Logical Replication in PostgreSQL

PostgreSQL provides two main types of replication: Physical Streaming Replication and Logical Replication. In this blog post, we explore the details of Logical Replication in PostgreSQL. We will compare it with Physical Streaming Replication and discuss various aspects such as how it works, use case, when it's useful, its limitations, and key points to keep in mind.

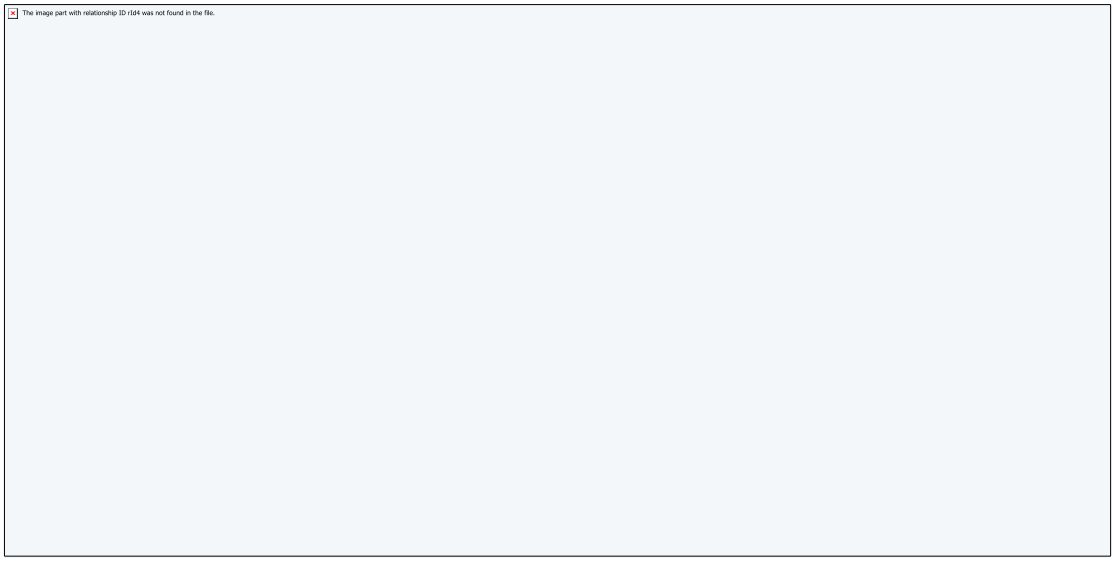
What is Logical Replication and how does it work?

Logical Replication in PostgreSQL is designed for replicating specific tables, rows, or columns between database servers. It uses a publisher-subscriber model where the publisher sends changes and the subscriber applies them. This is different from Physical Replication, which replicates the entire database at the block/page level using WAL records.

Key components in Logical Replication include:

1. **Logical Replication Worker**: Manages replication tasks. Checks worker state on subscriber side. When a new subscription is created/enabled, it spawns a walsender process on the publisher side.
2. **Walsender**: Decodes WAL contents and reassembles transaction changes, sending them to subscribers or discarding them if a transaction aborts.
3. **Decoder**: Uses the PostgreSQL standard plugin output (pgoutput) for decoding. Note that all transactions are fully decoded on the publisher and only then sent to the subscriber as a whole. This behavior is deduced by the streaming option while creating a subscription. Check more: Streaming option for Subscription.
4. **Initial Synchronization Worker**: Synchronizes initial/existing data from the publisher by creating a temporary replication slot and running a COPY command.
5. **Apply Worker**: Applies the incremental changes on the subscriber side.

The replication ensures transactional consistency by applying changes in the commit order on the subscriber side. Each subscription receives changes through one replication slot, and there can be multiple table synchronization workers to expedite the process, only one per table. After initial data copying, real-time changes are sent and applied.



Source: AWS

When should Logical Replication be used instead of Physical Streaming Replication, and what distinguishes it from Streaming Replication?

Here are some key reasons why logical replication is needed in PostgreSQL:

- Logical replication allows for the replication of chosen tables or specific rows and columns, rather than replicating the entire database as physical replication does. This is particularly useful when only certain parts of the data need to be replicated. This is also essential for complying with legal regulations in different regions. For instance, you can replicate non-sensitive data to a subscriber outside the US, while keeping all sensitive dat replicated within subscribers located in the US.
- Unlike streaming replication which requires the same major version, logical replication supports data replication across different major versions of PostgreSQL. This is beneficial for executing major version upgrades with minimal downtime.
- Logical replication supports the real-time consolidation of data from various sources into a single, centralized reporting or analytical database.
- Subscribers in logical replication setups can perform write operations, unlike streaming replication where replica is read-only mode. It also doesn't require the same system configurations between the publisher and subscriber.
- Logical replication in PostgreSQL can be utilized to set up a bi-directional replication system where each node can accept write operations and replicate these changes to other nodes. However, in such a scenario, it's essential to prepare for write-level conflicts and avoid circular replication. From PostgreSQL 16 onwards, a new option **ORIGIN** has been added to the subscription settings. It tells the publisher to send only changes that do not have replication origin(only send writes performed on publisher) or to send all changes, which includes both the local changes and those replicated from other sources.
- Logical replication utilizes WAL data but optimizes it by filtering and transmitting only the required data. This leads to reduced bandwidth and storage needs compared to physical replication, which replicates all WAL data.

What are the limitations of Logical Replication?

1. Tables being replicated logically must have a primary key or a replica identity set. We will discuss this further below.
2. Logical replication does not replicate DDL changes. For instance, changes like index creation, tablespace alterations, vacuum or altering the data type of a column are not replicated.
3. Logical replication is restricted to table data. It does not replicate other database objects like roles, sequences, or schema changes.
4. Logical replication does not resolve conflicts that may arise due to concurrent writes on the primary and the replica. Conflict management has to be handled externally.
5. Logical replication can introduce additional load on the primary database because it needs to transform WAL records into logical change records, which can be resource-intensive.
6. In cases of subscriber downtime, this can lead to increased disk space usage on the primary server. The primary server uses replication slots to keep subscribers in sync which means it needs to retain WAL logs until they are confirmed to be received by all subscribers.

What is Replica Identity?

For logical replication of UPDATE and DELETE operations in PostgreSQL, identifying the correct rows on the subscriber side requires one of the following:

1. **Primary Key (Default Replica Identity)**: When updating/deleting rows with a primary key, the system publishes the old primary key values and all new column values to the WAL on the publisher side, which are then sent and applied to the subscriber.
2. **Unique Index with Not Null Columns (Replica Identity Index)**: Updates/deletes on tables with a unique index result in the publication of the old index values (if the unique indexed column is updated) and all column values to the WAL, which are then transmitted to and applied on the subscriber.
3. **All Columns (Full Replica Identity)**: This method treats all columns as a single key, publishing both old and new values of all columns to the WAL. This approach can lead to excessive logging, increased data transfer, and unnecessary disk I/O.

How to add or remove columns in tables that are involved in logical replication?

When implementing schema or DDL changes in a system using logical replication, the order of applying these changes is crucial.

For adding columns, start by making changes on the subscriber side and then proceed to the publisher. Conversely, when dropping columns, remove them from the publisher first, followed by the subscribers. Not following this sequence can stop logical replication, which requires manual intervention.

How to speed up the initial data syncing process on the subscriber side?

First understand the following GUC parameters(Applies on subscriber side only):

max_logical_replication_workers = Specifies maximum number of logical replication workers. This includes both apply workers(On subscriber side) and table synchronization workers.

max_sync_workers_per_subscription = increasing max_sync_workers_per_subscription only affects the number of tables that are synchronized in parallel, not the number of workers per table.

- To enhance the initial synchronization speed of tables in logical replication, you should increase the values of **max_logical_replication_workers** and **max_sync_workers_per_subscription** on the subscriber side. Keep in mind that max_logical_replication_workers should not exceed **max_worker_processes**, and **max_sync_workers_per_subscription** should be less than or equal to **max_logical_replication_workers**.
- If dealing with large tables, Consider dividing your tables for example: put large tables in separate publications and small tables in another one.
- If dealing with large indexes, consider removing them during the initial sync and then recreating them using the **CREATE INDEX CONCURRENTLY** command to avoid blocking reads/writes.
- Always monitor disk and CPU usage on the subscriber side to ensure that there is no performance issue.

PostgreSQL 16 has introduced several enhancements to its logical replication capabilities. One of the key features is the ability to copy initial data using a binary format, which marks a significant improvement over the previous text format. Check here for Binary format inside COPY command: https://www.postgresql.org/docs/16/sql-copy.html

How to find tables with primary keys?

```
select tab.table_schema,
```

```
    tab.table_name

from information_schema.tables tab

left join information_schema.table_constraints tco

       on tab.table_schema = tco.table_schema

       and tab.table_name = tco.table_name

       and tco.constraint_type = 'PRIMARY KEY'

where tab.table_type = 'BASE TABLE'

   and tab.table_schema in (ADD SCHEMA NAMES HERE)

   and tco.constraint_name is not null

order by table_schema,

      table_name;Copy to Clipboard
```

What factors lead to Logical Replication lags?

- Replication lag can occur when data transmission between the publisher and the subscriber is slowed due to unstable network connections. This is especially true in environments where network reliability is an issue.
- If hardware resources such as CPU, memory, or disk I/O are insufficient on either the publisher or the subscriber, it can negatively affect the efficiency of the replication process.
- Large transactions occurring on the publisher can also cause a replication delay as these transactions are applied in commit order. In such scenarios, smaller committed transactions may end up lagging behind.

How to monitor Logical Replication?

Run the following query on publisher side:

```
SELECT slot_name,

       active,

       confirmed_flush_lsn,

       Pg_current_wal_lsn(),

       Pg_size_pretty(Pg_wal_lsn_diff(Pg_current_wal_lsn(), restart_lsn))AS retained_walsize,

       Pg_size_pretty(Pg_wal_lsn_diff(Pg_current_wal_lsn(), confirmed_flush_lsn)) AS
subscriber_lag

FROM pg_replication_slots;

slot_name | active | confirmed_flush_lsn | pg_current_wal_lsn | retained_walsize |
subscriber_lag

-----------+--------+---------------------+--------------------+------------------+-----------
-----

 mart_sub  | t  | 0/DC29108       | 0/DC29108      | 56 bytes     | 0 bytesCopy to Clipboard
```

slot_name is the name of the subscriber.

Active is the state of logical replication. ‘t’ means it is running without errors.

confirmed_flush_lsn is the wal lsn record replayed on the Subscriber side.

pg_current_wal_lsn is the current wal record number on the publisher.

retained_walsize is the size of the wal retained by the publisher for the slot. Subscriber will start from restart_lsn point after disconnection.

Subscriber_lag defines the overall replication delay between publisher and subscriber.

Make sure that the **active** column is equal to ‘t’. If it shows ‘f’, then check for errors inside the log file on the publisher side.

Another query to run on publisher side is:

```
select

  pid,

  application_name,

  pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), sent_lsn)) sending_lag,

 pg_size_pretty(pg_wal_lsn_diff(sent_lsn, flush_lsn)) receiving_lag,

  pg_size_pretty(pg_wal_lsn_diff(flush_lsn, replay_lsn)) replaying_lag,

  pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), replay_lsn)) total_lag

from pg_stat_replication;Copy to Clipboard
```

application_name shows the name of the subscriber.

sending_lag could indicate heavy load on primary.

receiving_lag could indicate network issues or replica under heavy load.

replaying_lag could indicate that the replica is under heavy load.

total_lag defines the overall replication delay between publisher and subscriber.

How to add new tables to existing logical replication?

Creating a publication using the **FOR ALL TABLES** option ensures that any tables added to the database in the future are automatically included in the publication.

Similarly, when a publication is established with the **FOR TABLES IN SCHEMA** option, it automatically incorporates any future tables created within that specific schema into the publication.

In cases where neither of these options is used, you must follow these steps:

- Include the table in the publication

```
ALTER P
```

Logical Replication in PostgreSQL

PostgreSQL provides two main types of replication: Physical Streaming Replication and Logical Replication. In this blog post, we explore the details of Logical Replication in PostgreSQL. We will compare it with Physical Streaming Replication and discuss various aspects such as how it works, use case, when it's useful, its limitations, and key points to keep in mind.

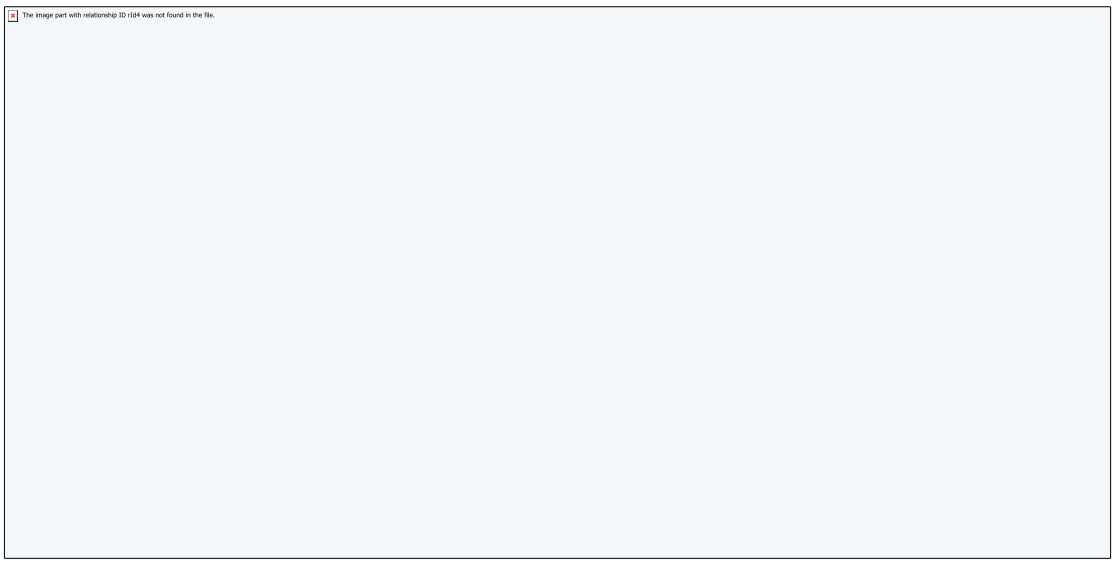
What is Logical Replication and how does it work?

Logical Replication in PostgreSQL is designed for replicating specific tables, rows, or columns between database servers. It uses a publisher-subscriber model where the publisher sends changes and the subscriber applies them. This is different from Physical Replication, which replicates the entire database at the block/page level using WAL records.

Key components in Logical Replication include:

1. **Logical Replication Worker**: Manages replication tasks. Checks worker state on subscriber side. When a new subscription is created/enabled, it spawns a walsender process on the publisher side.
2. **Walsender**: Decodes WAL contents and reassembles transaction changes, sending them to subscribers or discarding them if a transaction aborts.
3. **Decoder**: Uses the PostgreSQL standard plugin output (pgoutput) for decoding. Note that all transactions are fully decoded on the publisher and only then sent to the subscriber as a whole. This behavior is deduced by the streaming option while creating a subscription. Check more: Streaming option for Subscription.
4. **Initial Synchronization Worker**: Synchronizes initial/existing data from the publisher by creating a temporary replication slot and running a COPY command.
5. **Apply Worker**: Applies the incremental changes on the subscriber side.

The replication ensures transactional consistency by applying changes in the commit order on the subscriber side. Each subscription receives changes through one replication slot, and there can be multiple table synchronization workers to expedite the process, only one per table. After initial data copying, real-time changes are sent and applied.



Source: AWS

When should Logical Replication be used instead of Physical Streaming Replication, and what distinguishes it from Streaming Replication?

Here are some key reasons why logical replication is needed in PostgreSQL:

- Logical replication allows for the replication of chosen tables or specific rows and columns, rather than replicating the entire database as physical replication does. This is particularly useful when only certain parts of the data need to be replicated. This is also essential for complying with legal regulations in different regions. For instance, you can replicate non-sensitive data to a subscriber outside the US, while keeping all sensitive dat replicated within subscribers located in the US.
- Unlike streaming replication which requires the same major version, logical replication supports data replication across different major versions of PostgreSQL. This is beneficial for executing major version upgrades with minimal downtime.
- Logical replication supports the real-time consolidation of data from various sources into a single, centralized reporting or analytical database.
- Subscribers in logical replication setups can perform write operations, unlike streaming replication where replica is read-only mode. It also doesn't require the same system configurations between the publisher and subscriber.
- Logical replication in PostgreSQL can be utilized to set up a bi-directional replication system where each node can accept write operations and replicate these changes to other nodes. However, in such a scenario, it's essential to prepare for write-level conflicts and avoid circular replication. From PostgreSQL 16 onwards, a new option **ORIGIN** has been added to the subscription settings. It tells the publisher to send only changes that do not have replication origin(only send writes performed on publisher) or to send all changes, which includes both the local changes and those replicated from other sources.
- Logical replication utilizes WAL data but optimizes it by filtering and transmitting only the required data. This leads to reduced bandwidth and storage needs compared to physical replication, which replicates all WAL data.

What are the limitations of Logical Replication?

1. Tables being replicated logically must have a primary key or a replica identity set. We will discuss this further below.
2. Logical replication does not replicate DDL changes. For instance, changes like index creation, tablespace alterations, vacuum or altering the data type of a column are not replicated.
3. Logical replication is restricted to table data. It does not replicate other database objects like roles, sequences, or schema changes.
4. Logical replication does not resolve conflicts that may arise due to concurrent writes on the primary and the replica. Conflict management has to be handled externally.
5. Logical replication can introduce additional load on the primary database because it needs to transform WAL records into logical change records, which can be resource-intensive.
6. In cases of subscriber downtime, this can lead to increased disk space usage on the primary server. The primary server uses replication slots to keep subscribers in sync which means it needs to retain WAL logs until they are confirmed to be received by all subscribers.

What is Replica Identity?

For logical replication of UPDATE and DELETE operations in PostgreSQL, identifying the correct rows on the subscriber side requires one of the following:

1. **Primary Key (Default Replica Identity)**: When updating/deleting rows with a primary key, the system publishes the old primary key values and all new column values to the WAL on the publisher side, which are then sent and applied to the subscriber.
2. **Unique Index with Not Null Columns (Replica Identity Index)**: Updates/deletes on tables with a unique index result in the publication of the old index values (if the unique indexed column is updated) and all column values to the WAL, which are then transmitted to and applied on the subscriber.
3. **All Columns (Full Replica Identity)**: This method treats all columns as a single key, publishing both old and new values of all columns to the WAL. This approach can lead to excessive logging, increased data transfer, and unnecessary disk I/O.

How to add or remove columns in tables that are involved in logical replication?

When implementing schema or DDL changes in a system using logical replication, the order of applying these changes is crucial.

For adding columns, start by making changes on the subscriber side and then proceed to the publisher. Conversely, when dropping columns, remove them from the publisher first, followed by the subscribers. Not following this sequence can stop logical replication, which requires manual intervention.

How to speed up the initial data syncing process on the subscriber side?

First understand the following GUC parameters(Applies on subscriber side only):

max_logical_replication_workers = Specifies maximum number of logical replication workers. This includes both apply workers(On subscriber side) and table synchronization workers.

max_sync_workers_per_subscription = increasing max_sync_workers_per_subscription only affects the number of tables that are synchronized in parallel, not the number of workers per table.

- To enhance the initial synchronization speed of tables in logical replication, you should increase the values of **max_logical_replication_workers** and **max_sync_workers_per_subscription** on the subscriber side. Keep in mind that max_logical_replication_workers should not exceed **max_worker_processes**, and **max_sync_workers_per_subscription** should be less than or equal to **max_logical_replication_workers**.
- If dealing with large tables, Consider dividing your tables for example: put large tables in separate publications and small tables in another one.
- If dealing with large indexes, consider removing them during the initial sync and then recreating them using the **CREATE INDEX CONCURRENTLY** command to avoid blocking reads/writes.
- Always monitor disk and CPU usage on the subscriber side to ensure that there is no performance issue.

PostgreSQL 16 has introduced several enhancements to its logical replication capabilities. One of the key features is the ability to copy initial data using a binary format, which marks a significant improvement over the previous text format. Check here for Binary format inside COPY command: https://www.postgresql.org/docs/16/sql-copy.html

How to find tables with primary keys?

```
select tab.table_schema,
```

```
    tab.table_name

from information_schema.tables tab

left join information_schema.table_constraints tco

       on tab.table_schema = tco.table_schema

       and tab.table_name = tco.table_name

       and tco.constraint_type = 'PRIMARY KEY'

where tab.table_type = 'BASE TABLE'

   and tab.table_schema in (ADD SCHEMA NAMES HERE)

   and tco.constraint_name is not null

order by table_schema,

      table_name;Copy to Clipboard
```

What factors lead to Logical Replication lags?

- Replication lag can occur when data transmission between the publisher and the subscriber is slowed due to unstable network connections. This is especially true in environments where network reliability is an issue.
- If hardware resources such as CPU, memory, or disk I/O are insufficient on either the publisher or the subscriber, it can negatively affect the efficiency of the replication process.
- Large transactions occurring on the publisher can also cause a replication delay as these transactions are applied in commit order. In such scenarios, smaller committed transactions may end up lagging behind.

How to monitor Logical Replication?

Run the following query on publisher side:

```
SELECT slot_name,

       active,

       confirmed_flush_lsn,

       Pg_current_wal_lsn(),

       Pg_size_pretty(Pg_wal_lsn_diff(Pg_current_wal_lsn(), restart_lsn))AS retained_walsize,

       Pg_size_pretty(Pg_wal_lsn_diff(Pg_current_wal_lsn(), confirmed_flush_lsn)) AS
subscriber_lag

FROM pg_replication_slots;

slot_name | active | confirmed_flush_lsn | pg_current_wal_lsn | retained_walsize |
subscriber_lag

-----------+--------+---------------------+--------------------+------------------+-----------
-----

 mart_sub  | t  | 0/DC29108       | 0/DC29108      | 56 bytes     | 0 bytesCopy to Clipboard
```

slot_name is the name of the subscriber.

Active is the state of logical replication. ‘t’ means it is running without errors.

confirmed_flush_lsn is the wal lsn record replayed on the Subscriber side.

pg_current_wal_lsn is the current wal record number on the publisher.

retained_walsize is the size of the wal retained by the publisher for the slot. Subscriber will start from restart_lsn point after disconnection.

Subscriber_lag defines the overall replication delay between publisher and subscriber.

Make sure that the **active** column is equal to ‘t’. If it shows ‘f’, then check for errors inside the log file on the publisher side.

Another query to run on publisher side is:

```
select

  pid,

  application_name,

  pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), sent_lsn)) sending_lag,

 pg_size_pretty(pg_wal_lsn_diff(sent_lsn, flush_lsn)) receiving_lag,

  pg_size_pretty(pg_wal_lsn_diff(flush_lsn, replay_lsn)) replaying_lag,

  pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), replay_lsn)) total_lag

from pg_stat_replication;Copy to Clipboard
```

application_name shows the name of the subscriber.

sending_lag could indicate heavy load on primary.

receiving_lag could indicate network issues or replica under heavy load.

replaying_lag could indicate that the replica is under heavy load.

total_lag defines the overall replication delay between publisher and subscriber.

How to add new tables to existing logical replication?

Creating a publication using the **FOR ALL TABLES** option ensures that any tables added to the database in the future are automatically included in the publication.

Similarly, when a publication is established with the **FOR TABLES IN SCHEMA** option, it automatically incorporates any future tables created within that specific schema into the publication.

In cases where neither of these options is used, you must follow these steps:

- Include the table in the publication

```
ALTER PUBLICATION ADD TABLE TABLENAME;Copy to Clipboard
```

- Update the subscription to incorporate data from the newly added tables. This step is necessary in all scenarios.

```
ALTER SUBSCRIPTION subscription_name REFRESH PUBLICATION;Copy to Clipboard

UBLICATION ADD TABLE TABLENAME;Copy to Clipboard
```

- Update the subscription to incorporate data from the newly added tables. This step is necessary in all scenarios.

```
ALTER SUBSCRIPTION subscription_name REFRESH PUBLICATION;Copy to Clipboard
```