Postgres and the Write-Ahead Log

Abhijit Menon-Sen
2ndQuadrant

February 23, 2018

This talk is about…

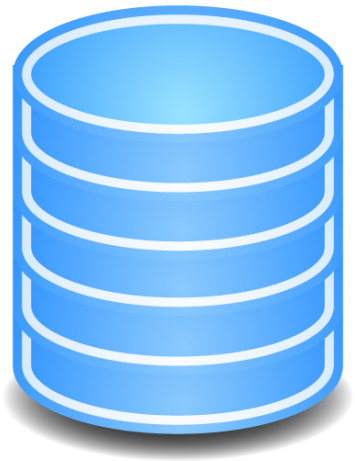
Write-Ahead Logging

Atomicity and Durability (in **A**CI**D**)

Replication in Postgres

Let's pretend to be a database

Try to feel like this:



Our first transaction

```
BEGIN;

UPDATE t SET a=42 WHERE b=1;

COMMIT;
```

Lots of complexity

Parsing, planning, execution, …

Catalogue lookups, table scan, …

We care only about writing the data

Writing the changes

Open a file

Seek to some position

Write some bytes

The simplest code

```
f = open(filename, O_RDWR);
seek(f, changepos);
write(f, somebytes, n);
close(f);
```

Well, why not?

Correctness and performance

Atomicity: All or nothing

(**C** and **I** not relevant here)

Durability: Don't lose committed transactions

Maintain high performance

Hostile environment

Power failures

Hardware and software crashes

(Disks can crash too)

WAL manages risk of interaction
with storage

Storage subsystem

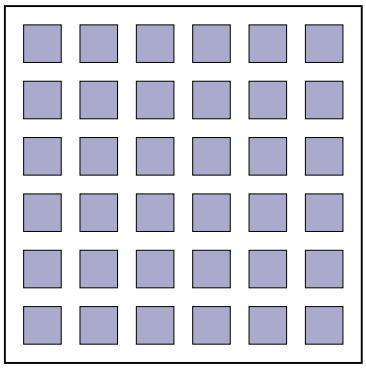
Many different kinds of devices

Wildly varied implementations

Varied characteristics, performance

Hidden behind "block device" abstraction

Block devices



Block device characteristics

Fixed-size blocks (e.g., 512B, 4KB)

Read or write one block at a time

Atomic writes of single blocks

Slow (compared to RAM)

Disks are slow

Caches everywhere

In hardware (controller, disks…)

In software (kernel, postgres…)

Write-through vs. Write-back

Flushing caches (extra slow)

Block-sized I/O

Bytes? Forget performance

Think in blocks instead

Somebody has to do it

Single-block writes are atomic

Writing multiple blocks?

Be prepared to lose some writes

Torn pages

(Note: not **A**CID atomicity)

Summary

Power may fail anytime

Pages may be "torn" when written

We need atomicity and durability

But it still has to be fast

Block management

Keep copies of disk blocks in RAM

Make all changes in memory

Mark modified pages "dirty"

Write out whole dirty pages

The kernel

Converts Byte ↔ Block I/O

Maintains a buffer cache

`dirty_ratio`, `dirty_background_ratio`, `dirty_writeback_centisecs`

Scheduling requests at block device layer

Back to the server

Postgres also caches 8KB pages (`shared_buffers`)

(The kernel buffer cache doesn't provide enough control)

Dirty buffers written out by `bgwriter`

Write-Ahead Logging

One possible solution

Not Postgres-specific

An alternative: command logging

How WAL works

Log file alongside the database

Log a description of each change

Flush the log to disk

Tell the client the transaction has committed

Make the changes

WAL provides crash safety

Durable log file

Complete change description

Changes can be replayed after a crash

Scenario #1: crash before log

We didn't tell the client "OK"

We didn't make changes to the data

Nothing to do

Scenario #2: crash during log

We crashed while writing a log entry

Record may be partially written

Checksum OK → Record OK → replay

Checksum bad? Ignore record

Scenario #3: crash after log

We may have said OK to client

We may have made some changes

Complete log record available

Just replay the record

Committed changes recovered

Checkpoints

Data page changes are written asynchronously

A checkpoint writes out and flushes all dirty buffers

Database on disk is up-to-date with all committed transactions

Limits WAL replay after crash

WAL in Postgres

Lives in pg_wal

A sequence of 16MB "segments"

Older WAL segments are recycled

Each record has an LSN/address

Generate some WAL records

```
BEGIN;

UPDATE t SET a=42 WHERE b=1;

COMMIT;
```

We want to see only this transaction's WAL

Switch to a new WAL segment

```
ams=# select pg_current_wal_lsn(),
             pg_walfile_name(pg_current_wal_lsn());
 pg_current_wal_lsn |     pg_walfile_name
--------------------+--------------------------
 0/164D958          | 000000010000000000000001
(1 row)

ams=# select pg_switch_wal();
 pg_switch_wal
---------------
 0/164D970
(1 row)
```

After the transaction

```
ams=> BEGIN; UPDATE t SET a=42 WHERE b=1; COMMIT;
BEGIN
UPDATE 3
COMMIT

ams=# select pg_switch_wal();
 pg_switch_wal
---------------
 0/2000278
(1 row)
```

Three WAL segments

```
ams=# select * from pg_ls_waldir() order by modification asc;
           name           |   size   |       modification
--------------------------+----------+---------------------------
 000000010000000000000001 | 16777216 | 2018-02-23 03:07:54+05:30
 000000010000000000000002 | 16777216 | 2018-02-23 03:09:20+05:30
 000000010000000000000003 | 16777216 | 2018-02-23 03:10:15+05:30
(3 rows)
```

pg_waldump

The WAL observer's best friend

Takes a segment name as input

Describes each record

Can summarise record types

The second WAL segment

```
postgres@kea:~/10/main/pg_wal$ /usr/lib/postgresql/10/bin/pg_waldump ./000000010000000000000002
rmgr: Standby     len (rec/tot):     50/    50, tx:          0, lsn: 0/02000028, prev 0/0164D958,
desc: RUNNING_XACTS nextXid 563 latestCompletedXid 562 oldestRunningXid 563
rmgr: Heap        len (rec/tot):     65/   269, tx:        563, lsn: 0/02000060, prev 0/02000028,
desc: HOT_UPDATE off 1 xmax 563 ; new off 5 xmax 0, blkref #0: rel 1663/16387/16388 blk 0 FPW
rmgr: Heap        len (rec/tot):     69/    69, tx:        563, lsn: 0/02000170, prev 0/02000060,
desc: HOT_UPDATE off 2 xmax 563 ; new off 6 xmax 0, blkref #0: rel 1663/16387/16388 blk 0
rmgr: Heap        len (rec/tot):     69/    69, tx:        563, lsn: 0/020001B8, prev 0/02000170,
desc: HOT_UPDATE off 3 xmax 563 ; new off 7 xmax 0, blkref #0: rel 1663/16387/16388 blk 0
rmgr: Transaction len (rec/tot):     34/    34, tx:        563, lsn: 0/02000200, prev 0/020001B8,
desc: COMMIT 2018-02-23 03:09:13.457168 IST
rmgr: Standby     len (rec/tot):     50/    50, tx:          0, lsn: 0/02000228, prev 0/02000200,
desc: RUNNING_XACTS nextXid 564 latestCompletedXid 563 oldestRunningXid 564
rmgr: XLOG        len (rec/tot):     24/    24, tx:          0, lsn: 0/02000260, prev 0/02000228,
desc: SWITCH
```

The changes we made

```
postgres@kea:~/10/main/pg_wal$ /usr/lib/postgresql/10/bin/pg_waldump ./000000010000000000000002
rmgr: Heap        len (rec/tot):     65/   269, tx:        563, lsn: 0/02000060, prev 0/02000028,
desc: HOT_UPDATE off 1 xmax 563 ; new off 5 xmax 0, blkref #0: rel 1663/16387/16388 blk 0 FPW
rmgr: Heap        len (rec/tot):     69/    69, tx:        563, lsn: 0/02000170, prev 0/02000060,
desc: HOT_UPDATE off 2 xmax 563 ; new off 6 xmax 0, blkref #0: rel 1663/16387/16388 blk 0
rmgr: Heap        len (rec/tot):     69/    69, tx:        563, lsn: 0/020001B8, prev 0/02000170,
desc: HOT_UPDATE off 3 xmax 563 ; new off 7 xmax 0, blkref #0: rel 1663/16387/16388 blk 0
rmgr: Transaction len (rec/tot):     34/    34, tx:        563, lsn: 0/02000200, prev 0/020001B8,
desc: COMMIT 2018-02-23 03:09:13.457168 IST
```

The segment switch

```
postgres@kea:~/10/main/pg_wal$ /usr/lib/postgresql/10/bin/pg_waldump ./000000010000000000000002
rmgr: XLOG        len (rec/tot):     24/    24, tx:          0, lsn: 0/02000260, prev 0/02000228,
desc: SWITCH
```

A checkpoint

```
rmgr: XLOG        len (rec/tot):    106/   106, tx:          0, lsn: 0/04000098, prev 0/04000060,
desc: CHECKPOINT_ONLINE redo 0/4000060; tli 1; prev tli 1; fpw true; xid 0:564; oid 24576; multi 1;
offset 0; oldest xid 548 in DB 1; oldest multi 1 in DB 1; oldest/newest commit timestamp xid: 0/0;
oldest running xid 564; online
```

Deletes and inserts

```
rmgr: Heap        len (rec/tot):     59/   335, tx:        564, lsn: 0/04000108, prev 0/04000098,
desc: DELETE off 4 KEYS_UPDATED , blkref #0: rel 1663/16387/16388 blk 0 FPW
rmgr: Heap        len (rec/tot):     54/    54, tx:        564, lsn: 0/04000258, prev 0/04000108,
desc: DELETE off 5 KEYS_UPDATED , blkref #0: rel 1663/16387/16388 blk 0
rmgr: Heap        len (rec/tot):     54/    54, tx:        564, lsn: 0/04000290, prev 0/04000258,
desc: DELETE off 6 KEYS_UPDATED , blkref #0: rel 1663/16387/16388 blk 0
rmgr: Heap        len (rec/tot):     54/    54, tx:        564, lsn: 0/040002C8, prev 0/04000290,
desc: DELETE off 7 KEYS_UPDATED , blkref #0: rel 1663/16387/16388 blk 0
rmgr: Transaction len (rec/tot):     34/    34, tx:        564, lsn: 0/04000300, prev 0/040002C8,
desc: COMMIT 2018-02-23 03:28:23.494133 IST
rmgr: Heap        len (rec/tot):     63/    63, tx:        565, lsn: 0/04000360, prev 0/04000328,
desc: INSERT off 8, blkref #0: rel 1663/16387/16388 blk 0
rmgr: Heap        len (rec/tot):     63/    63, tx:        565, lsn: 0/040003A0, prev 0/04000360,
desc: INSERT off 9, blkref #0: rel 1663/16387/16388 blk 0
rmgr: Transaction len (rec/tot):     34/    34, tx:        565, lsn: 0/040003E0, prev 0/040003A0,
desc: COMMIT 2018-02-23 03:28:38.134116 IST
```

WAL configuration

`checkpoint_timeout`: system load vs. recovery time

`checkpoint_completion_target`: reducing impact of checkpoints

`wal_buffers`: more WAL records → bigger

`max_wal_size`, `min_wal_size`: how many WAL segments

A partially written apology

WAL for atomicity and durability

In theory: A-OK

In Postgres: well…

Postgres uses MVCC instead

Postgres *happens to* use MVCC instead

Replication

Data plus all changes

For local crash recovery

What if we aren't local?

What if we didn't crash?

Voilà! Replication!

Copy data, send WAL

A replica starts with a copy of the database

It has to get WAL somehow

Log shipping, streaming

Minor operational differences

Standby records

```
rmgr: Standby     len (rec/tot):     50/    50, tx:          0, lsn: 0/02000028, prev 0/0164D958,
desc: RUNNING_XACTS nextXid 563 latestCompletedXid 562 oldestRunningXid 563
rmgr: Standby     len (rec/tot):     50/    50, tx:          0, lsn: 0/02000228, prev 0/02000200,
desc: RUNNING_XACTS nextXid 564 latestCompletedXid 563 oldestRunningXid 564
```

Logical replication

Still based on WAL

No physical block references

Change targets identified by primary key

Changes described as columns and values

Selective replay

Questions?

Thank you

(You can stop pretending to be a database now.)