Bidirectional Replication

Last updated by | Paul Haefeli | Feb 16, 2023 at 1:57 PM PST

Contents

- Intro
- Prerequisites
- Conflict handling
- Walkthrough
    - First: set up unidirectional (one-way) replication
    - Second: set up bidirectional (two-way) replication
    - Cleanup
- Diagnostic queries

This is not officially supported by the PostgreSQL Product Group - customers are to use this solution at their own risk!

Intro

Bidirectional (also known as active-active) replication allows changes between two Azure Database for PostgreSQL - Felxible servers to replicate between one another. Changes made on server "A" will be sent to server "B" and changes made on server "B" will be sent to server "A." We can accomplish this by leveraging the **pglogical** extension.

The official pglogical documentation can be found here - https://github.com/2ndQuadrant/pglogical

Prerequisites

- Deploy two Azure Database for PostgreSQL – Flexible servers.
- The servers must be networked. In the example given, they are both Public Access with appropriate firewall rules in place.
- Set the server parameter **wal_level** to **logical** (requires restart).
- Add **pglogical** to the parameters **extensions** and **shared_preload_libraries** (requires restart).

Conflict handling

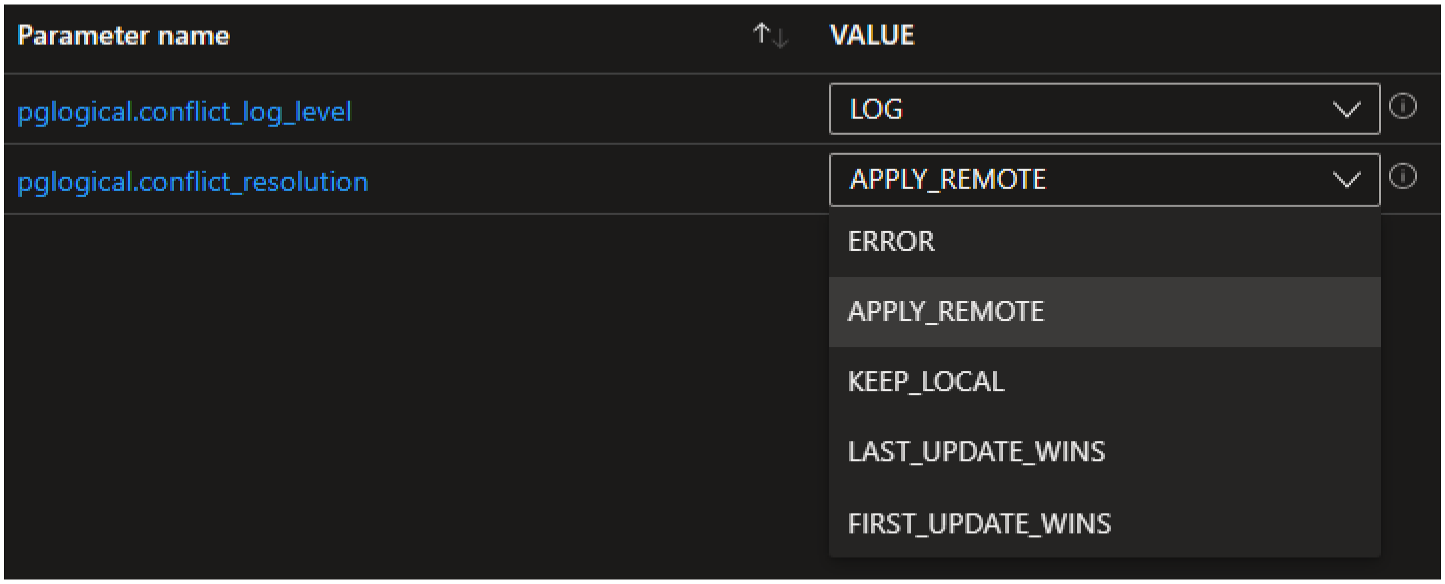
Conflict handling is documented by pglogical here - https://github.com/2ndQuadrant/pglogical#conflicts

In case the node is subscribed to multiple providers, or when local writes happen on a subscriber, conflicts can arise for the incoming changes. These are automatically detected and can be acted on depending on the configuration. The configuration of the conflicts resolver is done via the **pglogical.conflict_resolution** parameter.

- **error** - the replication will stop on error if conflict is detected and manual action is needed for resolving

- **apply_remote** - always apply the change that's conflicting with local data
- **keep_local** - keep the local version of the data and ignore the conflicting change that is coming from the remote node
- **last_update_wins** - the version of data with newest commit timestamp will be kept (this can be either local or remote version)
- **first_update_wins** - the version of the data with oldest timestamp will be kept (this can be either local or remote version)

The resolved conflicts are logged using the log level set using **pglogical.conflict_log_level**. This parameter defaults to **LOG**. If set to lower level than log_min_messages the resolved conflicts won't appear in the server log.



Parameter name	VALUE
pglogical.conflict_log_level	LOG
pglogical.conflict_resolution	APPLY_REMOTE

Walkthrough

First: set up unidirectional (one-way) replication

- **Set up the first Publisher - we will call this "node1."**

  1. Create the pglogical extension

     ```
     CREATE EXTENSION pglogical;
     ```

  2. Create the table we want to replicate

     ```
     CREATE TABLE tableRepTest (id int primary key, info text, value int);
     ```

  3. Insert some initial data (5 rows)

     ```
     INSERT INTO tableRepTest VALUES (1, 'initial data 1', 1),
     (2, 'initial data 1', 1), (3, 'initial data 1', 1),
     (4, 'initial data 1', 1), (5, 'initial data 1', 1);
     ```

4. Confirm data was inserted

```
SELECT * FROM tableRepTest;
```

5. Create the pglogical node

```
SELECT pglogical.create_node(
node_name := 'node1',
dsn := 'host=<hostname>.postgres.database.azure.com port=5432 sslmode=require dbname=<database> user=
```

6. Create a pglogical replication set – the following command adds all tables from the "public" schema to the set

```
SELECT pglogical.replication_set_add_all_tables('default', ARRAY['public']);
```

- **Set up the first Subscriber - we will call this "node2."**

1. Create the pglogical extension

```
CREATE EXTENSION pglogical;
```

2. Create the table we want to replicate (same as on node1)

```
CREATE TABLE tableRepTest (id int primary key, info text, value int);
```

3. Create the pglogical node

```
SELECT pglogical.create_node(
node_name := 'node2',
dsn := 'host=<hostname>.postgres.database.azure.com port=5432 sslmode=require dbname=<database> user=
```

4. Create the subscription on node2, subscribing to node1. "**synchronize_data**" is true so we will do an initial data load!

```
SELECT pglogical.create_subscription(
subscription_name := 'node2_sub',
provider_dsn := 'host=<hostname>.postgres.database.azure.com port=5432 sslmode=require dbname=<datab
replication_sets := ARRAY['default'],
synchronize_data := true,
forward_origins := '{}' );
```

5. Query the table – you should see data!

```
SELECT * FROM tableRepTest;
```

Second: set up bidirectional (two-way) replication

- On the original subscriber (node2), create a pglogical replication set – the following command adds all tables from the "public" schema to the set

```
SELECT pglogical.replication_set_add_all_tables('default', ARRAY['public']);
```

- On the original publisher (node1), create the subscription on the original Publisher (node1) subscribing to node2

```
SELECT pglogical.create_subscription(
 subscription_name := 'node1_sub',
 provider_dsn := 'host=<hostname>.postgres.database.azure.com port=5432 sslmode=require dbname=<database>
 replication_sets := ARRAY['default'],
 synchronize_data := false, -- "false" here does NOT do an initial data load since we don't need this now
 forward_origins := '{}' );
```

- Insert data from node2

```
INSERT INTO tableRepTest VALUES (12, 'manual insert from node 2', 1);
```

- Check the data in the table on node1, we can see it was replicated back:

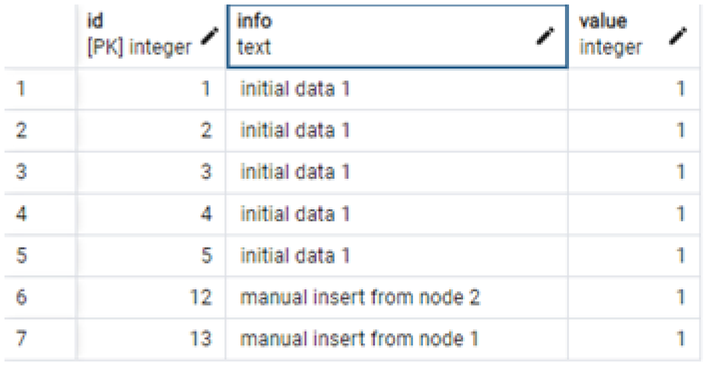
```
SELECT * FROM tableRepTest;
```

- Back on node 1 we can also insert some data:

```
INSERT INTO tableRepTest VALUES (13, 'manual insert from node 1', 1);
```

- Check the data in the table on node2, we can see it's all here!

```
SELECT * FROM tableRepTest;
```



	id [PK] integer	info text	value integer
1	1	initial data 1	1
2	2	initial data 1	1
3	3	initial data 1	1
4	4	initial data 1	1
5	5	initial data 1	1
6	12	manual insert from node 2	1
7	13	manual insert from node 1	1

Cleanup

- On Node1:

```
SELECT pglogical.drop_subscription(node_name := 'node1_sub');
SELECT pglogical.drop_node(node_name := 'node1');
```

- On Node2:

```
SELECT pglogical.drop_subscription(node_name := 'node2_sub');
SELECT pglogical.drop_node(node_name := 'node2');
```

Diagnostic queries

- Check pglogical replication status

```
select subscription_name, status FROM pglogical.show_subscription_status();
```

- Resynch if needed – shouldn't be needed as we do an initial sync

```
SELECT pglogical.alter_subscription_resynchronize_table(subscription_name := 'node2_sub', relation := 'ta
```