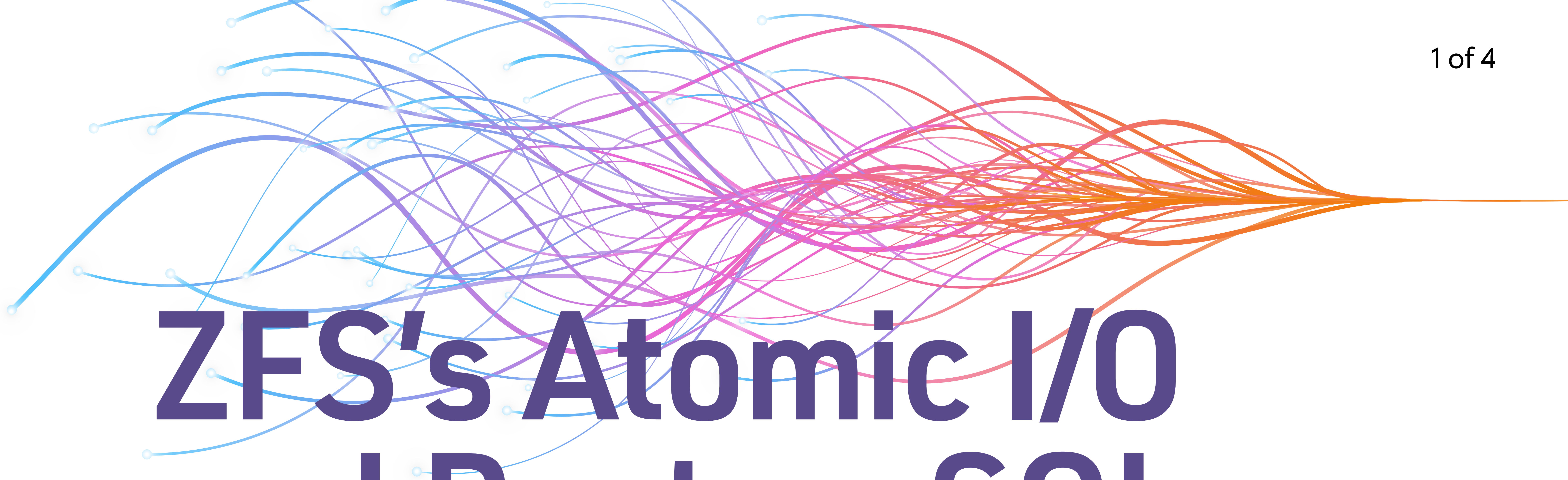

ZFS's Atomic I/O and PostgreSQL

BY THOMAS MUNRO

PostgreSQL is a relational database management system implementing the SQL standard, with a BSD-like license. Its pre-SQL ancestor POSTGRES began at Berkeley University in the mid 1980s. It's popular on FreeBSD, where it is usually deployed on ZFS storage.

Many articles about PostgreSQL on ZFS recommend changing ZFS's `recordsize` setting and PostgreSQL's `full_page_writes` setting. The real impact of the latter setting on performance and crash-safety is not often explained, perhaps because it's not generally safe to adjust it on most popular file systems. In this article I summarize the logic and trade-offs behind this mysterious mechanism—after a brief detour to talk about block sizes.

Blocks

Nearly all of PostgreSQL's disk I/O is aligned on 8KB blocks, or pages. It is possible to recompile it to use a different size, but that is rarely done. This size may originally have been chosen to match UFS's historical default block size (though note that FreeBSD's UFS now defaults to 32KB). ZFS uses the term record size, and defaults to 128KB. Unlike other file systems, ZFS allows the record size to be changed easily at any time, and to be configured separately for each dataset.

POSTGRES began at Berkeley University in the mid 1980s.

If the data will be accessed randomly, then in theory the size should ideally match PostgreSQL's 8KB blocks. Otherwise, random I/O could suffer from two effects:

- I/O amplification, because every read or write of an 8KB block also transfers extra neighboring data
- read-before-write when storage blocks are not currently in the OS's cache and an 8KB block must be written, so the neighboring data must be read first

If the data will be accessed mostly sequentially, or rarely, and especially if the benefits of ZFS compression using larger records outweigh concerns about I/O bandwidth and latency, then it can be a good idea.

Some sources make a blanket recommendation of 16KB, 32KB or 128KB record size, as a sweet spot for better compression without too much write amplification or latency. My aim here isn't to make such recommendations–I doubt there is one answer–but rather to explain what's going on.

Some applications have a mix of requirements for different kinds of data. Tablespaces can be used to store different tables in different ZFS datasets with different record size, compression or physical media. It's also possible for a table to be partitioned, for example with older data in one tablespace and current active data in another.

```
CREATE TABLESPACE compressed_tablespace
LOCATION '/tank/pgdata/compressed_tablespace';

ALTER TABLE t
SET TABLESPACE compressed_tablespace;
```

One problem reported with small ZFS record sizes is fragmentation. A table that receives frequent random updates might finish up with blocks scattered all over the place, and we'd prefer them to be physically clustered for good sequential read performance. A simple way to ask PostgreSQL to rewrite the files that hold a table and its indexes in order to defragment them at the ZFS level would be to issue **`VACUUM FULL table_name`** or **`CLUSTER table_name`**, if you are prepared to lock queries out of the table for the duration of the rewrite. Rewriting a table also allows a new record size to take effect, if it has been changed at the dataset level.

Torn Writes

The PostgreSQL setting **`full_page_writes`** defaults to on, and ZFS users often turn it off. The performance of write-intensive workloads then becomes faster and more consistent. For example, in a simple pgbench test on a low end cloud VM I measured a 32% increase in transactions per second by turning it off.

So what does it really do? That requires a surprising amount of background explanation.

The short version is that PostgreSQL uses *physiological logging* for crash safety, and that means that writes to individual database pages must be *atomic on power failure*, or it may not be able to recover after a crash. Unless you promise that your storage stack has that property, then PostgreSQL has to do some extra work to protect your data.

Atomicity on power failure is the property that if a physical write was in progress when power was lost, later we can expect to read back either the old version or the new version of a block, for some given block size, but not a partially modified or torn version. This is not to be confused with atomicity of concurrent reads and writes (see below). Physiological logging, short for *physical-to-the-page*, *logical-within-the-page*, is a term from textbook classifications of logging strategies, and it means that log records identify a block to be changed by file and block number, but then describe the change to make within that page in a notation that requires us to read in the existing page to understand how to modify it "logically", rather than just updating bits at a physical address.

After a crash, the recovery algorithm can cope with the "old" page contents or the "new" page contents, applying any logged changes required to bring it up to date. If it encounters a non-atomic mash-up of old and new data, then logical changes to the page cannot be replayed, and recovery fails! A superficial problem is that if **`data_checksums`** is enabled, then PostgreSQL's page-level checksum check will fail even to read the page in. If checksums are disabled, we'll get further, but a logical change such as "insert tuple (42,Fred) in slot 3" can't be replayed reliably. In order to apply the change in this example we need to understand a table of slots using pre-existing meta-data on the page, but it's potentially corrupted.

Physiological logging is a very widely used technique in the database industry, and different RDBMSs have found different solutions to the problem of torn pages. Since open source systems have been developed and used on a wide variety of low end systems often without various forms of hardware protection against power loss, failures were common and software solutions had to be developed.

PostgreSQL's current solution is to switch to page-level physical-only logging or *full page writes*, where the whole data page is dumped into the log, for the first modification to each data page after each checkpoint. Checkpointing is a periodic background activity, and in an ideal world would have minimal effects on foreground transaction performance. However, due to the first-touch rule, once a checkpoint starts, write-heavy workloads might suddenly start generating a lot more log data, as small updates suddenly require many 8KB pages to be logged. This effect typically decays gradually because subsequent modifications to each page go back to being physiological, until the next checkpoint, sometimes resulting in a sawtooth pattern in I/O bandwidth and transaction latency.

Another popular open source database has a different solution that also involves writing all data out twice with a synchronization barrier between, since both copies can't be torn.

ZFS needs none of that! It has record-level atomicity by virtue of its own copy-on-write design. It's not possible to see a mixture of the old and new contents of a ZFS record, because it doesn't physically overwrite them, and its system of TXGs and the ZIL makes writes transactional. Therefore, it is safe to set **`full_page_writes=off`** as long as **`recordsize`** is at least 8KB.

It's not possible to see a mixture of the old and new contents of a ZFS record.

Note that ZFS itself also physically writes data twice in some scenarios. A common recommendation is to consider setting **`logbias=throughput`** for the dataset holding the main data files (but perhaps not the one holding PostgreSQL's log directory **`pg_wal`**—a topic not explored in this article). That option tries to write blocks directly into their final location instead of logging them first in the ZIL. If you use the ZFS default **`logbias=latency`** and the PostgreSQL default **`full_page_writes=on`**, data may in fact be written out four times in total as both PostgreSQL and ZFS perform extra work to create record-level atomicity, while both of those changes bring it down to one copy.

Unfortunately there are two special scenarios where **`full_page_writes=on`** is still needed for correct behavior: while running **`pg_basebackup`** and **`pg_rewind`**. Those tools are used for backups, or to create or re-synchronize streaming replicas from another server; in the case of **`pg_basebackup`**, full page writes will be silently enabled while running the command, while in the case of **`pg_rewind`**, the command will refuse to run if it is not manually enabled (an annoying inconsistency in current releases). These tools make raw file system-level copies of data files, along with the logs required for crash recovery to deal with consistency problems caused by concurrent changes. Here we run into a different meaning of I/O atomicity: reading from a file that might be concurrently written to. The first problem is that file systems on Linux and Windows (but not ZFS, or any file system on FreeBSD, due to the use of range locks) can show readers a random selection of before and after bits when there is an overlapping concurrent write. Furthermore, the I/O is currently done in a way that isn't suitably aligned, so even on ZFS, torn pages could be copied. To defend against that, `full_page_writes` behavior is needed. This problem should eventually be fixed in PostgreSQL, by copying the raw data files with appropriate alignment and interlocking. Note that ZFS snapshots can be used instead of `pg_basebackup` if certain precautions are taken (primarily that the snapshot must atomically capture the logs and all data files), thus reducing the impact when cloning or backing up a very busy system.

Recovery

We've seen how `full_page_writes=off` improves the performance of write transactions, and ZFS makes that safe. Unfortunately there can also be negative performance implications for replication and crash recovery. These activities both perform *recovery*, meaning that they replay the log. Although full page images are a pessimization when they're written, they act as an optimization when they're replayed at recovery time. Instead of having to perform a random synchronous read that might block recovery's serial processing loop, we have the contents of the page to be modified already in our nice sequential log, and after that it is cached.

PostgreSQL 15 includes a partial solution to this problem: it looks ahead in the log to find pages that will soon be read, and issues `POSIX_FADV_WILLNEED` advice, to generate a configurable degree of I/O concurrency (a sort of poor man's asynchronous I/O). At the time of writing, FreeBSD ignores the advice, but a future version of OpenZFS will hopefully connect it up to FreeBSD's VFS (OpenZFS pull request #13958). Eventually, this should be replaced by a true asynchronous I/O subsystem that is currently being developed and proposed for a future version of PostgreSQL.

The effect of `full_page_writes=off` on recovery I/O stalls was studied by a group using PostgreSQL on ZFS on the illumos operating system at scale. They developed a tool called `pg_prefaulter` as a workaround. They had found that their streaming replicas couldn't keep up with their primary servers due to predictable I/O stalls. They may have been uniquely placed to see this effect since most large scale users of PostgreSQL don't even have the option of setting `full_page_writes=off`. `pg_prefaulter` may be a solution if you run into this problem, until built-in prefetching is available.

Looking Ahead

Block size alignment is likely to become a bigger topic in future PostgreSQL releases that will hopefully include proposed direct I/O support, which for now exists only in prototype form. This coincides happily with the development of direct I/O support for OpenZFS (pull request #10018), which will probably require block size agreement to work effectively (the current prototype reverts to the ARC otherwise; some other file systems simply refuse non-aligned direct I/O). Another OpenZFS feature in the works that is likely to be very useful for databases is block cloning (pull request #13392), along with new systems interfaces for FreeBSD, which PostgreSQL should hopefully be able to use for fast cloning of databases and database objects with finer granularity than whole datasets.

THOMAS MUNRO is an open source database hacker working for Microsoft Azure, who is usually logged into a FreeBSD box.