Real-World Logical Replication

Christophe Pettus

Nordic PGDay 2023

PGX
pgexperts.com

An instant history of PostgreSQL replication.

- 2001: The Write-Ahead Log (version 7.1).
- 2004: Trigger-based replication.
- 2005: WAL archiving (version 8.0).
- 2010: Streaming replication (version 9.0).
- **2014: Logical decoding (version 9.4).**
- **2017: Native logical replication (version 10).**

PGX
pgexperts.com

Everything starts with the WAL.

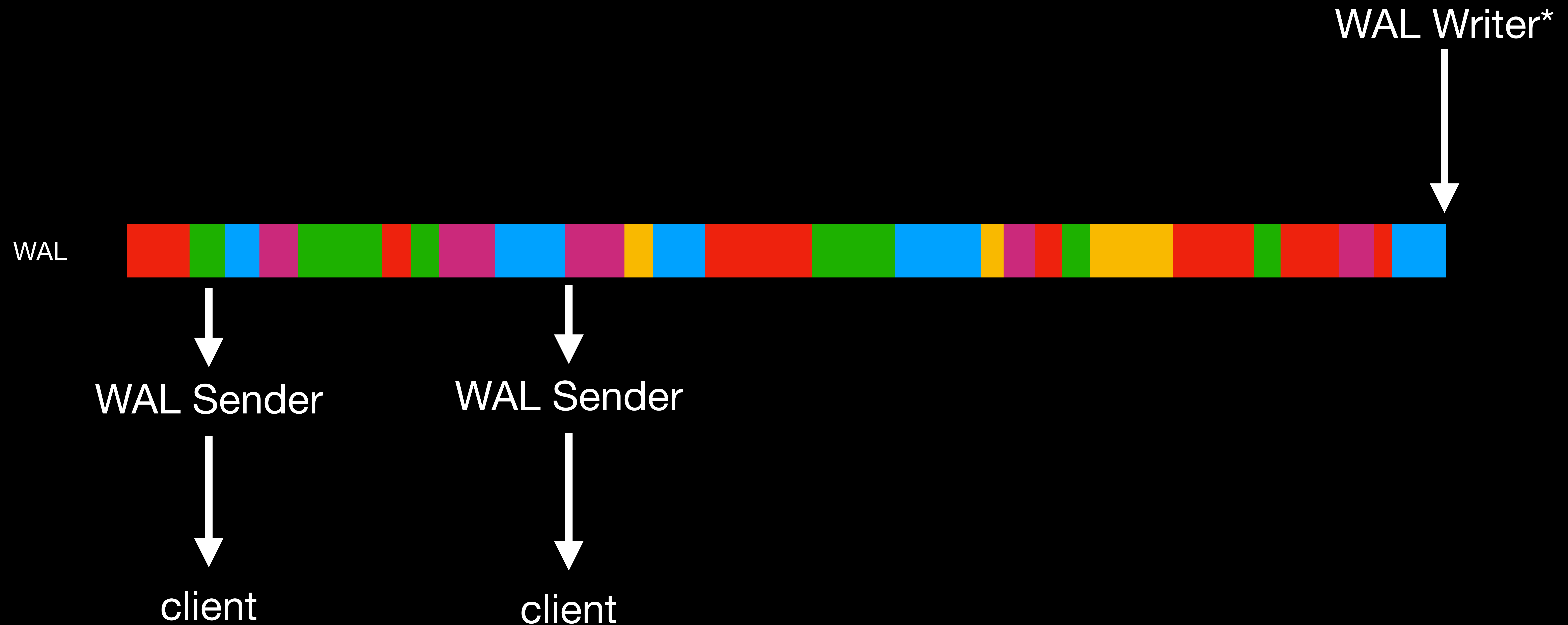
- The PostgreSQL write-ahead log captures (nearly) every change to the database.
- A cluster-wide, continuous stream of records, indexed by Log Sequence Numbers (LSNs).
    - LSNs are a 64-bit byte index into the WAL stream.
- Maintained as a series of 16MB files on disk.
- Originally for crash recovery, now supports all kinds of other features.

PGX
pgexperts.com

Binary Replication.

- The WAL is transmitted to another server, which replays it.
- Based on crash recovery, and still called “recovery” to this day.
- Can either send individual WAL segments (WAL archiving) or transmit the WAL directly over the network (streaming replication).
  - The results are the same between the two methods.

Binary Replication.



Binary Replication, the Good.

- The replica is an exact binary copy of the primary.
- This includes everything: DML, DDL, sequences, all kinds of stuff.
  - Except unlogged and temporary tables.
- Great for failure of the primary database: just switch over!
- Cheap ’n’ cheerful: Easy to understand, easy to set up.

Binary Replication, the Bad.

- The replica is an exact binary copy of the primary.
- This means the same databases, the same schema, the same indexes, the same everything.
    - Binary replicas cannot be written to by anything but the incoming replication stream.
- Query cancellations can make it hard to use them for substantial read load.
- Primary/replica must be same PostgreSQL major version, so can't be used for upgrades.

Along comes logical decoding.

- First released in version 9.4.
- Still based on the WAL.
- It is not logical replication: it’s a framework for building change capture tools, one of which might be logical replication.
- pglogical (still around) was the first logical replication system built on top of it.

Logical decoding, a quick overview.

- It does not capture SQL.
- Still based on the WAL.
- Reads the WAL, and transforms it into a series of events:
  - BEGIN / COMMIT
  - INSERT, UPDATE, DELETE
  - TRUNCATE (since version 11)
  - And others.

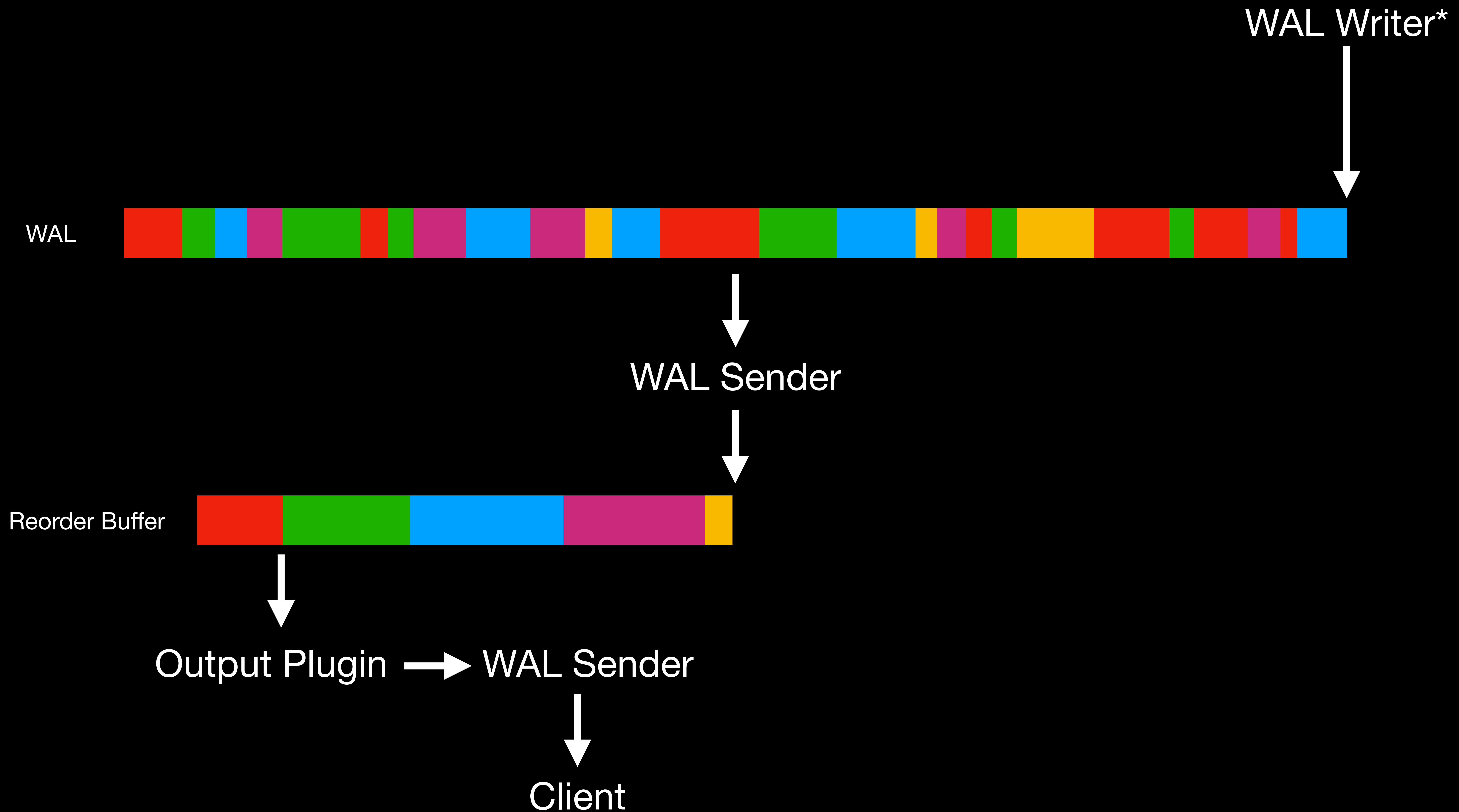
PGX
pgexperts.com

Logical decoding, a quick overview.

- The framework then presents these to an output plugin.
  - A piece of C code that runs as part of the PostgreSQL server.
- The output plugin accepts the events as calls to functions, and produces… stuff.
  - Logical decoding doesn’t care what the stuff is.
- That stuff is then sent to consumers of it by the server (not by the plugin directly).

PGX
pgexperts.com

Logical Decoding.



DID I MENTION IT'S NOT SQL?

- The calls to the output plugin are analogous to SQL, but they're not the original SQL.
  - Operations that affect multiple rows generate one event per row.
- There are no predicates: The event specifies the row that it operated on, not how it found that row.
- There are no virtual tuples: Everything that the output plugin gets is a real tuple really written to real storage.
  - But they *are* identified by column values, not CTIDs.

Some things aren't decoded.

- Either because of current framework limitations, or because they would be superfluous.
  - TRUNCATE wasn't replicated until version 11.
  - DDL is not (currently) replicated.
  - The current values of sequences are not replicated (the values that are stored in tuples from sequences are, of course).
- Index changes are not decoded.
- If it's not in the log (unlogged/temporary tables), it's not decoded.

Transactions and the single output plugin.

- Each output plugin receives all of the changes for a single transaction, in order.
- The transactions are ordered by COMMIT sequence.
- In the original API, only committed transactions are presented to the output plugin. (One change at a time.)
- In the newer (version 14) streaming API, it's possible that a transaction might be aborted even after the plugin has started getting its changes.
- Generally, these details only matter if you are writing an output plugin.

Now that we have that, what can we do?

- Change capture!
- Change capture enables a lot of cool things.
- And that's what this presentation is about!
- Logical replication, event pipelines, upgrades, repartitioning, data type changes, event notification systems…
- First, let's talk about some example output plugins.

The Test Plugin.

- Ships as part of the PostgreSQL distribution:
  - contrib/test_decoding
- Just outputs a series of text messages representing the change events.
- Intended for educational and entertainment purposes only.
- Didn’t stop Amazon from building the PostgreSQL side of their Data Migration Service on it!

wal2json

- https://github.com/eulerto/wal2json
- Outputs a JSON object for each committed transaction.
- Consumers can read it for a continuous stream of changes.
- Big transactions == big JSON objects, so be careful!
- Very easy to write a consumer (the Python psycopg2 library has an example in its documentation).

pgoutput

- Part the core, you just get it when you install PostgreSQL.
- Transforms the event stream into a pgoutput-specific (but documented) message stream.
- The consumer can be your own code, or another PostgreSQL server.
- And when the consumer is another PostgreSQL server, we call that **logical replication!**

Logical Replication is…

- Logical decoding change capture used to send changes from one PostgreSQL instance to another.
- Based around a **publication**: a set of tables whose changes you want to send.
- And **subscriptions**: consumers who read those changes and apply them to their local tables.
- Unlike binary replication, publications and subscriptions are local to a particular database, not an entire cluster.

Slot Machine.

- The connection between a particular logical decoding producer and consumer is a replication slot.
    - Replication slots exist for binary replication too, but are slightly different.
- Replication slots are persistent objects on the producer, and are created with a specific output plugin.
- The state of the logical decoding (such as the point in the WAL has been sent and acknowledged) is stored in the slot.
- The slot can be created at consumer connect time, or in advance (if in advance, referred to by name).

Fun with Publications.

- A publication specifies a set of tables to replicate, FOR TABLES IN SCHEMA, or ALL TABLES. If TABLES IN SCHEMA / ALL TABLES, new tables are automatically added to the publication.
- If specific tables are named, a subset of the columns can be published.
- The changes start being decoded when the first subscriber connects.
- Optionally, the state of the tables before the decoding started can be streamed over to the subscriber.
  - This uses a separate set of workers, and is based on COPY statements (not logical decoding).

PGX
pgexperts.com

Great, but what's all this good for?

- Upgrades.
- Changing data types.
- Repartitioning.
- Data warehousing/aggregation.
- Data pipelines/event streams.

Upgrades.

- The publisher and subscriber in logical replication do not have to be the same PostgreSQL major version.
- The publisher can continue to operate normally while logical replication is going on.
- Can do a major version upgrade with minimal downtime.
  - How long depends on how fast the application can be re-homed.

The steps…

- Create a new cluster (including binary replicas).
- Use pg_dump —schema-only to copy an empty schema to the new cluster.
- Publish all the tables in the current cluster.
- Subscribe all the tables in the new cluster.
- Let the initial copy complete.
- Stop the application on the old cluster.
- Wait for replication to catch up.
- Bring up the application on the new cluster.
- Profit!

So. Many. Gotchas.

- There are a lot of small details that can catch you here.
- The good news is that if you get something wrong, just tear down the subscriber cluster and start over.
- The bad news is that may take a long time.

Preparation on the Publisher.

- Logical decoding can only work on a table that has a REPLICA IDENTITY.
- This usually means “has a primary key” or “has a non-partial, non-NULL unique index.”
- If it doesn’t, add one.
- If every row is unique, you can use REPLICA IDENTITY FULL.
- Otherwise, you’ll need to make some decisions about how to get those table(s) over.

Preparation on the Subscriber.

- That initial copy can take a long time.
- Drop indexes and constraints on the subscriber, and recreate them later.
- Since the only data the subscriber is getting is from the publisher, constraints have already been enforced.

Publisher Settings.

- wal_level = logical
- max_replication_slots = subscribers * (1 + synchronization workers)
  - See Subscriber Settings for synchronization workers.
- max_wal_senders = max_replication_slots + senders for binary replication.

PGX
pgexperts.com

Subscriber Settings.

- max_replication_slots = number of subscriptions.
- max_sync_workers_per_subscription = For parallel workers during initial sync. Generally number of cores / number of subscriptions is reasonable (although >8 seems to be diminishing returns).
- max_logical_replication_workers = max_replication_slots * (1 + max_sync_workers_per_subscription)
- max_worker_processes = max_logical_replication_workers + everything else.
- https://thebuild.com/blog/2023/02/28/workers-of-the-world-unite/

A note on the initial copy.

- The initial copy can be a long process.
- While the copy is going on, WAL segments will be retained on the publisher.
  - To be processed when the copy is done.
- This can be a lot of WAL.
- Plan for the disk space consumption.

Other initial copy techniques.

- Each synchronization worker can only do one table at a time.
- Breaking up the database into multiple publications / subscriptions can speed things up.
- Remember to create enough worker processes to handle all this!
- If you drop constraints, you don’t need to worry about consistency until the very end.
- If you do not drop constraints, be aware of foreign key relationships between tables in different publications.

A word of caution.

- You may in the course of Googling find a blog post suggesting using file system snapshots instead of the initial copy to save time. This process involves cutting and pasting LSNs out of the server text logs into WAL-control functions.
- Cutting and pasting LSNs out of text logs into WAL-control functions is not recommended.
- Thank you for your cooperation.

The biggest gotcha.

- DDL is not decoded, which means it is not replicated.
  - The DDL-meets-DML TRUNCATE is an exception, version 11+.
- Schema changes must be applied in strict order, in a way that is upwards compatible.
- Generally this means adding columns as NULLable, to the subscribers first.
- Remember to refresh the publication!

Other gotchas.

- If you are coming from PostgreSQL version 10, TRUNCATE is not decoded and sent over.
- So, any TRUNCATEs on the publisher will not happen on the replica.
- Remember that sequences are not replicated at all, so you'll need to patch those up at the end during switchover.

Showtime!

- Once the initial copies are done, and any indexes / constraints are rebuilt, you can cut over.
- Stop the application. No, really, stop the application. If you don’t, you risk data loss.
- Wait for replication to finish sending over changes (pg_stat_replication is the view to monitor).
- Sync sequences.
- Rehome the application.
- You’re done.

Failback.

- pg_upgrade doesn't have a failback mechanism except "restore from backup."
- Using logical replication, you can set up a failback plan:
  - Once the replication from old -> new is done, tear down its publications / slots.
  - Set up reverse replication new -> old before rehoming the application.
- This allows a no- (or at least minimal-) data loss failback to the old database if required.

While you have the hood open…

- You can use this same method to change data types.
  - int keys to bigint keys are a very good example.
  - Just make sure the old and new types are compatible.
- You can use this method to partition tables.
  - The subscribed table can be partitioned where the published table is not.
  - You can also use this to re-partition an existing partitioned table.

Migration between systems.

- The upgrade process can be used to migrate between instances.
- Handy for cloud vendors that don’t let you easily shrink instances!
- Can also be used to do migrations not possible with binary replication:
  - Different processor architectures.
  - Different collations.
  - Turning on data checksums.

PGX
pgexperts.com

Analytics Databases.

- Binary replication interacts badly with long-running read queries.
  - Query cancellations vs replication lag is an inescapable push-pull.
- Many useful things (temporary tables, different indexes, materialized views) are not available on binary replicas.
- Logical replication to the rescue!

Analytics Databases.

- Logical replicas don't have the same query cancellation behavior as binary replicas.
  - Just standard MVCC visibility on new tuples in long-running transactions.
- Logical replicas can have roll-up tables, temporary tables, indexes, views that aren't present on the publisher.
- Logical replication lag will run higher than binary replication lag, but is more predictable in the presence of a high query load.

Data Consolidation.

- A single subscription can connect to multiple publications.
- Data from multiple source tables can thus flow into a single destination table.
- It’s up to you to handle primary key issues here!

Cross-product replication.

- Logical replication allows PostgreSQL to send changes to non-PostgreSQL RDBMs.
- Like when the corporation is an Oracle shop, but you (sensibly) use PostgreSQL in your area.
- Mostly the domain of commercial products.
  - But: https://www.symmetricds.org

You Can't Do These.

- Multi-master using in-core logical replication.
  - There are a bunch of solutions out there, of varying degrees of maturity.
- Use a logical replica as a failover candidate.
  - Too many ways to lose data here.
- Replicate a table within a single database.
  - Fully-qualified name is used to identify tables.

Big huge gigantic currently-unsolved issue.

- Replication slots are not replicated to binary replicas.
- This means that the state of the logical replication slot is lost on failover to a binary replica.
- This is bad.
- Patroni has implemented a solution:
  - https://www.percona.com/blog/how-patroni-addresses-the-problem-of-the-logical-replication-slot-failover-in-a-postgresql-cluster/
- … but it's not in core (yet?).

Logical decoding that’s not replication.

- Logical decoding can do a lot of stuff that’s not just replication.
- This generally requires…
  - … a custom output plugin, or,
  - … a consumer that understands the pgoutput protocol.

Data pipelines.

- Logical decoding can feed the data pipeline of your choice.
  - Open source solutions: Debezium, etc.
  - Proprietary solutions: Fivetran, Amazon DMS, etc.
- Trigger events on particular database changes.
- Insert data into non-PostgreSQL data sources.

REPLICA IDENTITY and its discontents.

- In order to logically decode changes to a table, it needs a proper REPLICA IDENTITY.
  - INSERTs don't require a REPLICA IDENTITY, if that's all you're doing.
- If all else fails, REPLICA IDENTITY FULL uses the entire row (including TOASTed objects) as the identity.
- This is generally not very efficient, but it works.

Abusing REPLICA IDENTITY.

- The output plugin receives as part of the change message on UPDATE or DELETE:
  - The replica identity (so it can tell what the row is).
  - The changed columns.
- It doesn't receive the old values of the columns unless they appear in the REPLICA IDENTITY.
- Temptation: Just set everything to REPLICA IDENTITY FULL so your change capture gets both old and new values!
- This is a hack, but it does work. However, consider the cost before doing something you'll regret.

WAL messages.

- The best logical decoding feature you've never heard of!
  - Introduced all the way back in 9.6.
- Allows applications to write *custom messages* into the WAL.
- These are presented to logical decoding plugins as-is.
- Messages can be transactional (only decode on COMMIT) or non-transactional (always decode).

PGX
pgexperts.com

WAL messages.

- Messages consist of a prefix and a payload.
- The prefix is usually the name of the plugin; all plugins get the message and decide if it’s for them.
- The payload can be text (which could be text-form JSON) or binary.
- With a little coding, these can be used to send messages into data pipelines.
    - https://www.infoq.com/articles/wonders-of-postgres-logical-decoding-messages/

THANK YOU!

QUESTIONS?

christophe.pettus@pgexperts.com

Twitter @xof

Mastodon @cep@fosstodon.org

thebuild.com

PGX

pgexperts.com

PGX

pgexperts.com