Selective Column Replication in PostgreSQL

Logical replication in PostgreSQL involves replicating data objects and their modifications based on their replication identity, typically a primary key. By default, this method replicates all columns in a table to another table. Additionally, it allows for the creation of multiple subscriber tables within the same server or across different servers to replicate changes as needed.

Selective column replication, a feature of logical replication in PostgreSQL, enables the selective transfer of data changes from one database to another. This feature offers flexibility by addressing bandwidth and storage optimization concerns. It allows users to choose specific columns to replicate rather than replicating entire tables, ensuring that only essential data is transferred. This selective approach reduces network traffic and storage space required for replication while maintaining data consistency across databases.

In this blog, I will demonstrate the process of replicating specific columns from a table using logical replication. Additionally, we will see how you can create multiple subscribers connected to a single publisher to replicate data across multiple locations.

Why do you generally want to store data in more than one location?

Storing data in multiple locations provides redundancy and ensures data availability in case of hardware failures, natural disasters, or other unexpected events. Additionally, it enhances performance by enabling data to be accessed from the nearest location, reducing latency for users accessing the data.

Install PostgreSQL by following the official guide: https://www.postgresql.org/download/

Scenario, we'll utilize three Docker containers within the same network: **primary**, **standby1**, and **standby2**. The primary server hosts a production table, and we'll selectively replicate its columns to both standby1 and standby2 servers.

Modify the **postgresql.conf** file on the primary server by changing the **wal_level** parameter to **logical**. Ensure to restart the server after making this configuration change.

```
wal_level = logicalCopy to Clipboard
```

Create a source tabled named **production** in the **primary server**

```
production=# CREATE TABLE production (id SERIAL PRIMARY KEY, first_name VARCHAR, last_name
VARCHAR, age INT, address VARCHAR);
CREATE TABLECopy to Clipboard
```

Establish a publication for the source table, The bellow query creates a publication named **production_publication** for the **production** table, specifying that only the columns **id**, **first_name**, and **address** should be replicated.

```
production=# CREATE PUBLICATION production_publication FOR TABLE production (id, first_name,
address);
CREATE PUBLICATIONCopy to Clipboard
```

Navigate to **standby1** and **standby2**, then create the necessary table with the desired columns, which will serve as the destination for replication.

```
standby1=# CREATE TABLE production (id SERIAL PRIMARY KEY, first_name VARCHAR, address VARCHAR);
CREATE TABLE

standby2=#  CREATE TABLE production (id SERIAL PRIMARY KEY, first_name VARCHAR, address VARCHAR);
CREATE TABLECopy to Clipboard
```

Establish a subscription on both servers.

NOTE: This action creates a replication slot on the primary server to monitor replication progress. Ensure subscription names on **standby1** and **standby2** are distinct

```
standby1=# CREATE SUBSCRIPTION standby1_production_subscription CONNECTION 'dbname=production
host=localhost user=postgres port=5432' PUBLICATION production_publication;
NOTICE:  created replication slot "standby1_production_subscription" on publisher
CREATE SUBSCRIPTION

standby2=# CREATE SUBSCRIPTION standby2_production_subscription CONNECTION 'dbname=production
host=localhost user=postgres port=5432' PUBLICATION production_publication;
NOTICE:  created replication slot "standby2_production_subscription" on publisher
CREATE SUBSCRIPTIONCopy to Clipboard
```

NOTE: Update parameters inside connection string W.R.T your database configurations

We'll populate our production table with data using PostgreSQL's built-in **generate_series** function, inserting 5 rows with concatenated values for **first_name**, **last_name**, and **address**, incrementing **age** from 1 to 5.

```
production=# INSERT INTO production (first_name, last_name, age, address) SELECT 'Semab' || x,
'Tariq' || x, x, 'Address' || x FROM generate_series(1, 5) AS x;
INSERT 0 5Copy to Clipboard
```

Data has been successfully inserted into the production table

```
production=# select * from production;
 id | first_name | last_name | age | address
----+------------+-----------+-----+----------
  1 | Semab1     | Tariq1    |   1 | Address1
  2 | Semab2     | Tariq2    |   2 | Address2
  3 | Semab3     | Tariq3    |   3 | Address3
  4 | Semab4     | Tariq4    |   4 | Address4
  5 | Semab5     | Tariq5    |   5 | Address5
(5 rows)Copy to Clipboard
```

Let's check if our data has been successfully replicated to **standby1** and **standby2** servers. Execute this query to display all records from the **production** table.

```
standby1=# select * from production;
 id | first_name | address
----+------------+----------
  1 | Semab1     | Address1
  2 | Semab2     | Address2
  3 | Semab3     | Address3
  4 | Semab4     | Address4
  5 | Semab5     | Address5
(5 rows)

standby2=# select * from production;
 id | first_name | address
----+------------+----------
```

```
  1 | Semab1     | Address1
  2 | Semab2     | Address2
  3 | Semab3     | Address3
  4 | Semab4     | Address4
  5 | Semab5     | Address5
(5 rows)Copy to Clipboard
```

We can see data is successfully replicated.

If a row is deleted from the production table, it will also be automatically removed from subscribed tables.

```
production=# DELETE FROM production WHERE id = 2;
DELETE 1

production=# select * from production;
 id | first_name | last_name | age | address
----+------------+-----------+-----+----------
  1 | Semab1     | Tariq1    |   1 | Address1
  3 | Semab3     | Tariq3    |   3 | Address3
  4 | Semab4     | Tariq4    |   4 | Address4
  5 | Semab5     | Tariq5    |   5 | Address5
(4 rows)

standby1=# select * from production;
 id | first_name | address
----+------------+----------
  1 | Semab1     | Address1
  3 | Semab3     | Address3
  4 | Semab4     | Address4
  5 | Semab5     | Address5
(4 rows)

standby2=# select * from production;
 id | first_name | address
----+------------+----------
  1 | Semab1     | Address1
  3 | Semab3     | Address3
  4 | Semab4     | Address4
  5 | Semab5     | Address5
(4 rows)Copy to Clipboard
```

Likewise, for inserts and deletions, updating a row in the original table will also update the corresponding rows in subscribed tables.

FAQ

Is it possible to alter table names in the source and destination servers?
No, table names cannot be changed within the source and destination servers. Therefore, consistency in table names is necessary for replication.

Is it possible to alter schema names for source and destination servers?
No, schema names cannot be altered. If a production table is created within the public schema, the schema cannot be changed on the destination server.

Is it possible to alter column names within source and destination tables?
No, altering column names is not supported within the source and destination servers.

How to overcome the renaming limitations of selective logical replication?
To overcome these limitations, consider using a custom **ETL** script. This script can transfer data between servers, adjusting table, schema, and column names as needed. It's essential to note that this approach falls outside the scope of logical replication and requires execution as a scheduled cron job. Stay tuned for an upcoming blog post detailing this **ETL** method. Also, you can use **Postgres Foreign Data Wrappers(FDW)** to overcome these issues

Is it possible to update the subscriber table directly?
Directly updating a subscriber table in PostgreSQL is possible but not recommended because it bypasses the replication mechanism, risking data inconsistency and violating the integrity of the replication process.