CANCEL,
    IORING_OP_CONNECT,
    ...
};

/* this goes last, obviously */
IORING_OP_LAST,
Constraints on AIO for PG

- Buffered IO needs to continue to be feasible
- Platform specific implementation details need to be abstracted
- Cross process AIO completions are needed:
  1) backend a: lock database object x exclusively
  2) backend b: submit read for block y
  3) backend b: submit read for block z
  4) backend a: try to access block y, IO_IN_PROGRESS causes wait
  5) backend b: try to lock database object x
Lower-Level AIO Interface

1) Acquire a shared "IO" handle
   
   \[
   \text{aio} = \text{pgaio\_io\_get}();
   \]

2) Optionally register callback to be called when IO completes
   
   \[
   \text{pgaio\_io\_on\_completion\_local(aio, prefetch\_more\_and\_other\_things)}
   \]

3) Stage some form of IO locally:
   
   \[
   \text{pgaio\_io\_start\_read\_buffer(aio, \ldots)}
   \]
      
   a) Go back to 1) many times if useful

4) Cause pending IO to be submitted
   
   a) By waiting for an individual IO:
      
      \[
      \text{pgaio\_io\_wait()}
      \]
   
   b) By explicitly issuing individual IO:
      
      \[
      \text{pgaio\_submit\_pending()}
      \]

5) \text{aio.c submits IO via io\_uring}
Higher Level AIO Interface

- **Streaming Read helper**
  - Tries to maintain N requests in flight, up to a certain distance from current point
  - Caller users `pg_streaming_read_get_next(pgsr);` to get the next block
  - Uses provided callback to inquire which IO is the next needed
    - heapam fetches sequentially
    - vacuum checks VM which is next
  - Uses `pgaio_io_on_completion_local()` callback to promptly issue new IOs

- **Streaming Write**
  - Controls the number of outstanding writes
  - allows to wait for all pending IOs (at end, or before a potentially blocking action)
Prototype Architecture

Shared Memory
- PgAioInPerBackend[]
- PgAioInProgress[]
- io_uring wal
- io_uring data #1
- io_uring data #2

Shared Buffers

Backend / Helper Process
- heapam.c
- vacuumlazy.c
  - Streaming Read Helper
  - checkpoint bgwriter
  - Streaming Write Helper
Prototype Results

- Helps most with very high throughput low latency drives and with high latency & high throughput
- analytics style queries:
  - often considerably faster (TPCH 100 has all faster, several > 2x)
  - highly parallel bulk reads scale poorly, known cause (one io_uring + locks)
  - seqscan ringbuffer + hot pruning can cause problems: Ring buffers don't use streaming write yet
- OLTP style reads/writes: A good bit better, to a bit slower
  - WAL AIO needs work
  - Better prefetching: See earlier talk by Thomas Munro
- VACUUM:
  - Much faster heap scan (~2x on low lat, >5x on high lat high throughput)
  - DIO noticeably slower for e.g. btree index scans: readahead helper not yet used, but trivial
  - Sometimes slower when creating a lot of dirty pages:
- Checkpointer: >2x
- Bgwriter: >3x
Next Big Things

- Use AIO helpers in more places
  - index vacuums
  - non-bufmgr page replacement
  - better use in bitmap heap scans
  - COPY & VACUUM streaming writes

- Scalability improvements (actually use more than one io_uring)

- Efficient AIO use in WAL

- Evaluate if process based fallback is feasible?
Resources

- **git tree**
  - [https://github.com/anarazel/postgres/tree/aio](https://github.com/anarazel/postgres/tree/aio)
  - [https://git.postgresql.org/gitweb/?p=users/andresfreund/postgres.git;a=shortlog;h=refs/heads/aio](https://git.postgresql.org/gitweb/?p=users/andresfreund/postgres.git;a=shortlog;h=refs/heads/aio)

- **Earlier talks related to AIO in PG**
  - [https://anarazel.de/talks/2020-01-31-fosdem-aio/aio.pdf](https://anarazel.de/talks/2020-01-31-fosdem-aio/aio.pdf)

- **io_uring**
  - “design” document: [https://kernel.dk/io_uring.pdf](https://kernel.dk/io_uring.pdf)
  - LWN articles:
    - [https://lwn.net/Articles/776703/](https://lwn.net/Articles/776703/)
    - [https://lwn.net/Articles/810414/](https://lwn.net/Articles/810414/)
  - man pages:
    - [https://manpages.debian.org/unstable/liburing-dev/io_uring_setup.2.en.html](https://manpages.debian.org/unstable/liburing-dev/io_uring_setup.2.en.html)
    - [https://manpages.debian.org/unstable/liburing-dev/io_uring_enter.2.en.html](https://manpages.debian.org/unstable/liburing-dev/io_uring_enter.2.en.html)