A lesser known superpower of Postgres Logical Replication: Protocol Versions

Kevin Biju, Engineering @ClickHouse

Introduction

https://github.com/PeerDB-io/peerdb

PeerDB-io / **peerdb** ★ 2.4k

ClickHouse / **ClickHouse** ★ 39.3k

+

ClickHouse acquires PeerDB to boost real-time analytics with Postgres CDC integration

ClickHouse

Overview of Postgres replication methods

- Postgres relies on WAL (write-ahead log) for transaction durability; all changes (even uncommitted ones!) are written to disk as part of a WAL file
- Log-shipping moves entire WAL files from the primary to the standby using an out of band process (`archive_command`/`restore_command`)
- Streaming replication moves individual WAL records over a connection between primary and standby (`walsender`/`walreceiver`)
- Since there is minimal processing of these WAL files, both these methods are relatively simple and pretty reliable. A lot of HA/replica setups use them.
- Since WAL is global, can't control what changes you want to replicate.

https://www.postgresql.org/docs/current/warm-standby.html

Agenda

- Overview of Postgres replication methods
- Logical Replication
  - Behind the scenes
  - Protocol messages - pgoutput
- Protocol **v1**
  - Demo
- Protocol **v2** - streaming of in-progress transactions
  - Demo
  - Challenges in implementing Protocol **v2**
  - Comparing **v1** and **v2**
- Protocol **v3** - prepared transactions
- Protocol v4 - parallel apply of large in-progress transactions
- Conclusion

Overview of Postgres Replication Methods

- Replication between Postgres useful for HA, backups, read replicas
- WAL - write-ahead log
  - Transaction durability
  - Stores all changes (even uncommitted)
- Log shipping (`archive_command`/`restore_command`)
  - Move entire WAL files from server to server
- Streaming Replication (`walsender`/`walreceiver`)
  - Ship individual WAL records over connection
- Minimal processing of WAL, reliable methods
  - Used by HA setups or read replicas
- WAL is global, can’t filter unneeded changes

https://www.postgresql.org/docs/current/warm-standby.html

How logical replication works

- Logical replication uses the streaming replication protocol but chooses to “process” the WAL records before sending to standby.
- Because of the processing, we’re able to filter out changes we don’t want at the table/column/DML type/row(!) level.
- Postgres instances of different versions can use logical replication with each other; useful for upgrades
- Exposed to users via Postgres publications and subscriptions. Users can easily setup and manage logical replication via SQL statements.
- Subscriptions wrap around a logical replication slot which does the actual CDC, external ETL tools work with replication slots directly.

https://www.postgresql.org/docs/current/sql-createsubscription.html

Logical Replication

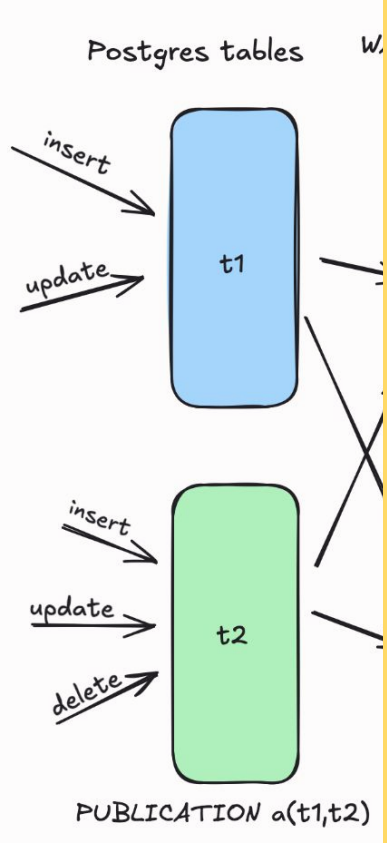
- Introduced in Postgres 10 (Oct 2017).
- More flexible method of replication
  - Online migrations/backups
  - Version upgrades/cross version replication
- Reuses streaming replication protocol
  - “Process” WAL records to map to txns before sending
- Ability to filter out changes at the table/column/DML type/row(!) level.
- Exposed to users via SQL `PUBLICATION/SUBSCRIPTION`
  - Publication on primary selects tables to replicate
  - Subscription on standby attaches to publication and replicates
  - Subscriptions wrap around logical replication slots which do CDC
  - ETL tools like PeerDB and Debezium use replication slots directly

https://www.postgresql.org/docs/current/sql-createsubscription.html

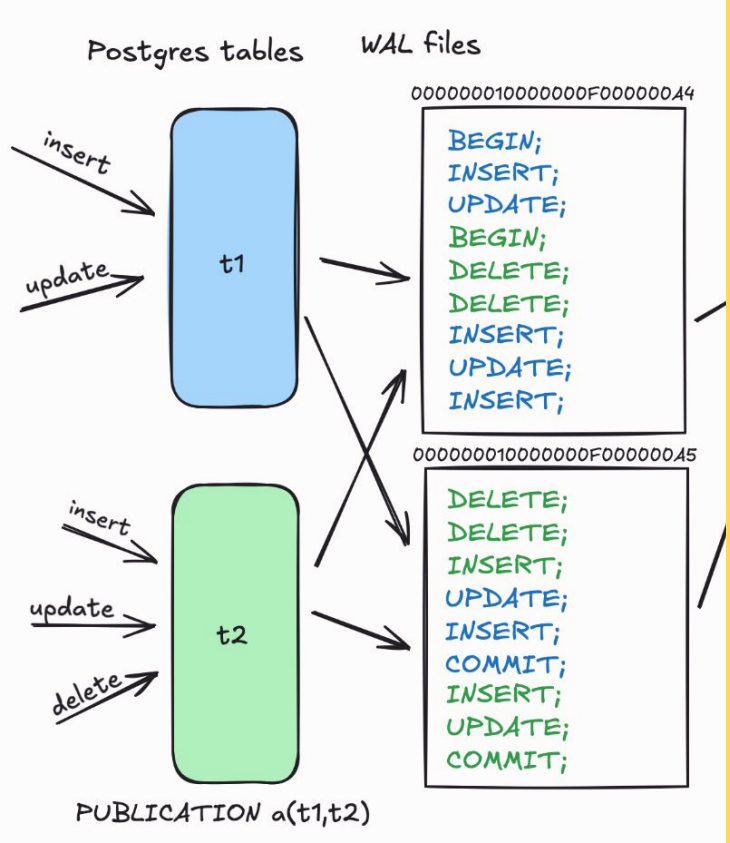
Logical Replication: behind the scenes

- The logical replication protocol requires a lot of infrastructure to turn WAL records into a sequence of messages that can fully represent a transaction.
- Reordering
  - Txn can span multiple WAL files and/or be interleaved with other txns
  - Need to be reassembled from WAL files aka “**reordering**”
  - Large txns can consume lots of memory or spill to disk
- Output plugin
  - Once reordered, passed to output plugin via callbacks
  - Most common one is **pgoutput**, but several others exist (just a C library)
  - Output plugins aim to write changes in a standardized format
- Subscriber periodically acks it has read messages up to a certain point (LSN) and the slot is then able to “advance” and discard older WAL.

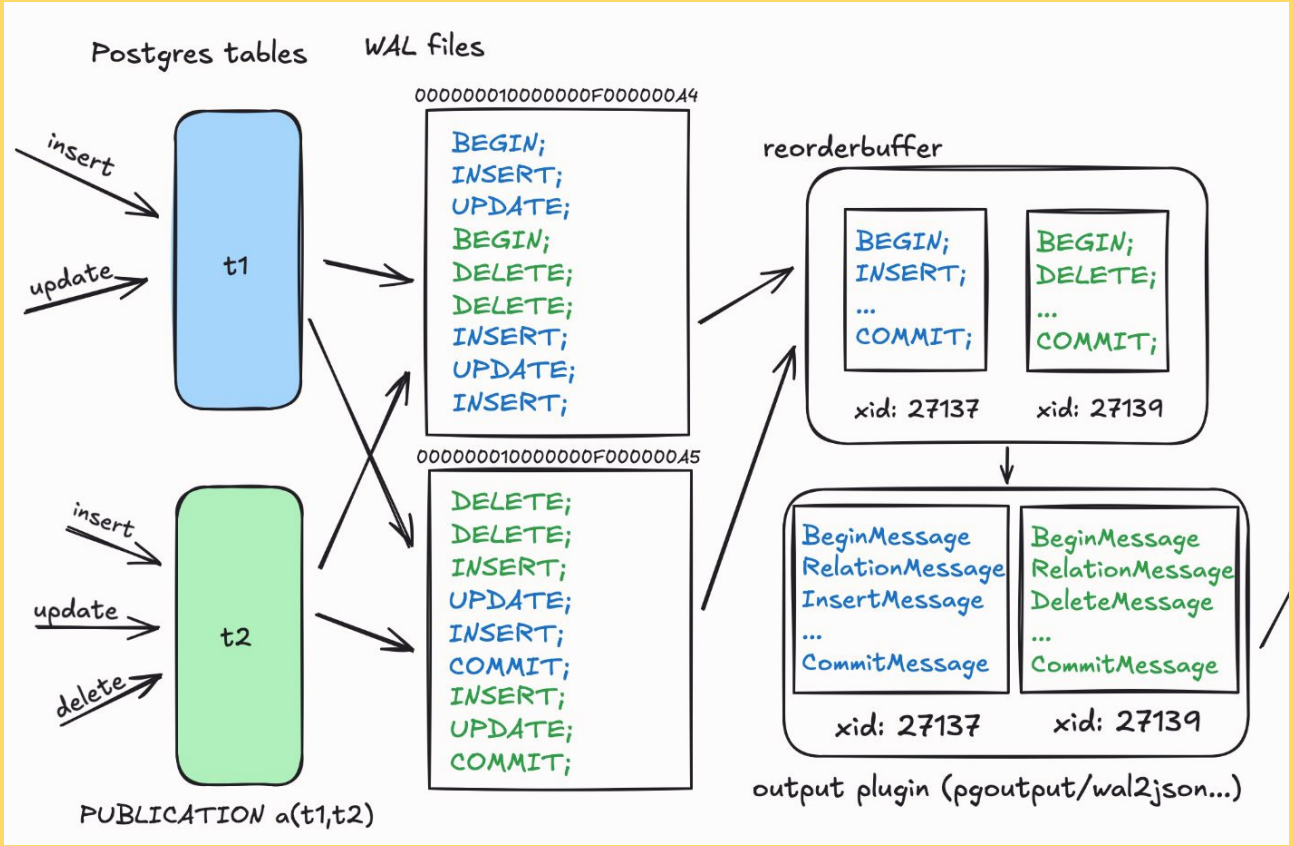
Logical Replication: behind the scenes



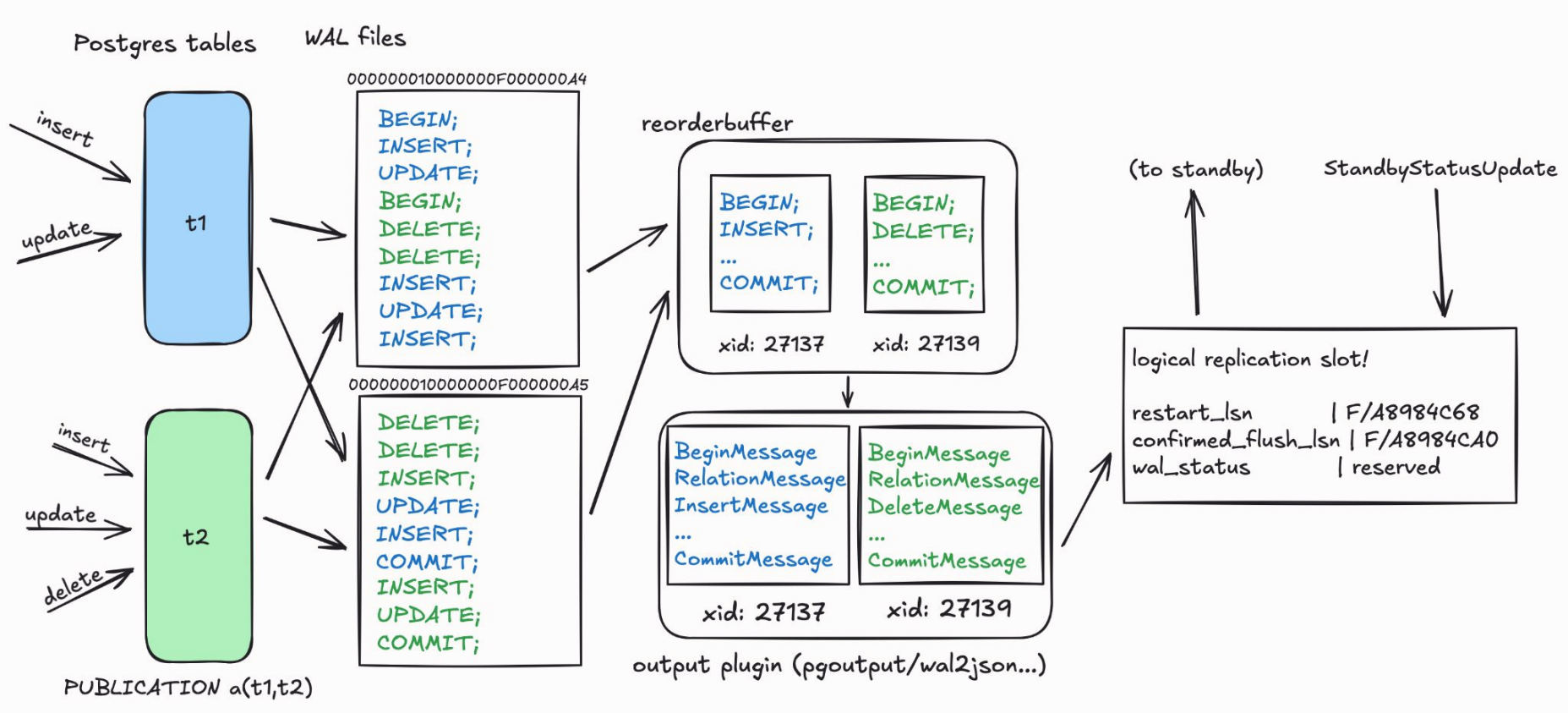
Logical Replication: behind the scenes



Logical Replication: behind the scenes



Logical Replication: behind the scenes



```
typedef struct OutputPluginCallbacks
{
    LogicalDecodeStartupCB startup_cb;
    LogicalDecodeBeginCB begin_cb;
    LogicalDecodeChangeCB change_cb;
    LogicalDecodeTruncateCB truncate_cb;
    LogicalDecodeCommitCB commit_cb;
    LogicalDecodeMessageCB message_cb;
    LogicalDecodeFilterByOriginCB filter_by_origin_cb;
    LogicalDecodeShutdownCB shutdown_cb;

    /* streaming of changes at prepare time */
    LogicalDecodeFilterPrepareCB filter_prepare_cb;
    LogicalDecodeBeginPrepareCB begin_prepare_cb;
    LogicalDecodePrepareCB prepare_cb;
    LogicalDecodeCommitPreparedCB commit_prepared_cb;
    LogicalDecodeRollbackPreparedCB rollback_prepared_cb;

    /* streaming of changes */
    LogicalDecodeStreamStartCB stream_start_cb;
    LogicalDecodeStreamStopCB stream_stop_cb;
    LogicalDecodeStreamAbortCB stream_abort_cb;
    LogicalDecodeStreamPrepareCB stream_prepare_cb;
    LogicalDecodeStreamCommitCB stream_commit_cb;
    LogicalDecodeStreamChangeCB stream_change_cb;
    LogicalDecodeStreamMessageCB stream_message_cb;
    LogicalDecodeStreamTruncateCB stream_truncate_cb;
} OutputPluginCallbacks;
```

Protocol Messages - pgoutput

```
MessageTypeBegin     MessageType = 'B'
MessageTypeMessage   MessageType = 'M'
MessageTypeCommit    MessageType = 'C'
MessageTypeOrigin    MessageType = 'O'
MessageTypeRelation  MessageType = 'R'
MessageTypeType      MessageType = 'Y'
MessageTypeInsert    MessageType = 'I'
MessageTypeUpdate    MessageType = 'U'
MessageTypeDelete    MessageType = 'D'
MessageTypeTruncate  MessageType = 'T'
```

```
type InsertMessage struct {
    baseMessage
    RelationID uint32
    Tuple      *TupleData
}
```

```
type BeginMessage struct {
    baseMessage
    //FinalLSN is the final LSN of the transaction.
    FinalLSN LSN
    // CommitTime is the commit timestamp of the transaction.
    CommitTime time.Time
    // Xid of the transaction.
    Xid uint32
}
```

```
type RelationMessage struct {
    baseMessage
    RelationID      uint32
    Namespace       string
    RelationName    string
    ReplicaIdentity uint8
    ColumnNum       uint16
    Columns         []*RelationMessageColumn
}
```

https://www.postgresql.org/docs/current/protocol-logicalrep-message-formats.html

Using the `START_REPLICATION` command, `pgoutput` accepts the following options:

`proto_version`

Protocol version. Currently versions `1`, `2`, `3`, and `4` are supported. A valid version is required.

Version `2` is supported only for server version 14 and above, and it allows streaming of large in-progress transactions.

Version `3` is supported only for server version 15 and above, and it allows streaming of two-phase commits.

Version `4` is supported only for server version 16 and above, and it allows streams of large in-progress transactions to be applied in parallel.

https://www.postgresql.org/docs/17/protocol-logical-replication.html

Protocol **v1** - the status quo

- Transactions are only sent on the slot once they have been committed, regardless of size
- Receive **BeginMessage** which contains the transaction id (**xid**)
- Receive **Insert/Update/DeleteMessages** for txn corresponding to each row
  - **RelationMessages** to decode them as needed
  - Can be applied immediately
- **CommitMessage** to signal end of txn, xid won't be seen again

Demo of Protocol v1

Protocol v1 In Action

```
temp=# BEGIN;
BEGIN
temp=*# INSERT INTO polorex_table VALUES(default,default);
INSERT 0 1
temp=*# INSERT INTO polorex_table VALUES(default,default);
INSERT 0 1
temp=*# COMMIT;
COMMIT
```

```
INFO received BeginMessage, beginning reading of committed transanction xid=28048975
INFO received RelationMessage for table relationID=59126 relationName=polorex_table
INFO InsertMessage processed id=412003 txt=520e8e84b2ee7b15e972c7c9f620e537 xid=28048975
INFO InsertMessage processed id=412004 txt=63aea18c90c807a3f965e2b5dd4cca72 xid=28048975
INFO received CommitMessage, finished reading of committed transanction commitLSN=F/A8FAB968
INFO Finished reading transaction minID=412003 maxID=412004 count=2 commitLSN=F/A8FAB968 txnReadTime=0s
```

https://github.com/PeerDB-io/polorex

Protocol **v2**

- Introduced in Postgres **14** (September 2021)
  - supports **streaming of large in-progress transactions**
  - needs to be opted in while starting replication.
- **StreamStartMessage** instead of **BeginMessage**
  - First time we see this for a given xid, we initialize this txn.
- **RelationMessage** followed by **Insert/Update/DeleteMessages**. But message format has been changed so each individual Message now contains the xid as well.
- **StreamStopMessage** is the end of the txn stream, but not the txn
- **StreamCommitMessage** for COMMIT or **StreamAbortMessage** for ROLLBACK
  - This could come at any point after the first StreamStart/StreamStop cycle.
  - Several txns can be in the middle of streaming, need to track them all independently.
- We can also still receive txns already committed as per the v1 protocol.

Demo of Protocol v2

Protocol v2 In Action

```
temp=# SHOW debug_logical_replication_streaming;
 debug_logical_replication_streaming
-------------------------------------
 immediate
(1 row)

temp=# BEGIN;
BEGIN
temp=*# INSERT INTO polorex_table VALUES(default,default);
INSERT 0 1
temp=*# INSERT INTO polorex_table VALUES(default,default);
INSERT 0 1
temp=*# COMMIT;
COMMIT
```

```
INFO received StreamStartMessage, reading messages for transaction xid=28048976
INFO received RelationMessage for table relationID=59126 relationName=polorex_table xid=28048976
INFO InsertMessage processed id=412005 txt=65c80b33f028a57496e0dbbc1dbd7d71 xid=28048976
INFO received StreamStopMessage, continuing xid=28048976 currentCount=1
INFO received StreamStartMessage, reading messages for transaction xid=28048976
INFO InsertMessage processed id=412006 txt=8f7098acb8836582f842330dd724fe32 xid=28048976
INFO received StreamStopMessage, continuing xid=28048976 currentCount=2
INFO received StreamCommitMessage, transaction finalized xid=28048976
INFO Finished reading transaction minID=412005 maxID=412006 count=2 commitLSN=F/A8FAEFE0 txnReadTime=40.406792ms streamCount=2
```

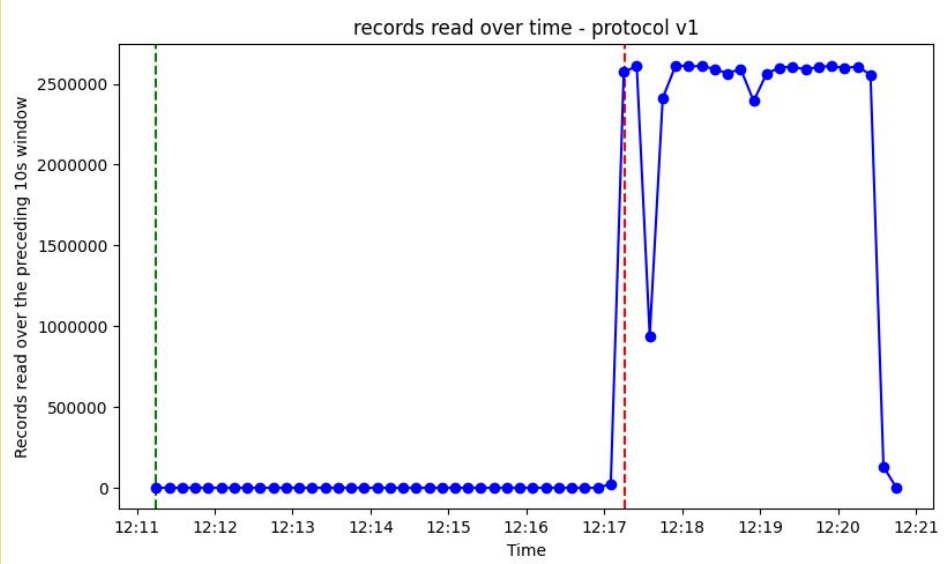
Challenges in implementing protocol v2

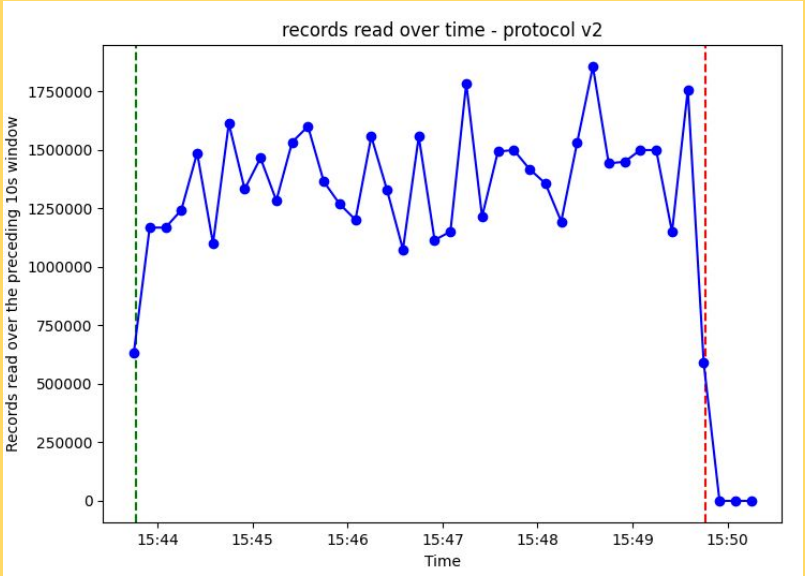
- How often a transaction streams changes is currently informed by a setting **logical_decoding_work_mem**. This controls how much memory the process of “reordering” txns can consume. Lower values imply more frequent streaming, and this can hurt overall throughput.
- Changes can’t be applied as we see it, need to wait for decision. Postgres opts for “spooling” these changes into a file and applying them after commit.
- Postgres supports transactional DDL, therefore RelationMessages for streamed transactions can only be used for that transaction, need to track them.
- Subtransactions (`SAVEPOINT`) exist, and they can abort independently. Only top level transaction can commit. This becomes visible when subtransactions are streamed out.

Challenges in Implementing Protocol v2

- Streaming frequency depends on setting **logical_decoding_work_mem**
  - Controls max memory used for “reordering” txns
  - Lower values lead to more frequent streaming
- Need to wait for COMMIT/ROLLBACK to apply
  - All changes for txn need to be staged
  - Postgres opts for “spooling” to file
- Postgres supports transactional DDL
  - RelationMessages for streamed txns is therefore per txn
  - Can be used elsewhere if txn COMMITs
- Subtransactions (`SAVEPOINT`)
  - Subtxns can abort independently, visible if streamed
  - Only toplevel txns can commit
  - Need to map changes and aborts to subtxns

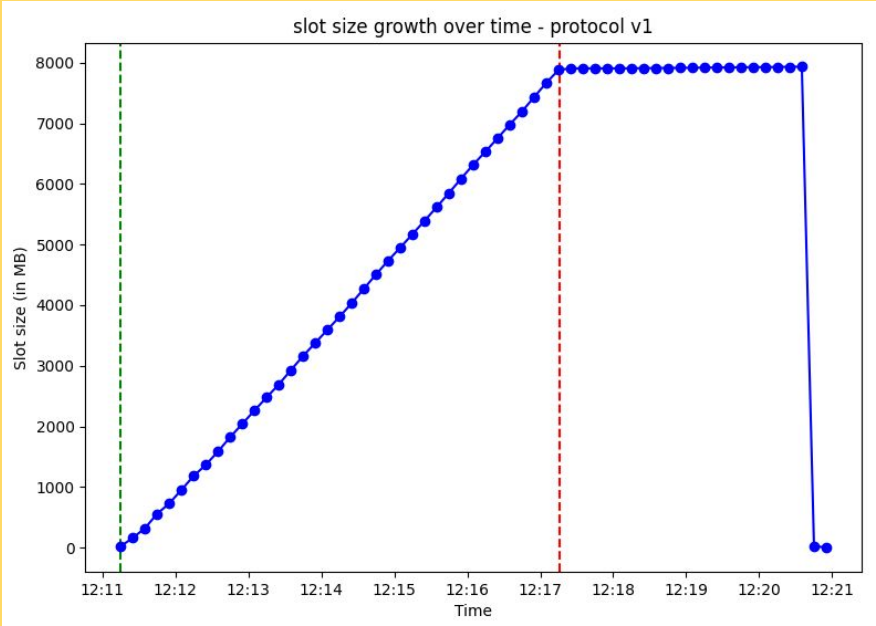
Comparing performance of Protocol v1 and v2 - records

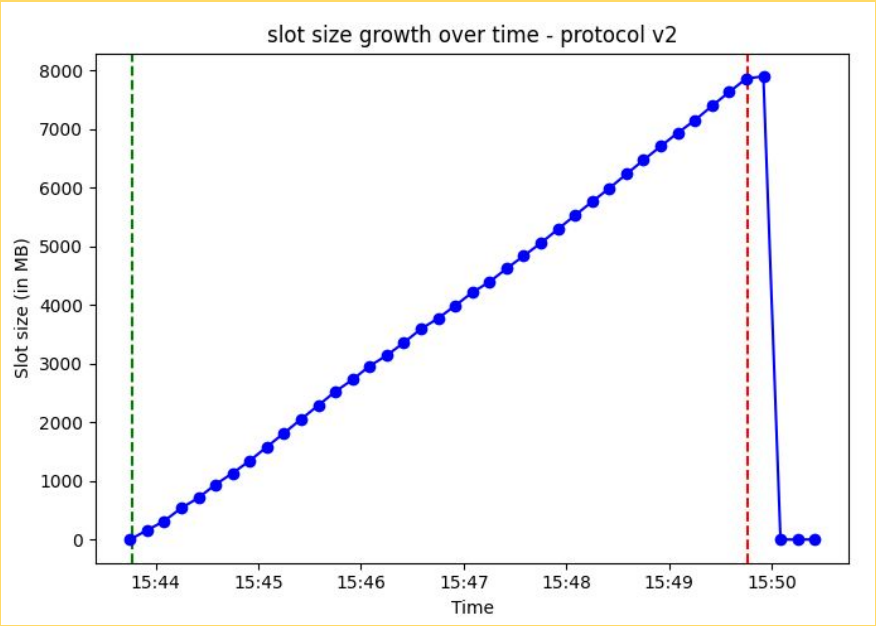




Comparing performance of Protocol v1 and v2 - Slot Size

All graphs show 2 concurrent txns inserting rows into same table. Both transactions insert rows in 100 batches of 250K rows, totalling 50 million rows.





https://blog.peerdb.io/exploring-versions-of-the-postgres-logical-replication-protocol

An Aside: 2PC and Prepared Transactions

- Two-phase commit
  - Distributed algorithm involves “coordinator” and “participants”
  - Enables executing a fully atomic txn across all participants if they support 2PC, even if they are different systems (DBs, message queues)
  - All participants prepare for commit, ensuring the commit can’t fail post preparation. If any participant is unable to prepare, ROLLBACK
  - After everyone prepares, coordinator can COMMIT without any failures possible
- Postgres 2PC via prepared transactions
  - Disabled by default
  - Can run **PREPARE** in a txn, which does most work needed for COMMIT
  - Still need to wait for COMMIT/ROLLBACK decision on a prepared txn
- While a transaction is in this prepared state, it has a lot of characteristics of long-running txns; hold locks open for long and can block VACUUMs.

An Aside: 2PC and Prepared Transactions

```
temp=# BEGIN;
BEGIN
temp=*# INSERT INTO polorex_table VALUES(default,default);
INSERT 0 1
temp=*# PREPARE TRANSACTION 'cool';
PREPARE TRANSACTION
```

```
temp=# SELECT * FROM pg_prepared_xacts;
-[ RECORD 1 ]-----------------------------
transaction | 28048830
gid         | cool
prepared    | 2025-03-02 12:07:59.237485+00
owner       | postgres
database    | temp
```

```
temp=# SELECT * FROM pg_locks WHERE transactionid=28048830;
-[ RECORD 1 ]------+--------------
locktype           | transactionid
database           |
relation           |
page               |
tuple              |
virtualxid         |
transactionid      | 28048830
classid            |
objid              |
objsubid           |
virtualtransaction | 14/3
pid                |
mode               | ExclusiveLock
granted            | t
fastpath           | f
waitstart          |
```

```
temp=# COMMIT PREPARED 'cool';
COMMIT PREPARED
```

Protocol **v3**: Prepared Transactions

- Introduced in Postgres **15** (September 2022)
  - supports **prepared transactions**
  - needs to be opted in **while creating the replication slot**.
- Before this change, prepared txns showed up only after COMMIT PREPARED or had **StreamCommit/StreamAbort** if streamed.
- Prepared transactions
  - Start with **BeginPrepare** (not **Begin**) message
  - End with a **Prepare** (not **Commit**) message
  - To be applied if we see **CommitPrepared**, else **RollbackPrepared**
- Prepared transactions that were streamed
  - End with a **StreamPrepare** message instead of **StreamCommit**
  - Afterwards same as earlier

Protocol v4: Parallel Apply

- Introduced in Postgres **16** (September 2023)
  - supports **parallel apply of large streamed transactions**
  - needs to be opted in while starting replication.
- Instead of streaming txn being “spooled” to file, a new “parallel apply worker” is spun up
  - Changes for this txn are sent to parallel apply worker and applied
  - Assigned till txn commit
  - Helps with applying large transactions faster
- Parallel apply not always used
  - Can lead to data inconsistencies and deadlocks
  - Some transactions fallback to spooling
- Not many changes in the overall protocol
  - subscriber side changes in how txns are handled
  - spin up and logic of parallel workers.

Conclusion

- Over the past 4 years, significant changes have been introduced in the logical replication protocol by the Postgres team.
- These have led to improved performance for replicating some workloads and also better support for some features of Postgres
- For users of Postgres SUBSCRIPTIONs, these improvements can simply be opted into by upgrading and choosing parameters when they create/start logical replication.
- For other ETL tools, there can be significant improvements in performance if we make changes to support newer protocol versions (especially v2) but these changes need to be done carefully.
- We hope the Postgres team continues to invest in logical replication and makes it more performant while keeping it easy to use.