PostgreSQL Replication

Christophe Pettus
PostgreSQL Experts
PerconaLive, April 25, 2018

Christophe Pettus

CEO, PostgreSQL Experts, Inc.

christophe.pettus@pgexperts.com

thebuild.com

twitter @xof

Questions Welcome!

"It's more of a comment…"

"It's more of a comment..."

EXIT

Replication Options.

- WAL shipping.
- Streaming replication.
- Trigger-based replication.
- Logical decoding-based replication.
- And exotic animals.

The Write-Ahead Log.

- A continuous stream of database changes written to local storage.
- Consists of a series of records, each a specific change to a specific underlying disk-level item.
  - “Update field 3 of ctid (12312,6) of relation OID 918123 to ‘cat’.”
- Written to disk as a series of 16MB “segments,” with large unpleasant hexadecimal names, like:

  `00000001000002A00000065`

Crash Recovery.

- The WAL was originally designed to provide crash recovery.

- On a startup, PostgreSQL goes back to the WAL segment that contains the last checkpoint, and starts replaying the activity in it.

- Once it reaches a consistent state again, the database can start up and receive connections.

Hm.

- The system doing the recovery doesn’t have to be the one that crashed, of course.
- So… what if the original system was still in operation?
- We could replay the WAL segments on a different system, and keep it up to date with the primary?
- And, voilà, WAL shipping was born (in version 8.0).

WAL Shipping.

- When the primary completes a WAL segment, it runs archive_command.
- archive_command can do anything, but it usually copies the file over to the secondary (or to a place the secondary can get at it).
- The secondary repeatedly runs restore_command (in recovery.conf) to replay the WAL information.

WAL Shipping: The Good.

- Cheap and cheerful to set up: You just need to be able to copy files around.
- Works well on slow or unreliable networks, like WANs, since no persistent connection is required.
- You can use it as a basis for point-in-time recovery, if you keep WAL information and base backups around to handle it.
- Works on really old versions of PostgreSQL that you shouldn’t be running anymore.

WAL Shipping: More Good.

- DDL changes in PostgreSQL are WAL-logged, so…
- … they're pushed down to secondaries automatically.
- The secondary is a perfect mirror (allowing for replication lag) of the primary.
- The secondary is readable (if set up right) for read-only queries.
- Failover is as easy as just promoting the secondary and letting it come back up; it takes <1 minute, usually.

WAL Shipping: The Bad.

- Secondary is only as up-to-date as the last 16MB WAL segment: You can lose some changes.
- WAL segments have to be managed lest you run out of disk space.
- Replicating to multiple secondaries requires some complex orchestration.
- The secondary cannot be written, at all, including temporary tables and materialized views.

WAL Shipping: More Bad.

- Since the WAL is a global resource across all databases in the PostgreSQL server, you cannot pick and choose anything.
- You must replicate all fields in all columns in all tables in all the databases.
- You cannot consolidate multiple servers into one using WAL shipping.
- Cannot replicate between major versions of PostgreSQL, so can’t use it for zero-downtime upgrades.

Streaming Replication.

- Well, what if we didn’t just ship files, but transmitted the WAL information down a network connection?
- The secondary could stay much “closer” to the primary.
- And that’s what streaming replication is: The same (pretty much) WAL information, only transmitted down to the secondary.

Streaming Replication: The Basics.

- recovery.conf is used to “point” the secondary at the primary.
- The secondary connects to the primary, and receives a stream of the WAL information.
- Otherwise, largely the same as WAL shipping, with the same limitations and benefits.

Stream Replication: The Good.

- The secondary stays close to the primary, in terms of transaction activity.
- With (optional) synchronous replication, the chance of a lost transaction (committed on the primary but not the secondary) is essentially zero.
- Replicas can cascade for more complex topologies.

WAL-Based Replication Weirdnesses.

Replication Delay.

- When a WAL change to the data in a relation comes into a secondary, and that secondary is running a query that uses that relation, what should we do?
- If we applied the change “under” the query, the result could be wrong.
- Option 1: Delay applying the change until the query completes.
- Option 2: Cancel the query.

max_standby_*_delay

- Two parameters (one for streaming, one for WAL shipping) that control how long to wait before cancelling the query.
- Higher settings mean more potential replication lag.
- Advice: Dedicate a server for failover with these set to 0, and other servers for read-only traffic with higher values.

hot_standby_feedback

- If “on”, sends feedback upstream telling the primary what tables are being queried on the secondary.
- The primary will then defer vacuuming those to avoid query cancellations on the secondary.
- This can result in table bloat, if there’s enough query traffic on the secondary.
- It does not completely eliminate query cancellations.
- In general, it’s a good idea, but monitor bloat.

vacuum_defer_cleanup_age

- Don’t bother.

Trigger-Based Replication.

Trigger-Based Replication.

- WAL-based replication has a lot of restrictions.
  - No selectivity on replication, same major version, etc.
- But PostgreSQL has a very elaborate trigger mechanism!
- What if we attached a trigger to each table, caught update / insert / delete operations, and pushed them to the secondary that way?

Why, yes, we could do that.

- Actually predated WAL-based replication, in the form of Slony 1.
- Now we have:
  - Slony (C)
  - Londiste (Python)
  - Bucardo (Perl)
- … plus some others that basically work the same way.

Triggers: The Good.

- Much more flexible than WAL-based replication.
- Depending on the particular package, can:
  - Replicate only some databases.
  - Replicate only some tables.
  - Replicate only some fields.
  - Filter changes based on rules on the primary before sending them over.

Triggers: More Good.

- Can build exotic topologies.
- Can consolidate multiple databases into a single database (for data warehousing, etc.).
- Bucardo (only) does multi-master replication.
- Works between different PostgreSQL versions, so can use them for zero-downtime upgrading.

Triggers: The Bad.

- Tedious and fiddly to set up.
- Every table that is going to be replicated needs a primary key (at least a de facto one).
- Initial copies can take a long time.
- Awkward fit with WAL-based replication for failover.
- All those triggers firing all the time and the log tables required have a performance impact.
- No automatic DDL change distribution: That’s on you.

Comparison

- Slony tends to be the highest-performance of the lot.
  - … but requires C-language extensions.
- Londiste requires PL/PythonU availability.
- Bucardo can work entirely outside the subscriber (but not provider) system, thus suitable for RDS.
- Bucardo also supports multi-master and primary key updates.

Triggers: Advice.

- If you can use more modern logical decoding-based replication, use that instead.
- Still useful for major version upgrades, when the old version <9.4.
- Sometimes required for specialized environments where you don’t have access to built-in logical replication or the WAL stream (in specific, RDS).

Logical Decoding.

Logical Decoding.

- First introduced in PostgreSQL 9.4.
- It’s not a packaged system like streaming replication; it’s a framework for implementing logical replication and other fun things.
- Really required 9.6+ to get going.

How It Works.

- The framework turns WAL records back into SQL-type operations.
- “Update field 3 of ctid (12312,6) of relation OID 918123 to ‘cat’” becomes “UPDATE menagerie SET animal_type=‘cat’ WHERE ctid=‘(12312,6)’” (to a first approximation).
- Doesn’t reconstruct the actual SQL that made the change, or build actual SQL strings.

Replication Slots.

- A logical replication slot is a named database object that “captures” the WAL stream.
- Once created, the framework delivers the decoded WAL stream to the slot’s specified plug-in, which can do whatever it wants with it.
- The plug-in reports back to the framework when it has processed the WAL stream, so that the local WAL segments can be recycled.

Replication Slots, *The Horrible Truth*

- A replication slot keeps track of the WAL position of its consumer (in the case of logical replication, the plug-in).
- If the consumer stops consuming, the framework retains WAL information so it can catch up.
- This results in WAL segments not being recycled.
- So you can run yourself out disk space.
- So, monitor your disk space already!

Replication Plug-Ins.

- A replication plugin is a bit of C code installed in the primary server (like any extension) that receives the stream of decoding WAL records.
- It can do anything it wants with them: logging, auditing, feeding to an external data system, etc.
  - https://github.com/confluentinc/bottledwater-pg
    Logical replication into Kafka!
- PostgreSQL ships with a test plugin that provides example code and logging, but it’s not useful for any actual production use.

PostgreSQL-to-PostgreSQL Logical Replication Options.

- On PostgreSQL 10+, built-in logical replication.
- On PostgreSQL 9.4+, pglogical.
    - https://www.2ndquadrant.com/en/resources/pglogical/

The High-Level View.

- Each takes the stream of decoded changes, applies them at the SQL level.
- This means (most) constraints are enforced, rows are locked, triggers (can) fire, MVCC happens, etc.
- A database can be both a publisher of changes and a subscriber to changes.
  - A single table can be both a source and target.
  - A single table cannot replicate bidirectionally, however.

General Setup.

- Use pg_dump —schema-only to copy the schema over to the subscriber node.
- When it first connects, can do an initial bulk copy of the existing data, followed by replicating data going forward.
- DDL changes are not propagated (pglogical provides a function to run DDL changes on each node; in-core leaves it to you).

Row Identity

- All tables that are to be replicated should have some kind of row identity.
- Ideally, a primary key or a UNIQUE index.
- pglogical requires either a PK or a single UNIQUE index.
- In-core logical replication can use the entire row value as to identify the row if all else fails.

Sequences.

- Sequence values are not replicated (row values set off of a sequence are, of course).
  - pglogical can replicate them in batch as a background process.
- If consolidating to a single table, use disjoint ranges from the source databases (or non-sequence keys).
  - UUIDs! UUIDs! UUIDs!
- Logical replicas are generally not suitable for failover due to this restriction.

TRUNCATE

- In-core logical replication does not replicate TRUNCATE at all (at present).
- pglogical replicates TRUNCATE, but does not cascade TRUNCATE CASCADEs.

Reality Check.

- You can only replicate a “real” table to a “real” table.
- So, no materialized views, views, foreign tables, or partition root tables.
- If you are using PostgreSQL 10 partitioning, the root table is not a real table, so cannot participate in logical replication (either source or destination).
- Old-style partitioning should still be possible with ENABLE ALWAYS triggers (unverified at press time).

Uses the WAL, so…

- Cannot replicate temporary or unlogged tables.
- COPY operations are broken into individual INSERTs.
- Individual statements are “unrolled”.
  - A single UPDATE changing 10,000 rows will be applied as 10,000 UPDATEs.

Compare and Contrast

- pglogical has several features in-core replication does not.
    - Flexible conflict handling, row/column filtering, sequence replication, etc.
- pglogical requires an extension to be built and installed; not part of the core distribution.
- pglogical is operated by functions; in-core replication uses SQL statements.

Battle of the Replications

- Only a primary node can be a logical replication publisher or subscriber.
- If a primary with logical subscribers fails over to a secondary, the current logical replication state is not passed over to the secondary.
- So, synchronization problems can happen.
    - Changes on the streaming secondary that have not been pushed down to the logical subscribers, for example.
- PostgreSQL 11 should address this.

On Amazon RDS?

- Pre version 10, your logical replication options are… limited.
- RDS supports a somewhat quirky set of logical decoding plugins.
- No general table-to-table replication at the moment.
- PostgreSQL 10 is now available, and supports in-core logical replication.

Exotic Animals.

pgpool2 statement-based replication.

- pgpool2 can “split” the incoming query stream between two servers.
- Thus, all operations are applied to both.
- Please do not use this feature.

Amazon DMS.

- On PostgreSQL, based on logical decoding.
- Primarily designed for migration between different database system types.
- Does not support some important PostgreSQL types (like TIMESTAMPTZ).
    - Thus, not really useful for PostgreSQL-to-PostgreSQL replication.

2nd Quadrant BDR.

- Shares many similarities to pglogical.
- Currently, a closed-source proprietary product.
    - 2Q indicated it will be open-source in the future.
- Can do bidirectional (i.e., multi-master) replication.

Other commercial solutions.

- Lots of other commercial solutions.
- Pretty much all trigger-based.
- Generally most useful as packaged solution for between-database-product migrations.

In Sum!

Advice?

- For failover, use streaming replication.
- For read-only queries, use streaming replicas that are not dedicated to failover.
- If you need logical replication:
  - Use in-core logical replication unless you need a pglogical feature.

Thank you!

Questions?

Christophe Pettus

CEO, PostgreSQL Experts, Inc.

christophe.pettus@pgexperts.com

thebuild.com

twitter @xof

PGX

POSTGRESQL EXPERTS, INC.

pgexperts.com