CYBERTEC
POSTGRESQL SERVICES & SUPPORT

PostgreSQL Replication: 20 Pitfalls and Solutions

Julian Markwort

pgconf.eu 2023

Introduction

Julian Markwort
Senior Database Consultant

CYBERTEC
POSTGRESQL SERVICES & SUPPORT

- PostgreSQL Consulting
- PostgreSQL Support
- PostgreSQL Remote DBA
- and more . . .

Motivation

- we consult on, deploy, and administrate a lot of PostgreSQL clusters
  - binary replication
  - logical replication
- we get confronted with replication problems regularly

PostgreSQL replication problems

PostgreSQL replication is not bad in itself

- usually it's errors by humans or automation
- often it's wrong assumptions and misunderstandings
- sometimes it's misleading or missing documentation
- seldomly it's bugs in PostgreSQL

Precursor

We've identified 20 common pitfalls

- we'll go through the pitfalls one by one
- we'll introduce any concepts required for understanding as needed
- we'll outline solutions for each

This talk is mostly aimed at raising awareness for these pitfalls, not about discussing them until everyone is bored.

WAL related Pitfalls

1 - WAL Recycling - outline

- PostgreSQL writes WAL to ensure crash recovery is possible
- crash recovery always starts at the latest `CHECKPOINT`
- so PostgreSQL can *recycle* all WAL that is older

1 - WAL Recycling - problem

- binary replication is just continuous recovery
- (continuous) recovery only works if there are no gaps in WAL
- if the primary recycles WAL, the replicas can’t use it any more to catch up

1 - WAL Recycling - solution

- use `wal_keep_size` setting
- use replication slots
- use archiving

2 - Archiving - outline

- WAL is split into 16MB *segment* files
- as soon as PostgreSQL is done writing to a WAL segment
  - it switches to a new one
  - it calls the `archive_command` (if `archive_mode` is enabled) on the old one
  - if that returns success, PostgreSQL can recycle the old file when it wants
- `archive_command` copies all WAL to a central location
- `restore_command` can be used by replicas to get WAL that the primary has already recycled

2 - Archiving - problem

- `archive_command` can fail
- `archive_command` can be too slow
- so PostgreSQL cannot clean up that file (and any subsequent ones)
- this can quickly lead to *out of disk space* situations

2 - Archiving - solution

- monitor your archiving (`SELECT * FROM pg_stat_archiver;`)
- make `pg_wal` mount large enough
- make sure you can increase your `pg_wal` mount in a hurry

3 - Replication Slots - outline

- replication slots can be created manually, by your failover tool, basebackup etc.
- replication slots track the replication process
  - the replica *advances* a `restart_lsn`
  - this LSN (*Logical Sequence Number*) identifies the point at which the replica could request WAL after it crashes, or loses the network connection

3 - Replication Slots - problem

- the primary needs to keep all WAL since that `restart_lsn`
  - even if there is no replica connected to advance it
- this can quickly lead to *out of disk space* situations

3 - Replication Slots - solution

- monitor your replications slots (`SELECT * FROM pg_replication_slots;`)
- use `max_slot_wal_keep_size`
- make `pg_wal` large enough

4 - Replication of Replication Slots - outline

- replication slots are not replicated

4 - Replication of Replication Slots - problem

- if your primary breaks and you promote a replica, that new primary doesn’t know anything about the slots on the old primary

4 - Replication of Replication Slots - solution

- use `permanent` replication slot feature in **patroni**
- use **pg_failover_slots** extension
- will perhaps be added to PG 17

5 - Parameter Dance - outline

- there are a handful of parameters that are used to allocate fixed memory for tracking things, such as running transactions

```
max_connections
max_locks_per_transactions
max_worker_processes
max_prepared_transactions
wal_level
track_commit_timestamp
```

5 - Parameter Dance - problem

- these need to be the same (or larger) on the replica, otherwise it can’t reconstruct all transactions
- starting a replica with too low settings will fail

5 - Parameter Dance - solution

- when increasing these values, you should increase them on the replicas first
- when decreasing these values, you should decrease them on the primary first

Switchover related Pitfalls

6 - Split Brain

- you should only have one primary in your clusters
- if you accept transactions on two primaries, you cannot *merge* their WAL
- always double check your old primary is down before promoting a replica
- analyze how your HA solutions handles this (it should use something like locking)

7 - Timeline Switches - outline

- Node A is primary, Node B is replica, both nodes are on Timeline (*TL*) 1

```
Node A  |  TL 1  |  1  2  3  4  5  6  7  *Node A crashes*
Node B  |  TL 1  |  1  2  3  4  *connection to Node A lost*
```

- **promote** Node B

```
Node B  |  TL 2  |              5  6  7  8  9
```

- Node A restarts

```
Node A  |  TL 1  |          ..  5  6  7
```

- need to throw away all conflicting data on node A (TL 1, records 5-7)

7 - Timeline Switches - problem

- PostgreSQL only has a REDO Transaction log
- can only move *forward*
- no way (using WAL alone) to undo those changes

7 - Timeline Switches - solution

- grab a new copy of the data directory from the primary
  - easy, foolproof, but expensive (IO, bandwidth)
- use pg_rewind
  - identifies *point of divergence*
  - rebuilds replica from primary by comparing WAL between the two
  - rebuilds only affected parts of table data files
  - at the end, the replica can start recovery at *point of divergence* and follow the timeline switch

8 - Switchover Implications for Autovacuum - outline

- some things are not replicated for performance reasons
- this includes the statistics collector (`pg_stat_user_tables` etc.)

8 - Switchover Implications for Autovacuum - problem

- these are things like usage counters, storing them durably and replicating them is too slow and not necessary for consistency
- `pg_stat_user_tables` and similar views are what **autovacuum** relies on to decide when it needs to run
- the same problem occurs even in standalone databases that do crash recovery

8 - Switchover Implications for Autovacuum - solution

- run `ANALYZE` after a switchover
  - at least on your tables that don't get picked up by autovacuum quickly
  - that mostly happens to very large tables with a comparatively small amount of regular inserts/updates/deletes
- monitor your autovacuum and table bloat!

9 - Transaction Loss after Failover - outline

- by default, PostgreSQL does not wait for replication feedback from replicas
- you can accidentially promote replicas that have not received all transactions

10 - Transaction Loss after Failover - solution

- make sure to only promote replicas that don’t lag too much
- you can use *synchronous replication*

10 - Synchronous Replication - outline

you turn on synchronous replication

- `COMMIT` latencies rise
  - deal with it
- you cannot `COMMIT` transactions when the replica is gone
  - add a second replica
- you do a failover and there is still a need for a rewind
  - did you lose any transactions?

10 - Synchronous Replication - problem

synchronous commit only waits for replicas to confirm `COMMIT` WAL records

- all other records are asynchronous
  - you can still lose some changes, just like you would with a single instance that crashes before you `COMMIT`ed

Read-Only-Replicas related Pitfalls

11 - Consistency when querying replicas - problem

- consistency across instances is sometimes weird.
- in asynchronous mode:
  - on a replica you can't see some data that was already committed on the primary
- in synchronous mode:
  - you can see data on one replica, but the primary is still waiting for feedback from other replicas

11 - Consistency when querying replicas - solution

- monitor replication lag
- don’t consider replicas healthy (for reading) if they have lag
- try to have your application “stick” to the same instance

12 - Vacuum and Replication Conflicts - outline

- replay can be blocked by open transactions on replicas
- there are no writes, so why can there be conflicts?
  - every transaction has a snapshot, that is used to ensure you can see the same versions of rows even if there are concurrent updates
  - there are snapshots on the replicas as well, of course

12 - Vacuum and Replication Conflicts - problem

- the primary regularly runs autovacuum and other maintenance tasks
- autovacuum wants to remove some row versions that can't be seen by anyone on the primary
- autovacuum's changes are of course written to WAL
  - replaying those changes on the replica conflicts with reads on the queries that *might* want to see the old versions

12 - Vacuum and Replication Conflicts - solution

- there's a tradeoff between
  - allowing transactions on the replica to finish
  - continuing with WAL replay
- this can be tweaked using `max_standby_streaming_delay`
  - how long can the replay process wait between receiving a change and applying it
  - default is 30 seconds
- does *not* refer directly to the duration of conflicting transactions

12 - Vacuum and Replication Conflicts - solution

- don't use replicas that can delay replay indefinitely as candidates for switchover
  - they would need to replay all the transactions when asked to promote.

12 - Vacuum and Replication Conflicts - solution

- you can also configure the replica to inform the primary about which snapshots it still needs to see: `hot_standby_feedback`
  - this means that autovacuum progress is slowed down on the primary

13 - Prepared Transactions and Recovery

- prepared transactions are WAL-logged
  - survive recovery and thus switchovers
  - ensure your transaction manager can manage this
- there *was* a bug related to recovery in hot_standby with prepared transactions in PG 13 and 14

14 - Hot Standby doesn't work - outline

a replica in hot standby needs to know which transactions are currently in flight on the primary to know what data it can show to reading queries

- the primary writes a `XLOG_RUNNING_XACTS` record into WAL regularly
- replicas can serve queries when they have seen such a record since their *Minimum recovery ending location* (`pg_controldata`)

14 - Hot Standby doesn't work - problem

- all instances in a cluster crash
- you start them as replicas, wait until they allow connections
  - they might have no `XLOG_RUNNING_XACTS` record in WAL after *Minimum recovery ending location*
  - they will not open up for reading connections and wait indefinitely

14 - Hot Standby doesn’t work - solution

you need to choose a replica (ideally the one with most transactions) and promote it manually

15 - Hot Standby doesn't work - bonus problem

a `XLOG_RUNNING_XACTS` record only has limited space for subtransactions

- if you have too many subtransactions, this record cannot keep track of all of them
- `XLOG_RUNNING_XACTS` record will have it's `suboverflowed` flag set
- we cannot go into hot standby if we don't know all (sub-) transactions in flight

15 - Hot Standby doesn't work - bonus solution

- don't use any subtransactions (there are known performance problems)
- don't use too many subtransactions
  - don't use `SAVEPOINT` like it's free
  - don't use `PL/pgSQL` exception blocks like it's free
- don't have long running transactions

Logical Replication Related Pitfalls

16 - Logical Replication Conflicts - problem

- a `subscriber` is just a regular database that needs to run as a primary
- this means there is no straigh-forward mechanism that prevents you from writing to your `subscriber`
- there is no conflict resolution in *in-core* logical replication

16 - Logical Replication Conflicts - solution

- ensure nobody writes to your `subscriber`, e.g. by using different roles with only `SELECT` privileges

17 - DDL trouble - problem

- logical replication relies on the table schema being the same
- you can work around some differences (depends on the PG version)
- DDL is not replicated at all

17 - DDL trouble - solution

- don’t change your schema at all
- if you must change it, do it in a way that will not block replay on the `subscriber`

18 - long running Transactions and Snapshots - outline

when creating a subscriber, you usually start with an existing table

- so you need to copy the table contents
- logical replication can do this for you
- you don't want to miss any transactions between start and end of the copying

18 - long running Transactions and Snapshots - outline

- the initial table copy worker has to create a snapshot
- it needs to wait for all previous transactions to finish
- and it needs to keep track of all transactions created in the meantime

18 - long running Transactions and Snapshots - problem

- this snapshot can grow too large
  - if there are long running transactions, and lots of short transactions
- then the subscriber must start over again
  - and will likely fail again if there are still long running transactions

18 - long running Transactions and Snapshots - solution

Don’t allow long running transactions

19 - max_replication_slots and table sync workers - outline

When adding a subscription for a lot of tables, there will usually be multiple `table sync` workers

- the number is configurable using `max_sync_workers_per_subscription`
- the subscription creates a replication slot
- every sync worker creates a replication slot
- so you need at least 1 + `max_sync_workers_per_subscription` slots on the publisher
- you also need at least as many `max_logical_replication_workers` and `max_worker_processes`

19 - max_replication_slots and table sync workers - problem

how many slots do you need on the subscriber?

- there will be no replication slots created on the subscriber
- but `max_replication_slots` is used to size an array
- this array holds the state for every table sync job
  - this is not cleaned up quickly enough in some cases (especiall when there are lots of small tables)

19 - max_replication_slots and table sync workers - solution

- you need as many `max_replication_slots` on the subscriber as you have tables to sync, if you want to be on the safe side

20 - long running transactions and apply (PG < 14) - problem

before PG 14, the output plugin was only able to send out complete transactions

- the output plugin tracked all changes of that transaction in memory
- if that was too much this was stored on disk
- when the transaction finally commits, all of the changes are sent to the subscriber
  - high potential for replication lag

20 - long running transactions and apply (PG < 14) - solution

since PG 14, the changes can be “streamed” to the subscriber

- now the subscriber is in charge of reassembling the whole transaction
- this can even be done in parallel in PG 16

Conclusion

Conclusion

don’t use long running transactions!

Thank You

Archiving after pg_upgrade

- pg_upgrade requires creation of a new (empty) data directory
- this means the LSN counter starts over
  - same goes for the timeline
- you need to switch to a new archive after the pg_upgrade
  - otherwise there can be conflicts when archiving (file already exists with different contents)
  - or there could be problems when trying to do Point in Time Recovery

Timeline Switches - bonus problem

- pg_rewind doesn't only make sure that table data files match the primary
- it copies config files and directories it cannot rebuild by comparing the WAL
  - this frequently includes the `log` directory
  - so all logs from the old primary before the failover are easily lost
- don't store logs in the data dir

Timeline Switches - bonus bonus problem

- Node A, B and C are on TL 1
- A fails
  - you promote B to TL 2
- B fails
  - you promote C to TL 3
- C fails
- B restarts (as primary), is still on TL 2
- B lives happily ever after

There is now an *abandoned* TL 3 in your archive.

- next time you do a PITR and tell recovery to find the newest timeline, it will try (and most likely fail) to go to TL 3.
- this can also happen in streaming replication setups
- make sure you know how to manually do PITR and change the `recovery_target_timeline` to something other than `latest`

Switchover Implications - bonus problem

- memory allocations are not replicated (`shared_buffers`)

Switchover Implications - bonus solution

- run `pg_prewarm` after a switchover to quickly get your important tables into `shared_buffers`
- monitor buffer hit rates for critical tables

Synchronous Replication - expectation management

When you send a `COMMIT` **request** to the database, this happens:

1. the transaction is finished (concurrency contol, triggers, constraint checks)
2. the `COMMIT` record is written to WAL
3. the record is sent to replicas
4. the replicas process the record, store it in their WAL and flush that
5. the replicas send feedback to the primary
6. once the primary has *enough* feedback, it can send the `COMMIT` **reply**

When you received the `COMMIT` **reply**, you can assume that the transaction is *durable* on primary and replicas.