Recovering from a Split Brain

(starring pg_waldump and pg_rewind)

About Me

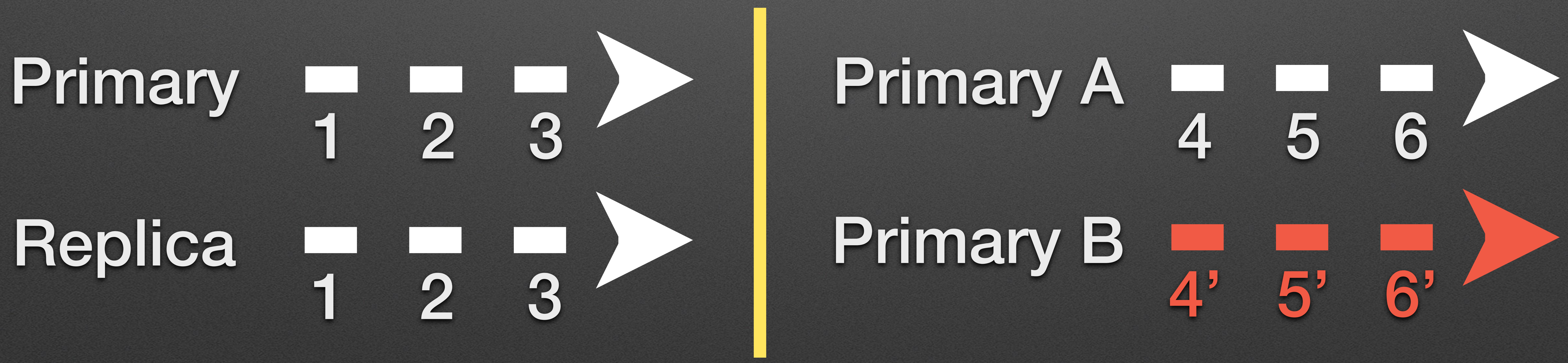
- Software Engineer
  - Team DB
  - Braintree Payments
- PostgreSQL Contributor

Background

- PostgreSQL clusters are often deployed with at least two nodes: a primary and a synchronous replica (via physical replication).
- Typically availability of nodes in that cluster is managed automatically by external control.
  - In our case, Pacemaker manages failovers, and, before promoting a replica fences/STONITHs the primary via PDU control.
- We would rather take an outage than suffer a split brain.

What’s a Split-Brain?

- In a cluster of database nodes, a *split brain* occurs when (often due to some kind of network partition) multiple nodes believe they are the primary node.
- Suppose we have a timeline of operations:



Sidebar: HA Configuration

- Unless you absolutely value uptime over data consistency, a failure to fence the current primary must mean failing to promote a new primary.

- Understanding tradeoffs between availability and consistency is important.

  - Personal opinion: it's easy to assume you would prefer uptime over data consistency. But data inconsistency, e.g. a split brain, can be extremely painful.

  - Know how your setup works and what tradeoffs the business is comfortable with!

Suppose Fencing Fails…

- …but reports success.
- Now we have a split-brain!
  - (Not fun in production, but…fun for a presentation!)

Sidebar:

Why Should You Care?

Sidebar: Why Care?

- Even with all of the “right” tooling, the longer you run and the larger you grow, *something* (more than one thing) is going to bite you in production.
- It’s a good idea to think about potential failure modes in advance, and have an idea of how you might investigate and respond to various ones.
    - In the moment is *not* the time you want to be trying to find out the tools we’re using in this talk even exist!
- I’ve read postmortems of more than one high-profile incident.

We’ve split-brained; now what?

- First, we want to investigate what’s changed on each primary since the split.
- WAL encodes all changes, so how about:
  - Logical decoding? Nope, can’t replay.
  - pg_waldump/pg_xlogdump

pg_waldump

- Docs:
  - “display a human-readable rendering of the write-ahead log…”
  - “…is mainly useful for debugging or educational purposes.”
- Let’s try it out!

<terminal demo>

Investigating a Split Brain

- First we need to know the point in WAL where the two primaries diverged.

```
LOG:   received promote request
FATAL:   terminating walreceiver process due to administrator
         command
LOG:   invalid record length at 3583/A6D4B9A0: wanted 24, got 0
LOG:   redo done at 3583/A6D4B960
LOG:   last completed transaction was at log time
       2019-08-22 22:06:31.775485+00
LOG:   selected new timeline ID: 6
```

Investigating a Split Brain

- So we have two indexes into the WAL stream to guide us:
  - 3583/A6D4B960: Last successfully applied record from primary.
  - 3583/A6D4B9A0: First divergent record.

Sidebar: WAL Position Numbering

- A position in WAL is a 64-bit integer, but is printed as two 32-bit hex-encoded values separated by a slash, trimming more than one leading zero on the values.
  - E.g., 3583/A6D4B960 is really hex 00-00-35-83-A6-D4-B9-60

Sidebar: WAL Segment Numbering

- Postgres includes many functions for working with WAL positions and segment filenames to make this easier.
- A much more detailed explanation is available in this blog post:
  - http://eulerto.blogspot.com/2011/11/understanding-wal-nomenclature.html
- But as a quick summary…

Sidebar: WAL Segment Numbering

- WAL file segments are named on disk as a 24 character hex string; 8 for the timeline, 8 for the logical WAL file, and 8 for the offset within that logical WAL file.
  - E.g., WAL position 3583/A6D4B960 (assuming timeline 1) is in the WAL segment named 0000000100003583000000A6.
  - Note: watch out for dropped leading zeros when trying to figure this out!

Investigating a Split Brain

- First we have to have a split brain to investigate!
- Pretty simple to manually simulate:
  - Just promote a replica without fencing the existing primary.
- Let’s try it out!

<terminal demo>

Understanding the Divergence

- We can look at pg_waldump output and see the *kinds* of operations that have occurred since the divergence, but that output isn't overly helpful at the application or business domain level.
  - Exception: if there are no COMMIT records on one of the primaries after the divergence point, then we can conclude there is no *functional* divergence.
- But we really want to know *domain* impact. For example, we want know the tables (and ideally tuples values) changed on the divergent primary.

Understanding the Divergence

- So how do we determine *domain* impact?
  - Identify all transaction IDs that were committed after the divergence point.
  - Convert WAL operations into tuple data.
  - Manually investigate business impact/conflicts/etc.

Understanding the Divergence

- Identify all transaction IDs that were committed after the divergence point.
  - As simple as using grep, awk, and sed on pg_waldump output.

```
pg_waldump … |
    grep COMMIT | awk '{ print $8; }' | sed ’s/,//'
    > committed_txids.txt
```

Understanding the Divergence

- Convert WAL operations into tuple data.
  - First, dump relevant WAL. Consider this sequence of operations:

```
1. BEGIN;
2. INSERT …;
3. <split brain>
4. COMMIT;
```

  - Have to start far enough *before* the divergence point to include all transactions in flight at the divergence point.

Understanding the Divergence

- Convert WAL operations into tuple data.
  - Second, parse out txid, relfilenode, block, offset, (logical) operation type.
  - Additionally, while parsing fields, keep track of chain of ctids to find the most recent tuple. Consider this sequence of operations:

```
1. <split brain>
2. UPDATE … WHERE pk = 1;
3. UPDATE … WHERE pk = 1;
4. COMMIT;
```

  - We only need (and can only easily find) the last version of a given row.

Understanding the Divergence

- Convert WAL operations into tuple data.
    - Finally, we can use that information to query the diverging primary to find the actual data inserted or updated.
    - Unfortunately we can't easily figure out things that were deleted (unless it still exists on the original primary and we can find it there).
    - We also lose intermediate states of rows.
    - But even so we can get a reasonable view of activity post-divergence.

Understanding the Divergence

- Convert WAL operations into tuple data.
  - It all sounds intriguing, but how do we actually do it?
  - This is where the “and some custom scripting” in the abstract comes into play.
- Let’s try it out!

<terminal demo>

Understanding the Divergence

- Manually investigate business impact/conflicts/etc.
  - May want to investigate both primaries; whichever has the highest number of changes *might* be the one you want to keep around as the long-term primary.
  - This step is really up to you!

Restoring the Cluster

- Now that we’ve captured the information necessary to investigate the split brain, we want to bring the diverging node back into the cluster.
- Prior to PostgreSQL 9.5, we had to re-sync the data directory, much as if we were adding an entirely new node to the cluster. But that takes a long time with TB of data!
- Enter pg_rewind (added to PostgreSQL in version 9.5)!

pg_rewind

- Conceptually: according to the docs, resets the state of the data directory to the point* at which the divergence happened.
- Requirements:
  - Cluster was initialized with data checksums or has wal_log_hints on.
  - The replica to have all WAL (beginning before the divergence) available (or, if not directly, you can set a restore command to retrieve it).

pg_rewind: Details

- Copies all config files, so be careful to make sure they're correct!
- Resets data files to the divergence point *plus* any changes on the source primary to the same blocks.
- Therefore, by itself does not result in an immediately usable node.

pg_rewind: Details

- After "rewinding", the replica needs to stream/restore all of the *primary's* WAL beginning at the divergence point to be consistent.
- The WAL is part of syncing the data directory, so when PostgreSQL starts that WAL will be replayed.
    - But if you don't setup a recovery.conf first you'll be at a split brain again!
- Let's try it out!

<terminal demo>

Summary

- Your HA configuration should make split brains impossible.
- We used pg_waldump's (semi) human-readable output to diagnose what happened after a split brain.
- We used pg_rewind to restore the divergent node to a consistent replica state and reintroduced it to the cluster.

Q/A

Talk Materials: https://github.com/jcoleman/Split-Brain-Recovery