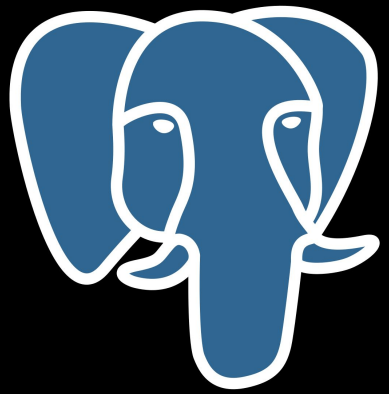


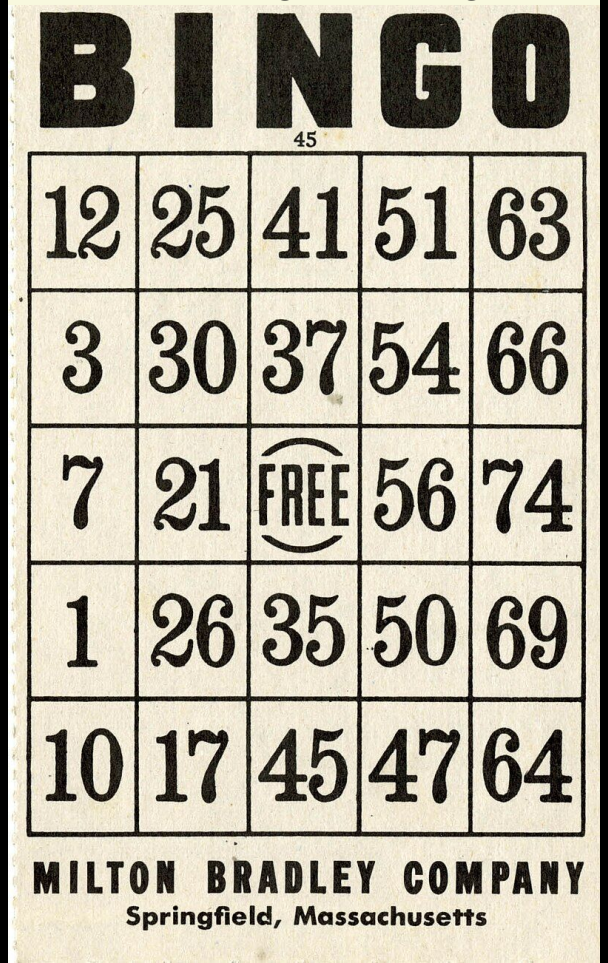
Streaming I/O

New abstractions for efficient file I/O

PGConf.EU 2024 | Athens
Thomas Munro & Nazir Bilal Yavuz
Open source database hackers working at Microsoft

Part I: Review of OS facilities

Database
I/O Programming



1. direct I/O vs buffered I/O
2. vectored I/O (also called scatter/gather)
3. asynchronous I/O vs synchronous I/O

Unix line of systems

MULTICS (’65) read, write, worksync, iowait

UNIX (’69) read, write

UNIX deliberately simplified: only synchronous buffered I/O

BSD, IRIX, … (’80s-’90s)

p… = with position
…v = vectored
p…v = both

O_DIRECT

POSIX (’93) aio_read, aio_write, …

Linux (’03?) libaio + kernel support

Linux (’19) io_uring

Contemporary systems

IBM S/360 (’65)

DEC RX11 (’71)

VMS (’77)

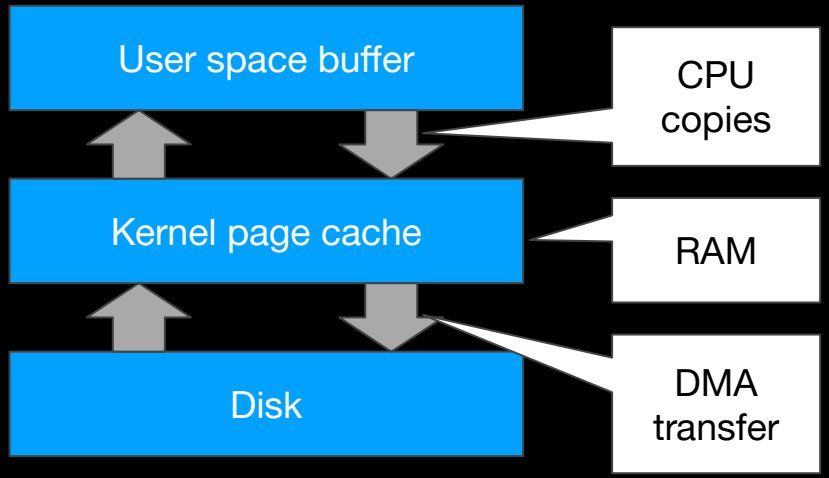
NT (’93)

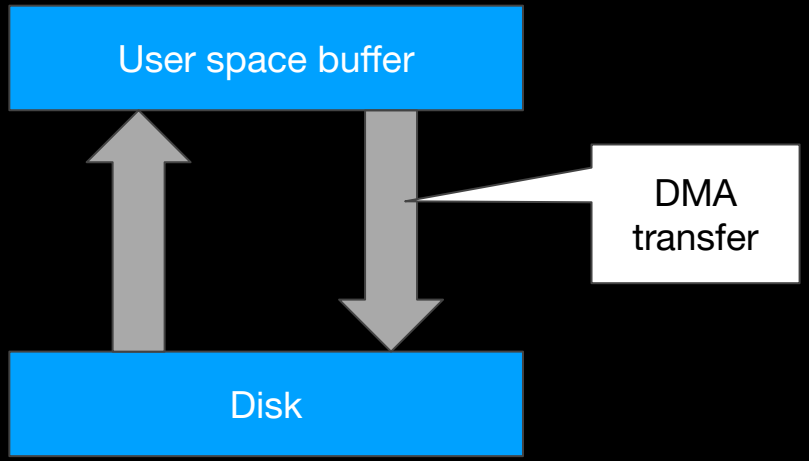
All had/have various forms of asynchronous I/O interface

Direct I/O


```
fd = open("path", O_RDWR);
read(fd, …)     write(fd, …)
```

```
fd = open("path", O_RDWR | O_DIRECT);
read(fd, …)     write(fd, …)
```



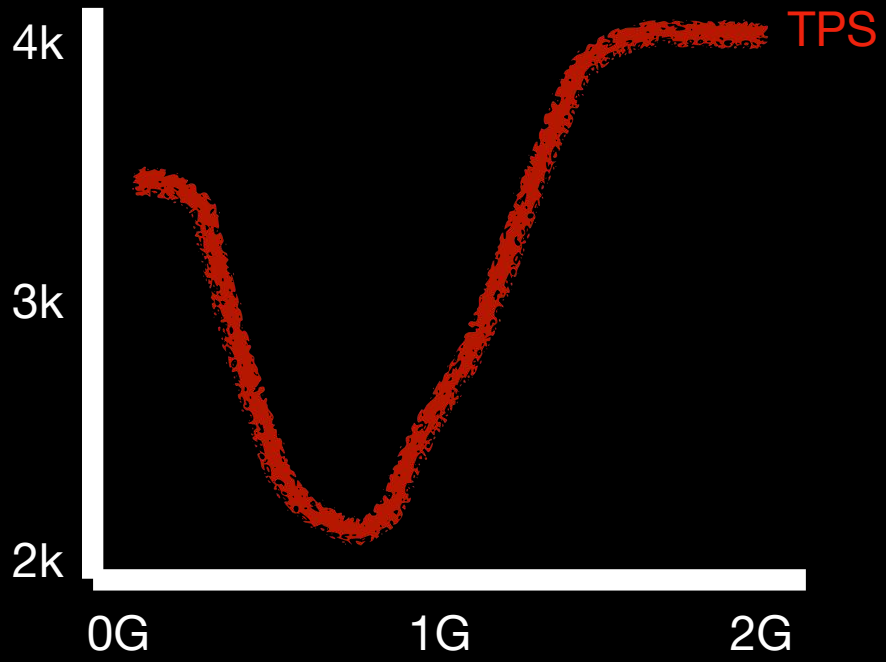


Direct I/O is an optimisation (CPU, RAM) *and* a pessimisation (when synchronous)!

Who wants direct I/O?

Systems that manage their own buffer pool (basically, databases*)

- Our user space buffer *is* a cache already, similar to kernel page cache!
- I/O buffering wastes your RAM and your CPU, throughput is reduced
- But… to skip the page cache effectively, we also need our own I/O combining, concurrency, read-ahead, write-behind, and to tune the buffer pool size more carefully



`debug_io_direct (string)`

Ask the kernel to minimize caching effects for relation data and WAL files using `O_DIRECT` (most Unix-like systems), `F_NOCACHE` (macOS) or `FILE_FLAG_NO_BUFFERING` (Windows).

May be set to an empty string (the default) to disable use of direct I/O, or a comma-separated list of operations that should use direct I/O. The valid options are `data` for main data files, `wal` for WAL files, and `wal_init` for WAL files when being initially allocated.

Some operating systems and file systems do not support direct I/O, so non-default settings may be rejected at startup or cause errors.

Currently this feature reduces performance, and is intended for developer testing only.

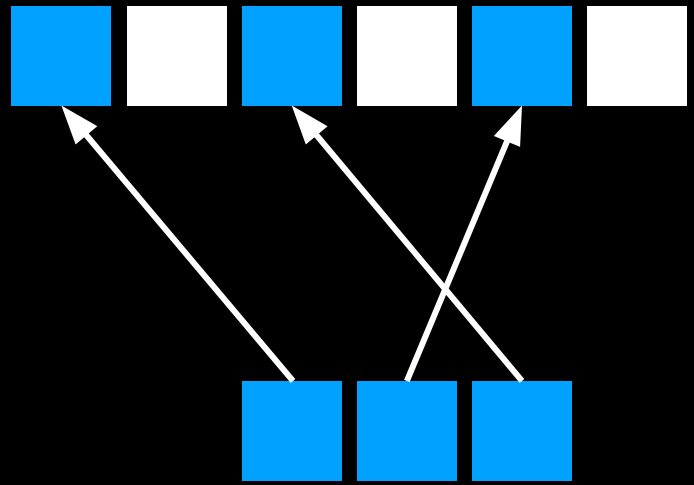
Vectored I/O… who needs it?

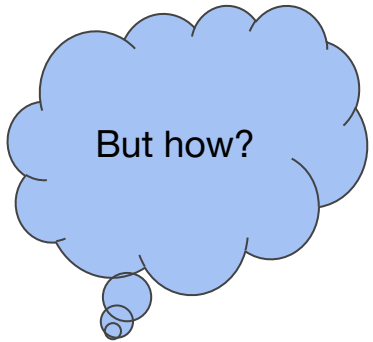
Systems that manage their own buffer pool (basically, databases)

```
struct iovec
{
    void        *iov_base;
    size_t       iov_len;
};
```

```
ssize_t pread (int filedes, void *buf, size_t nbytes,       off_t offset)
ssize_t preadv(int filedes, struct iovec *iov, int iovcnt, off_t offset)
```

- We want to read large contiguous chunks of a file into memory in one operation
- The buffer replacement algorithm doesn't try to find contiguous memory blocks (and shouldn't!)
- Kernel helps only with buffered I/O





`io_combine_limit (integer)`

Controls the largest I/O size in operations that combine I/O. The default is 128kB.

Asynchronous I/O: who needs it?

People using direct I/O! (and others…)

- While executing a query, we don't want our thread to "go to sleep" waiting for an I/O operation
- Simple portable implementation is to have I/O worker threads/processes running `preadv`/`pwritev` system call
- Modern (and ancient) OSes offer ways to skip the scheduling and IPC overheads of using a extra threads/processes
- Infrastructure not present in PostgreSQL yet as of v17; patches exist, testing and review welcome

What architectural changes do we need to use all of these features effectively?

Part II: Read Streams

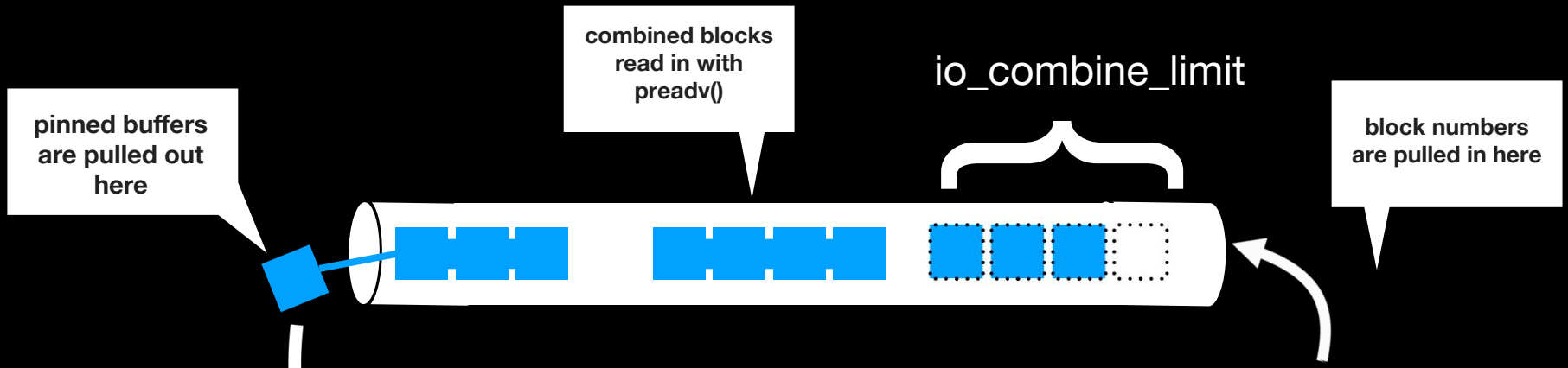
“Reading” blocks of relation data

A very common operation

- PostgreSQL works in terms of 8KB blocks, traditionally calling `ReadBuffer(relation identifier, block_number)` to access each one
- If the buffer is already in the buffer pool, it is pinned
- If the buffer is not already in the buffer pool, it must be loaded from disk, possibly after evicting something else to make space
- In order to build larger I/Os and start the physical I/O asynchronously, we need to find all the places that do that, and somehow convince them to participate in a new prediction and grouping system

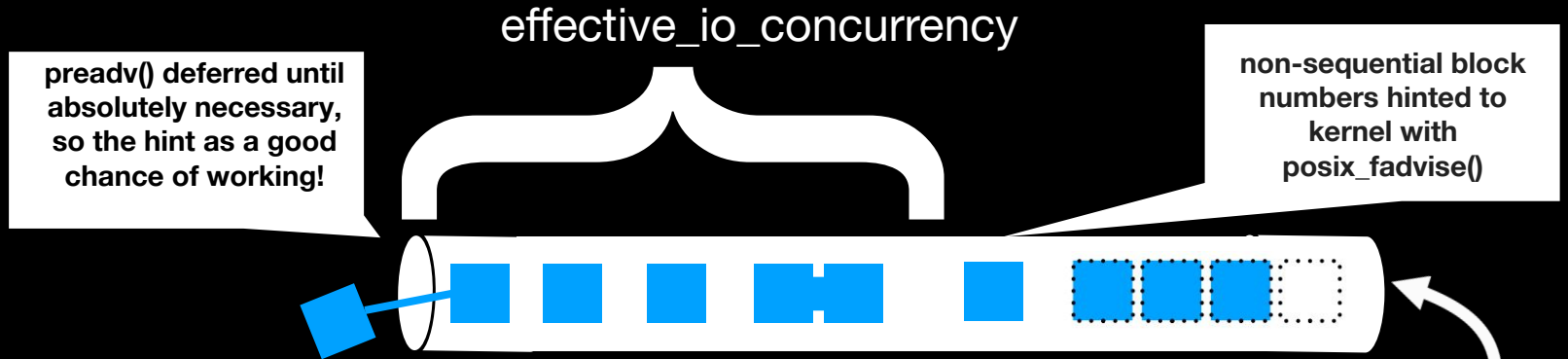
Ad hoc grouping and read-ahead at every call site

Re-usable stream mechanism




```
static BlockNumber my_blocknum_callback(void *private_data);

stream = read_stream_begin_relation(…,
                                    my_blocknum_callback,
                                    &my_callback_state, …);

for (i = 0; i < nblocks; ++i)
{
    buf = read_stream_next(stream);
    ReleaseBuffer(buf);
}
read_stream_end(stream);
```

```
for (i = 0; i < nblocks; ++i)
{
    buf = ReadBuffer(…, i);
    ReleaseBuffer(buf);
}
```

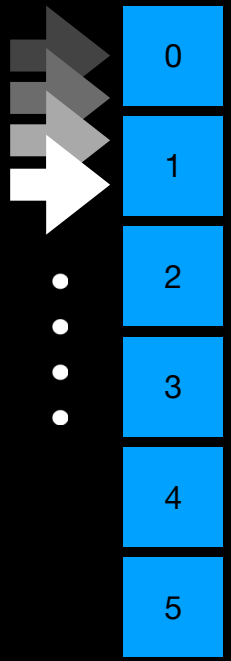


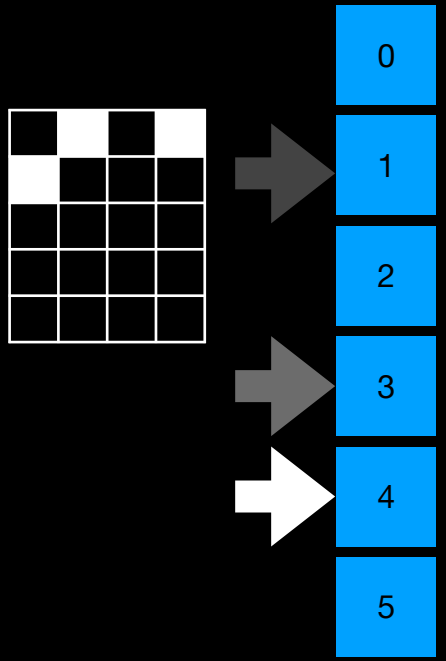

```
static BlockNumber my_blocknum_callback(void *private_data);

stream = read_stream_begin_relation(…,

my_blocknum_callback,

&my_callback_state, …);
for (i = 0; i < nblocks; ++i)
{
    buf = read_stream_next(stream);
    ReleaseBuffer(buf);
}
read_stream_end(stream);
```

By issuing POSIX_FADV_WILLNEED as soon as possible and preadv() as late as possible, we get a sort of poor man's asynchronous I/O.

Prediction is difficult, especially about the future

- Danish proverb about look-ahead callback functions





Arithmetic-driven:
seq scan (v17)
ANALYZE sampling (v17)

Data-driven:
bitmap heapscan (WIP)
recovery (WIP)

Callback of ANALYZE

```
static BlockNumber
block_sampling_read_stream_next(ReadStream *stream,
                                void *callback_private_data,
                                void *per_buffer_data)
{
    BlockSamplerData *bs = callback_private_data;

    return BlockSampler_HasMore(bs) ? BlockSampler_Next(bs)  : InvalidBlockNumber;
}
```

Knuth's sampling algorithm is used to select block numbers to analyze. Block numbers are increasing not always consecutive.

Callback of bitmap heap scan

```
static BlockNumber
heap_bitmap_scan_stream_read_next(ReadStream *stream,
                                  void *callback_private_data,
                                  void *per_buffer_data)
{
    TBMIterateResult *tbmres = per_buffer_data;
    BitmapHeapScanDesc *bscan = callback_private_data;
    HeapScanDesc *hscan = &bscan->rs_heap_base;

    for (;;)
    {
        CHECK_FOR_INTERRUPTS();

        tbm_iterate(&hscan->rs_base.tbmiterator, tbmres);

        /* no more entries in the bitmap */
        if (!BlockNumberIsValid(tbmres->blockno))
            return InvalidBlockNumber;

        if (!IsolationIsSerializable() && tbmres->blockno >= hscan->rs_nblocks)
            continue;

        if (!(hscan->rs_base.rs_flags & SO_NEED_TUPLES) &&
            !tbmres->recheck &&
            VM_ALL_VISIBLE(hscan->rs_base.rs_rd, tbmres->blockno, &bscan->rs_vmbuffer))
        {
            Assert(tbmres->ntuples >= 0);
            Assert(bscan->rs_empty_tuples_pending >= 0);

            bscan->rs_empty_tuples_pending += tbmres->ntuples;
            continue;
        }

        return tbmres->blockno;
    }

    /* not reachable */
    Assert(false);
}
```

Iterating through bitmap

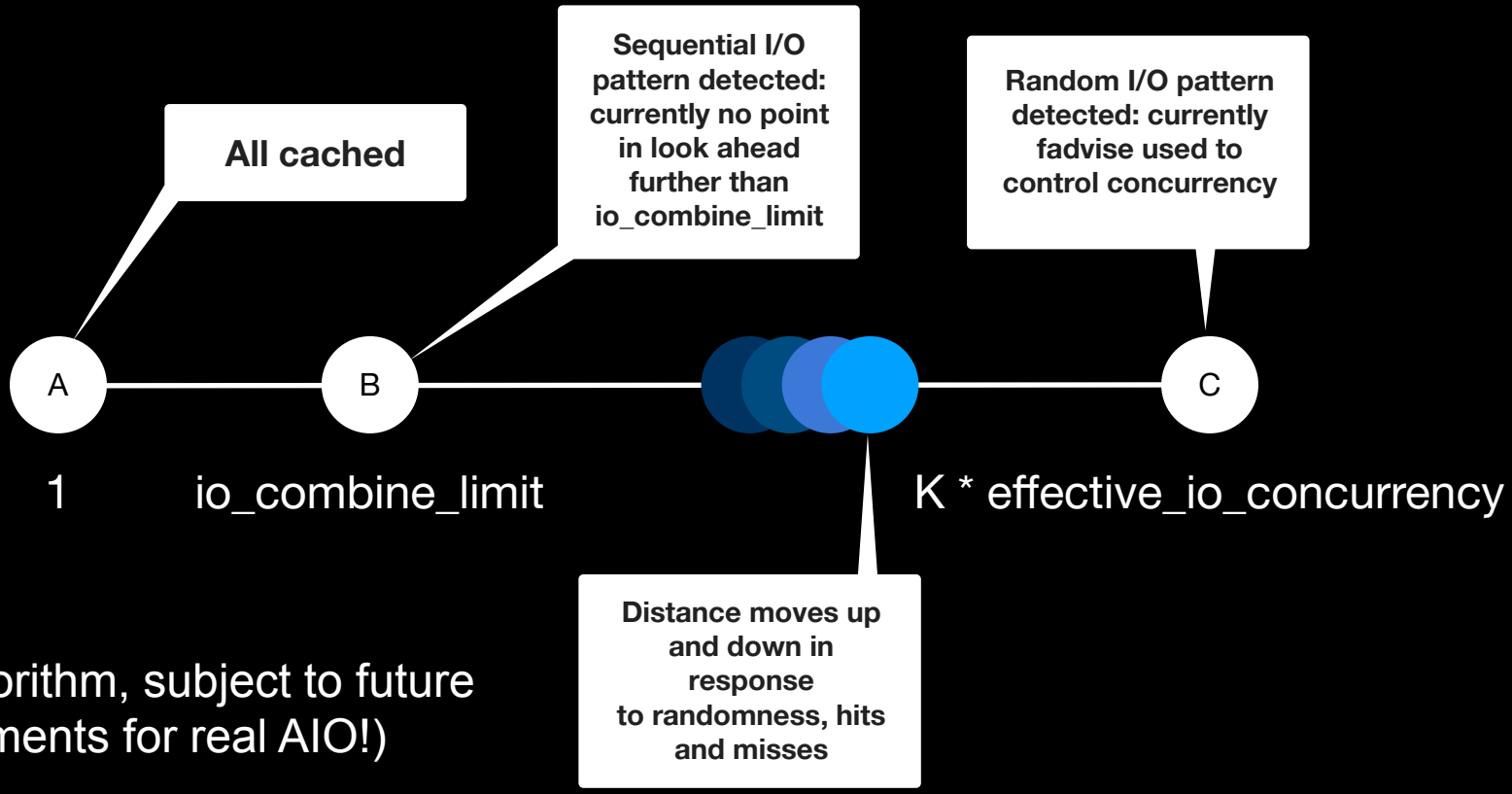
* https://www.postgresql.org/message-id/CAAKRu_ZwCwWFeL_H3ia26bP2e7HiKLWt0ZmGXPVwPO6uXq0vaA%40mail.gmail.com

Deciding how far ahead to look

- A stream doesn’t generally know if e.g. `SELECT … LIMIT 1` needs more than one block, so it starts out reading just a single block and increases the look ahead distance only while that seems to be useful.

- In this way we don’t pay extra overheads such as extra pins and bookkeeping unless there is some benefit to it.

Tuning the look-ahead distance



(V17 algorithm, subject to future Improvements for real AIO!)

Sequential Scan - `strace` output

```
recvfrom(10, "Q\0\0\0002SELECT * from pgbench_accou"..., 8192, 0, NULL, NULL)
pread64() = 8192
preadv() = 16384
preadv() = 32768
preadv() = 65536
preadv() = 131072
preadv() = 131072
preadv() = 131072
preadv() = 131072
…
preadv() = 131072
preadv() = 131072
preadv() = 131072
preadv() = 131072
preadv() = 122880
recvfrom(10, 0x564b68d59b60, 8192, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily
unavailable)
```

Distance increases
quickly up to
io_combine_limit

Random Scans - `strace` output

```
recvfrom(10, "Q\0\0\0\36ANALYZE pgbench_accounts;\0", 8192, 0, NULL, NULL) = 31
pread64(18, "..."..., 8192, 524288) = 8192
fadvise64(18, 548864, 8192, POSIX_FADV_WILLNEED) = 0
pread64(18, "..."..., 8192, 548864) = 8192
fadvise64(18, 737280, 8192, POSIX_FADV_WILLNEED) = 0
fadvise64(18, 950272, 8192, POSIX_FADV_WILLNEED) = 0
fadvise64(18, 1564672, 8192, POSIX_FADV_WILLNEED) = 0
pread64(18, "..."..., 8192, 737280) = 8192
fadvise64(18, 1638400, 8192, POSIX_FADV_WILLNEED) = 0
fadvise64(18, 1974272, 16384, POSIX_FADV_WILLNEED) = 0
fadvise64(18, 2097152, 8192, POSIX_FADV_WILLNEED) = 0
fadvise64(18, 2383872, 8192, POSIX_FADV_WILLNEED) = 0
pread64(18, "..."..., 8192, 950272) = 8192
fadvise64(18, 2400256, 8192, POSIX_FADV_WILLNEED) = 0
fadvise64(18, 2531328, 8192, POSIX_FADV_WILLNEED) = 0
fadvise64(18, 2654208, 8192, POSIX_FADV_WILLNEED) = 0
...
pread64(18, "..."..., 8192, 1564672) = 8192
fadvise64(18, 3276800, 8192, POSIX_FADV_WILLNEED) = 0
pread64(18, "..."..., 16384, 1974272) = 16384
fadvise64(18, 3792896, 8192, POSIX_FADV_WILLNEED) = 0
pread64(18, "..."..., 8192, 2097152) = 8192
```

Issuing POSIX_FADV_WILLNEED early, anticipating later pread

I/O combined when neighbouring blocks are sampled

Some “streamification” projects

Read Stream user	Status
Sequential Scan (heap AM)	v17
ANALYZE (heap AM)	v17
pg_prewarm	v17
CREATE DATABASE (strategy = wal_log)	Committed, v18
pg_visibility	Committed, v18
VACUUM (heap AM)	WIP
autoprewarm	WIP
Bitmap Heap Scan	WIP
Recovery	WIP

Many more opportunities to “streamify” things

- Index scans in core
  - Many types of index need patches to use streams
- Extension AMs
  - Every table AM and index AM is a potential candidate for streamification
  - In v17, extensions that start using streams will benefit from I/O combining and read-ahead advice for random access
- All code that is using the stream abstraction will automatically benefit from future improvements to support true AIO in later releases
- Streams should be the preferred way to access predictable sequences of relation data

Part III: More experimental work on I/O streaming

Research on other kinds of Read Stream

- POC: Multi-relation read stream
  - Developed for recovery/replication; other users are possible
- POC: Automatic read stream
  - Drop-in replacement for traditional ReadBuffer() that speculatively reads ahead with simple consecutive block heuristics, for cases that can’t be easily predicted but today benefit from kernel read-ahead
- POC: Out-of-order streams: return already-cached data first
- POC: Raw files, by-passing the buffer pool
- Ideas: Non-I/O speed-ups may be possible with streams
  - Even for data that is fully cached already and thus don’t need I/O, it can still be useful to look ahead: memory can be prefetched into high cache levels
  - Future work on buffer mapping may use a tree structure, and be able to find consecutive block numbers in memory faster with fewer locks

Experiment: streamifying `pgvector` HNSW search

- Gaph traversals with trivially predictable block access, and also some speculative prediction opportunities
- Streamifying just the easy part already gives measurable speedup and reduced variation with cold indexes (see pgsql-hackers list for patch)
- Cold HNSW may not be interesting in practice… but DiskANN-like indexes (e.g. pgvectorscale) might be a good target?

		linux (xfs)		
branch	eic	avg	speedup	stdev
master		73.959	1.0	24.168
stream	0	70.117	1.1	36.699
stream	1	57.983	1.3	5.845
stream	2	35.629	2.1	4.088
stream	3	28.477	2.6	2.607
stream	4	26.493	2.8	3.691
stream	5	23.711	3.1	2.435
stream	6	22.885	3.2	1.908
stream	7	21.910	3.4	2.153
stream	8	20.741	3.6	1.594
stream	9	22.471	3.3	3.094
stream	10	19.895	3.7	1.695
stream	11	19.447	3.8	1.647
stream	12	18.658	4.0	1.503
stream	13	18.886	3.9	0.874
stream	14	18.667	4.0	1.692
stream	15	19.080	3.9	1.429
stream	16	18.929	3.9	3.469

Writing: WIP

- Initial focus was on an API for reading
  - Reads happen all over the tree
  - Important to make a suitable read abstraction available for wider use ASAP
- Writing happens in fewer more centralised places: WriteStream POCs exist
  - Checkpointer
  - Background writer
  - Evicting individual buffers
  - Evicting buffers used in a BufferAccessStrategy (“ring” of reusable buffers)
  - Raw relation writing that bypasses buffer pool

Part IV: Introduction to true AIO

Andres Freund's proposed AIO subsystem

https://github.com/anarazel/postgres/tree/aio-2 (note 2!)

- Advice-based prefetching is replaced with background reading
  - posix_fadvise(..., POSIX_FADV_WILLNEED), intermediate work, preadv(...) becomes:
  - [start read], intermediate work, [wait for completion]
- Mechanism used is selected with `io_method` setting
  - `synchronous` – portable
  - `worker` – portable
  - `io_uring` – Linux
- Other implementations are possible
  - `iocp` – Windows overlapped
  - `posix_aio` – FreeBSD
  - <extension>? – useful for distributed/network storage projects?

- Anything using the stream abstraction automatically starts using asynchronous I/O
- Running I/O operations are represented as an object in shared memory
- The work done so far on I/O combining and streaming was an architectural change to prepare for DIO and AIO
  - Parellelising the streamification work
  - Avoiding potential regressions

Part V: Trying out AIO patches

Try it yourself

```
$ git remote add andres https://github.com/anarazel/postgres.git
$ git fetch andres aio-2
$ git checkout aio-2
$ cd build
$ ninja install
$ path/to/bin/initdb -D pgdata
$ path/to/bin/postgres -D pgdata
```

* More recent

```
/path/to/bin/postgres -D pgdata
 ├─ postgres: io worker worker: 1
 ├─ postgres: io worker worker: 0
 ├─ postgres: io worker worker: 2
 ├─ postgres: checkpointer
 ├─ postgres: background writer
 ├─ postgres: walwriter
 ├─ postgres: autovacuum launcher
 ├─ postgres: logical replication launcher
 └─ postgres: user postgres [local] idle
```

https://www.postgresql.org/message-id/uvrtrknj4kdytuboidbhwclo4gxhswwcpgadptsjvjqcluzmah%40brqs62irg4dt

io_method = sync

- Works just like v17, no AIO, useful mainly for comparison/understanding
- Synchronous system calls
  - Relying on system read-ahead for sequential access
  - Issuing read-ahead advice for random access
- Performs badly with direct I/O enabled, because read-ahead (heuristic or advice-based) is not possible

io_method = io_worker

- I/O is offloaded to worker processes
- Number of I/O workers is controlled by `io_workers` setting
- Should probably be more dynamic (future work)

- Process tree when io_workers = 3

```
68410 ?     Ss   0:00 postgres: io worker worker: 0
68411 ?     Ss   0:00 postgres: io worker worker: 1
68412 ?     Ss   0:00 postgres: io worker worker: 2
68413 ?     Ss   0:00 postgres: checkpointer
68414 ?     Ss   0:00 postgres: background writer
68416 ?     Ss   0:00 postgres: walwriter
68417 ?     Ss   0:00 postgres: logical replication launcher
```

io_worker:

Query execution process (regular backend):

```
kill(69236, SIGURG) = 0
epoll_wait() = 1
kill(69236, SIGURG) = 0
epoll_wait() = 1
kill(69236, SIGURG) = 0
epoll_wait() = 1
kill(69236, SIGURG) = 0
epoll_wait() = 1
```

- Backend process signals worker process to start a read operations before it needs the data
- In the best case the read is finished before it needs the data, but if not it waits for the I/O worker to finish

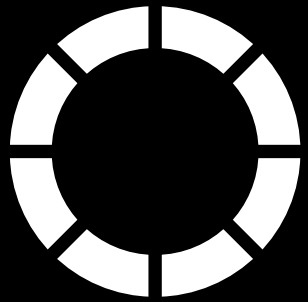
I/O worker process:

```
pread64() = 8192
kill(69247, SIGURG) = 0
pread64() = 16384
kill(69247, SIGURG) = 0
epoll_wait() = 1
pread64() = 32768
kill(69247, SIGURG) = 0
epoll_wait() = 1
pread64() = 65536
kill(69247, SIGURG) = 0
epoll_wait() = 1
pread64() = 131072
kill(69247, SIGURG) = 0
epoll_wait() = 1
pread64() = 131072
```

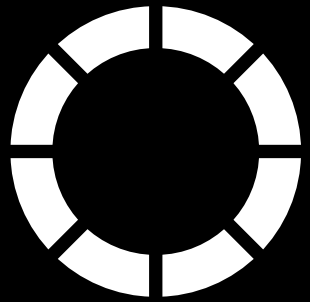
- Worker process does the read
- Then signals backend process, saying the read is finished, but only if it is waiting
- If the queue of I/O requests is empty, it waits for more instructions

io_method = io_uring

submission queue entries



completion queue entries



- `io_uring_enter()`: initiate and/or wait for many operations
- Start multiple operations at once by writing them into a submission queue in user space memory and then telling the kernel
- Consume completion notifications, either directly from user space memory if possible, or by waiting if not

io_method = io_uring

```
recvfrom(138, "Q\0\0\0002SELECT * from pgbench_accou"..., 8192, 0, NULL, NULL) = 51
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
…
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
io_uring_enter(4, 1, 0, 0, NULL, 8) = 1
recvfrom(138, 0x55ba69263d20, 8192, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
```

Simple benchmark results

Configuration:

- `autovacuum = off`
- `effective_io_concurrency = 128`
- `io_combine_limit = 32`

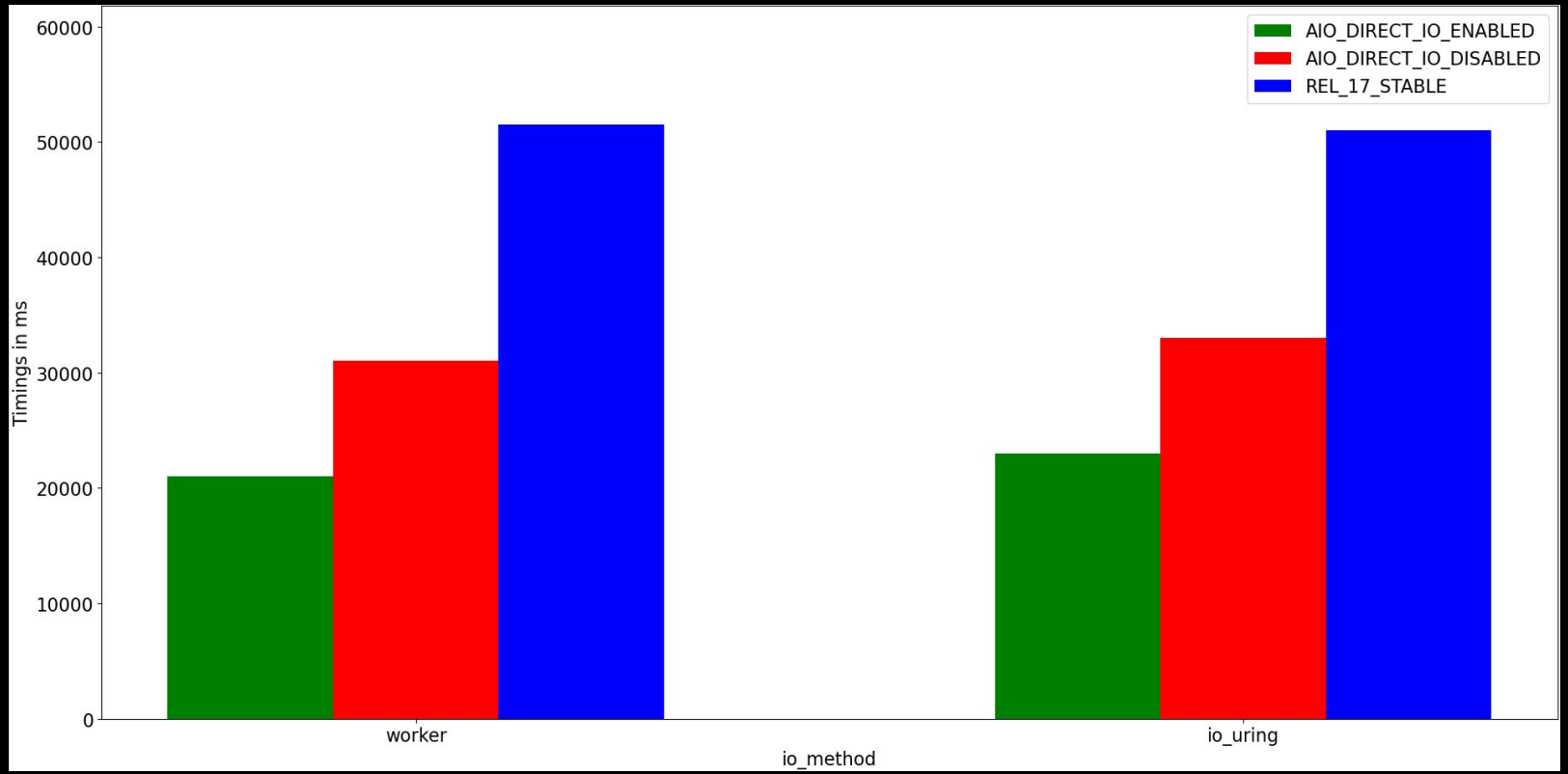
Create table:

- `$ pgbench -i -s 5000 $DB` ⟶ 73 GB table

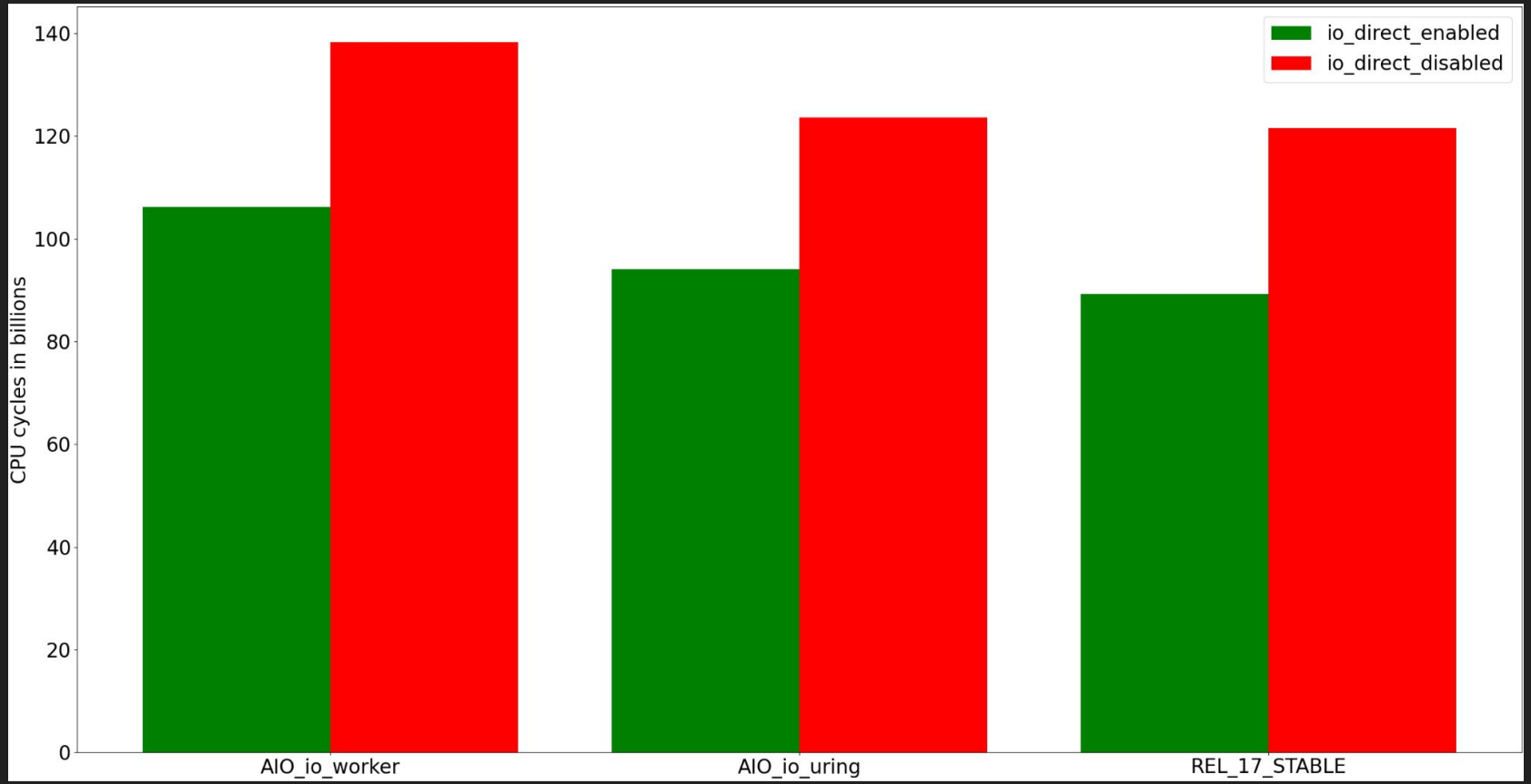
Query:

- `SELECT sum(abalance) FROM pgbench_accounts;`

io_method - Timings



io_direct - CPU cycles



Conclusion

- Streams enable optimisations, current and future
- Consider streamifying your extension or parts of PostgreSQL you are interested in, we’re happy to help if we can!
- If you can’t for technical reasons, we’re very interested to know why and how we can improve the infrastructure
- Try out the AIO v2 patch set

The end
το τέλος