Scaling WAL Performance

Eliminate replication lag and reduce startup times with pg_prefaulter
What is WAL?

Write Ahead Log
Where is WAL?

The "heap" (a.k.a. your data)

WAL files
% ls -lA $PGDATA/pg_xlog/
-rw------- 1 seanc staff 16777216 May 31 12:02 $PGDATA/pg_xlog/000000010000000000000001
-rw------- 1 seanc staff 16777216 May 31 12:02 $PGDATA/pg_xlog/000000010000000000000002
-rw------- 1 seanc staff 16777216 May 31 12:02 $PGDATA/pg_xlog/000000010000000000000003
-rw------- 1 seanc staff 16777216 May 31 12:02 $PGDATA/pg_xlog/000000010000000000000004
Heaps of SQL

```
postgres@[local]:5432/postgres# CREATE DATABASE test;
CREATE DATABASE
Time: 358.395 ms
^Z
% tree -ld $PGDATA/base
|-- 1
| -- 12668
|-- 12669
|-- 16387
4 directories
```

Creates new DB

New directory
Table Data as Files

postgres@[local]:5432/postgres# \c test
You are now connected to database "test" as user "postgres".
postgres@[local]:5432/test# CREATE TABLE t1 (i INT);
CREATE TABLE
Time: 2.273 ms
postgres@[local]:5432/test# SELECT pg_relation_filepath('t1');
  pg_relation_filepath
----------------------
   base/16387/16388
(1 row)
Time: 1.160 ms
^Z
% stat -f "%-Sp %z %N" $PGDATA/base/16387/16388
-rw------- 0 $PGDATA/base/16387/16388

Empty file
Physical Storage of Data

postgres@[local]:5432/test# INSERT INTO t1 VALUES (1);
INSERT 0 1
Time: 0.581 ms
^Z
% stat -f "%Sp %z %N" $PGDATA/base/16387/16388
-rw------- 8192 $PGDATA/base/16387/16388
% fg
postgres@[local]:5432/test# INSERT INTO t1 VALUES (2);
UPDATE 1
Time: 5.985 ms
^Z
% stat -f "%Sp %z %N" $PGDATA/base/16387/16388
-rw------- 8192 $PGDATA/base/16387/16388

PG Page Size (8K)
How does the WAL relate to the heap?

1. Modifications to the heap are appended to the WAL first.

2. Committed transactions in the WAL are applied in the heap during a CHECKPOINT.

3. Crash recovery walks backwards through the WAL to the last completed CHECKPOINT (then rolls forward through committed transactions to prevent data loss).
## Things to keep in mind

<table>
<thead>
<tr>
<th>Write</th>
<th>WAL</th>
<th>Ahead</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The WAL receives sequential append operations</td>
<td>2. WAL can be read forward and backwards</td>
<td>3. Recently written transaction data exists only in memory and in WAL</td>
<td>4. WAL is <em>probably</em> your performance friend (deferred random IO against the heap)</td>
</tr>
</tbody>
</table>
Tuples, Pages, Relations, and you!

https://www.postgresql.org/docs/current/static/wal.html
Why do you care about apply lag?

`synchronous_commit="remote_write"`
Manta is an HTTP Frontend to ZFS

- Files distributed across different ZFS storage servers
- Metadata stored in PostgreSQL

Caution: shapes in the diagram may appear more simple than they actually are
PostgreSQL Replication is Awesome

\[\text{synchronous\_commit} = "XXX"\]

\[\text{PG}_{\text{primary}} \rightarrow \text{PG}_{\text{follower}} \rightarrow \text{PG}_{\text{async}}\]
ez-mode HA Durability FTW

synchronous_commit="XXX"

PG_{primary} \xrightarrow{remote\_write} PG_{follower}

PG_{async} \xrightarrow{on}
Hardware fails right on time, every time

synchronous_commit="XXX"

remote_write

PG_follower

PG_async

on
CAP: Can haz A?
This isn't a hardware problem

synchronous_commit="XXX"

remote_write

PGfollower

on

PGasync
It's gunna be a while, m'kay?

HINT: That's 19hrs of apply lag
How did we get into this mess?
Cloudy with a chance of single threaded execution
Context is everything

INSERT INTO...

PG_{primary} -> WAL Stream -> PG_{follower}
"Many [INSERTS], handle it!"
Context is everything?

\[ \text{INSERT INTO...} \]

\[ \text{PG}_{\text{primary}} \rightarrow \text{WAL Stream} \rightarrow \text{PG}_{\text{follower}} \]
OH HAI!

INSERT INTO...

**PG** \(_{\text{primary}}\)

\[ \text{pg pg pg pg pg pg} \]

WAL Sender

WAL Stream

**PG** \(_{\text{follower}}\)

WAL Receiver
If we're lucky...

Userspace:
WAL Receiver

filesystem Cache

Disk IO

WAL Page
But we're not because **REALITY**

Diagram:

- **Userspace:** WAL Receiver
- **Filesystem Cache**
- **Disk IO**

- WAL Page → Filesystem Cache → Disk IO

---

**Note:** The diagram illustrates a workflow where data moves from the WAL Receiver to the Filesystem Cache and then to Disk IO.
And I lied to you. This:

Userspace: WAL Receiver

WAL Page

Filesystem Cache

Disk IO
...is actually this.

Userspace:
WAL Receiver

Filesystem Cache

Disk IO

WAL Page

~5-10µs
~5-10µs
~5-10µs
~5-10µs
~5-10µs
And this isn't drawn to scale...

Userspace: WAL Receiver

Filesystem Cache

Disk IO

WAL Page

~5-10µs

~5-10µs

~10-30ms

~5-10µs

~10-30ms
Pixel Correct Timeline

WAL Page → Userspace: WAL Receiver → Filesystem Cache → Disk IO

5µs == 0.1pt

15ms == 300pt
THIS JUST IN

DISK IO CAN BE OVER 3000X SLOWER THAN MEMORY.
And that RAID array you have? It's Idle.

Storage math:

150 iops/disk * 16 disks = ~2400 IOPS (if perfectly scheduled)
And that RAID array you have?
It's Idle.

• Storage math:
  150 iops/disk * 16 disks = ~2400 IOPS
And that RAID array you have? It's Idle.

- Storage math: 
  150 iops/disk * 16 disks = ~2400 IOPS

- Single WAL Receiver process issuing `pread(2)`

- Max 150 IOPS or ~6% utilization of disks

- Busy primaries will overrun followers, permanently
It's gunna be a while, m'kay?
Fixed It
Installation

1. Install Go
2. go get github.com/joyent/pg_prefaulter
3. Configure
4. Run
Configuration

[log]
# level can be set to "DEBUG", "INFO", "WARN", "ERROR", or "FATAL"
#level = "INFO"

[postgresql]
#pgdata = "pgdata"
#database = "postgres"
#host = "/tmp"
#password = ""
#port = 5432
#user = "postgres"

[postgresql.xlog]
#pg_xlogdump-path = "/usr/local/bin/pg_xlogdump"
Run: Primary

% env PGPASSWORD=`cat .pwfile` ./pg_prefaulter run --config pg_prefaulter-primary.toml
2018-05-31T11:59:01.41391821-04:00 DEBU <nil> config-file=pg_prefaulter-primary.toml
2018-05-31T11:59:01.414189771-04:00 DEBU args: []
2018-05-31T11:59:01.414315299-04:00 DEBU starting gops(1) agent
2018-05-31T11:59:01.414475394-04:00 DEBU starting pprof endpoing agent pprof-port=4242
2018-05-31T11:59:01.415005542-04:00 INFO Starting pg_prefaulter pid=39865
2018-05-31T11:59:01.417634192-04:00 DEBU filehandle cache initialized filehandle-cache-size=2000 filehandle-cache-ttl=300000 rlimit-nofile=7168
2018-05-31T11:59:01.426437960-04:00 INFO started IO worker threads io-worker-threads=3600
2018-05-31T11:59:01.426437960-04:00 INFO started WAL worker threads wal-worker-threads=4
2018-05-31T11:59:01.455209806-04:00 DEBU Starting wait
2018-05-31T11:59:01.498278613-04:00 DEBU established DB connection backend-pid=39867 version="PostgreSQL 9.6.3 on x86_64-apple-darwin16.5.0, compiled by Apple LLVM version 8.1.0 (clang=802.0.42), 64-bit"
2018-05-31T11:59:01.513085485-04:00 INFO skipping REDO record for database database=0 input="rmgr: Heap len (rec/tot): 14/ 469, tx: 4, lsn: 0/01007750, prev 0/01007728, desc: HOT_UPDATE off 1 xmax 4 ; new off 3 x max 0, blkref #0: rel 1664/0/1260 blk 0 FPW"
2018-05-31T11:59:01.513213488-04:00 INFO skipping REDO record for database database=0 input="rmgr: Heap len (rec/tot): 2/ 337, tx: 0, lsn: 0/01007988, prev 0/01007950, desc: INPLACE off 1, blkref #0: rel 1664/0/1262 blk 0 FPW"
2018-05-31T11:59:01.558219381-04:00 INFO skipping REDO record for database database=0 input="rmgr: Heap len (rec/tot): 3/ 80, tx: 22, lsn: 0/0116B050, prev 0/0116B028, desc: INSERT+INIT off 1, blkref #0: rel 16$
Run: Followers

% env PGPASSWORD=Kdr6zmvYOgWTKnol7HcULw91o15KhA6c ./pg_prefaulter run --config pg_prefaulter-follower.toml --pprof-port=4243
2018-05-31T12:02:15.364191007-04:00 | DEBU | <nil> config-file=pg_prefaulter-follower.toml
2018-05-31T12:02:15.364357715-04:00 | DEBU | args: []
2018-05-31T12:02:15.364448823-04:00 | DEBU | starting gops(1) agent
2018-05-31T12:02:15.364508931-04:00 | DEBU | starting pprof endpoint agent pprof-port=4243
2018-05-31T12:02:15.365189238-04:00 | INFO | Starting pg_prefaulter pid=40018
2018-05-31T12:02:15.367508589-04:00 | DEBU | filehandle cache initialized filehandle-cache-size=2000
2018-05-31T12:02:15.367522308-04:00 | INFO | started IO worker threads io-worker-threads=3600
2018-05-31T12:02:15.367525672-04:00 | DEBU | started WAL worker threads wal-worker-threads=4
2018-05-31T12:02:15.377063872-04:00 | DEBU | Starting wait
2018-05-31T12:02:15.377104519-04:00 | INFO | Starting pg_prefaulter agent commit=none date=unknown tag=version=dev
2018-05-31T12:02:15.413981503-04:00 | DEBU | established DB connection backend-pid=40019 version="PostgreSQL 9.6.3 on x86_64-apple-darwin16.5.0, compiled by Apple LLVM version 8.1.0 (clang-802.0.42), 64-bit"
2018-05-31T12:02:15.414627296-04:00 | DEBU | found redo WAL segment from DB type=redo
walfile=000000010000000000000000004
What's the voodoo?
pg_prefaulter(1) Design

1. Find WAL files
2. Process WAL files using \texttt{pg_xlogdump(1)}
3. Read the text output from \texttt{pg_xlogdump(1)}
4. Translate output into offsets into relations (i.e. tables/indexes)
5. Dispatch \texttt{pread(2)} calls in parallel
6. Warm the OS cache before the WAL apply process faults a page in by itself
7. Dump all internal caches if process notices primary/follower change
8. Profit (or at least, fail less hard on failover or startup)
Finding WAL Files

1. Connect to PostgreSQL
2. Search for hints in process titles
• Platform and WAL file version agnostic way of extracting WAL information
• Elided the need for writing a customer WAL parser
pg_prefaulter(1) Architecture

- **OS**
- **WAL File**
- **WAL File Scanner**
- **WAL Filename Cache**
- **IO Request Cache**
- **FD Cache**
- **IO Thread N**
- **pread(2)**
- **pg_xlogdump(1)**
- **Proc Titles**
- **System Catalogs**
- **PostgreSQL**
- **WAL Receiver**

Diagram shows the interaction between these components and the layers of the PostgreSQL architecture.
Requirements

1. PostgreSQL 9.6
   (an update to support 10 and 11 is coming soon)

2. Go compiler to build the binary

3. `pg_xlogdump(1)`
Where to use **pg_prefaulter(1)**

1. On the primary
2. On all followers
3. Useful at startup for primaries and followers
4. Useful for promotion of followers
5. Useful on standalone PostgreSQL instances not using replication
6. Any database that you want to see start faster or where you care about availability (i.e. everywhere, on all PG instances)
7. Any PostgreSQL database that replicates and **VACUUM**s or **pg_repack(1)**s - i.e. generates lots of WAL activity
Don't be laggin' like this...
Be prefaultin' like this!

pg_prefaulter deployed
Recovery Visualized

Falling behind at 0.2s/s

Falling behind at 0.8s/s

pg prefaulter deployed

Recovering at -0.6s/s

Fully Recovered
Steady As She Goes
Thank you!

https://github.com/joyent/pg_prefaulter

We're Hiring!

@SeanChittenden
seanc@joyent.com
seanc@FreeBSD.org
sean@chittenden.org