pglogical

Release 2.4.1

EnterpriseDB Corporation

Apr 05, 2023

CONTENTS

1 Release Notes **3**
- 1.1 pglogical 2.4.1 . . . 3
  - 1.1.1 Changes . . . 3
- 1.2 pglogical 2.4.0 . . . 3
  - 1.2.1 Changes . . . 3

2 Requirements **5**

3 Installation **7**
- 3.1 Packages . . . 7
  - 3.1.1 Installing pglogical with YUM . . . 7
  - 3.1.2 Pre-requisites . . . 7
  - 3.1.3 Installation . . . 8
  - 3.1.4 Installing pglogical with APT . . . 8
  - 3.1.5 Pre-requisites . . . 9
  - 3.1.6 Installation . . . 9
- 3.2 From source code . . . 9

4 Usage **11**
- 4.1 Quick setup . . . 11
- 4.2 Creating subscriber nodes with base backups . . . 12
- 4.3 Node management . . . 12
- 4.4 Subscription management . . . 13
- 4.5 Replication sets . . . 16
  - 4.5.1 Automatic assignment of replication sets for new tables . . . 18
- 4.6 Additional functions . . . 18
- 4.7 Row Filtering . . . 19
  - 4.7.1 Row Filtering on Provider . . . 19
  - 4.7.2 Row Filtering on Subscriber . . . 19

5 Synchronous Replication **21**

6 Conflicts **23**

7 Configuration options **25**

8 Limitations and restrictions **27**
- 8.1 Superuser is required . . . 27
- 8.2 `UNLOGGED` and `TEMPORARY` not replicated . . . 27
- 8.3 One database at a time . . . 27
- 8.4 PRIMARY KEY or REPLICA IDENTITY required . . . 27

8.5 Only one unique index/constraint/PK . . . 27
8.6 Unique constraints must not be deferrable . . . 28
8.7 DDL . . . 28
8.8 No replication queue flush . . . 28
8.9 FOREIGN KEYS . . . 28
8.10 TRUNCATE . . . 29
8.11 Sequences . . . 29
8.12 Triggers . . . 29
8.13 PostgreSQL Version differences . . . 29
8.14 Database encoding differences . . . 30
8.15 Large objects . . . 30
8.16 Postgres-XL . . . 30

The pglogical 2 extension provides logical streaming replication for PostgreSQL, using a publish/subscribe model. It is based on technology developed as part of EDB Postgres Distributed .

We use the following terms to describe data streams between nodes, deliberately reused from the earlier Slony technology:

- Nodes - PostgreSQL database instances
- Providers and Subscribers - roles taken by Nodes
- Replication Set - a collection of tables

pglogical is utilising the latest in-core features, so we have these version restrictions:

- Provider & subscriber nodes must run PostgreSQL 9.4+
- PostgreSQL 9.5+ is required for replication origin filtering and conflict detection
- Additionally, subscriber can be Postgres-XL 9.5+

Use cases supported are:

- Upgrades between major versions (given the above restrictions)
- Full database replication
- Selective replication of sets of tables using replication sets
- Selective replication of table rows at either publisher or subscriber side (row_filter)
- Selective replication of table columns at publisher side
- Data gather/merge from multiple upstream servers

Architectural details:

- pglogical works on a per-database level, not whole server level like physical streaming replication
- One Provider may feed multiple Subscribers without incurring additional disk write overhead
- One Subscriber can merge changes from several origins and detect conflict between changes with automatic and configurable conflict resolution (some, but not all aspects required for multi-master).
- Cascading replication is implemented in the form of changeset forwarding.

CHAPTER

ONE

RELEASE NOTES

1.1 pglogical 2.4.1

Version 2.4.1 is a maintenance release of pglogical 2.

1.1.1 Changes

- Fix "snapshot still active" warnings and snapshot handling with domains. Problems introduced in version 2.4.0.
- Flush error state when having failed to drop remote replication slot

1.2 pglogical 2.4.0

Version 2.4.0 is a maintenance release of pglogical 2.

1.2.1 Changes

- Add preliminary support for PostgreSQL 14 (beta)
- Fix pglogical_show_subscription_table to return row rather than set of rows
- Fix snapshot handling in output plugin and apply worker
- Fix command quoting on Windows so that it actually works Seems like the API used before has 1024 limit on command line length
- Make sure that the schema syncing process can be interrupted on Windows
- Fix compilation issues with pglogical_create_subscriber on Windows
- Fix double closing of relation when a BEFORE ROW DELETE trigger returns NULL in the apply worker
- Fix multi-insert crash in the SPI apply worker
- Fix multi-insert doing insert of last tuple multiple times in apply worker
- Make sure debug_query_string is always set Newer versions of PostgreSQL require that debug_query_string is always set.

CHAPTER

TWO

REQUIREMENTS

To use pglogical the provider and subscriber must be running PostgreSQL 9.4 or newer.

The `pglogical` extension must be installed on both provider and subscriber. You must `CREATE EXTENSION pglogical` on both.

Tables on the provider and subscriber must have the same names and be in the same schema. Future revisions may add mapping features.

Tables on the provider and subscriber must have the same columns, with the same data types in each column. `CHECK` constraints, `NOT NULL` constraints, etc., must be the same or weaker (more permissive) on the subscriber than the provider.

Tables must have the same `PRIMARY KEY` s. It is not recommended to add additional `UNIQUE` constraints other than the `PRIMARY KEY` (see below).

Some additional requirements are covered in "Limitations and Restrictions", below.

CHAPTER

THREE

INSTALLATION

3.1 Packages

pglogical is available as RPMs via yum for Fedora, CentOS, & RHEL, and as DEBs via apt for Debian and Ubuntu, or as source code here. Please see below for instructions on installing from source.

3.1.1 Installing pglogical with YUM

The instructions below are valid for Red Hat family of operating systems (RHEL, CentOS, Fedora). Pre-Requisites

3.1.2 Pre-requisites

These RPMs all require the PGDG PostgreSQL releases from http://yum.postgresql.org/. You cannot use them with stock PostgreSQL releases included in Fedora and RHEL. If you don't have PostgreSQL already:

- Install the appropriate PGDG repo rpm from http://yum.postgresql.org/repopackages.php
- Install PostgreSQL
- PostgreSQL 9.4: `yum install postgresql94-server postgresql94-contrib`
- PostgreSQL 9.5: `yum install postgresql95-server postgresql95-contrib`
- PostgreSQL 9.6: `yum install postgresql96-server postgresql96-contrib`
- PostgreSQL 10: `yum install postgresql10-server postgresql10-contrib`
- PostgreSQL 11: `yum install postgresql11-server postgresql11-contrib`
- PostgreSQL 12: `yum install postgresql12-server postgresql12-contrib`
- PostgreSQL 13: `yum install postgresql13-server postgresql13-contrib`
- PostgreSQL 14: `yum install postgresql14-server postgresql14-contrib`

Then install the "2ndQuadrant's General Public" repository for your PostgreSQL version, by running the following instructions as root on the destination Linux server:

- PostgreSQL 9.4: `curl https://techsupport.enterprisedb.com/api/repository/dl/default/release/9.4/rpm | bash`
- PostgreSQL 9.5: `curl https://techsupport.enterprisedb.com/api/repository/dl/default/release/9.5/rpm | bash`
- PostgreSQL 9.6: `curl https://techsupport.enterprisedb.com/api/repository/dl/default/release/9.6/rpm | bash`

- PostgreSQL 10: `curl https://techsupport.enterprisedb.com/api/repository/dl/default/release/10/rpm | bash`
- PostgreSQL 11: `curl https://techsupport.enterprisedb.com/api/repository/dl/default/release/11/rpm | bash`
- PostgreSQL 12: `curl https://techsupport.enterprisedb.com/api/repository/dl/default/release/12/rpm | bash`
- PostgreSQL 13: `curl https://techsupport.enterprisedb.com/api/repository/dl/default/release/13/rpm | bash`
- PostgreSQL 14: `curl https://techsupport.enterprisedb.com/api/repository/dl/default/release/14/rpm | bash`

3.1.3 Installation

Once the repository is installed, you can proceed to pglogical for your PostgreSQL version:

- PostgreSQL 9.4: `yum install postgresql94-pglogical`
- PostgreSQL 9.5: `yum install postgresql95-pglogical`
- PostgreSQL 9.6: `yum install postgresql96-pglogical`
- PostgreSQL 10: `yum install postgresql10-pglogical`
- PostgreSQL 11: `yum install postgresql11-pglogical`
- PostgreSQL 12: `yum install postgresql12-pglogical`
- PostgreSQL 13: `yum install postgresql13-pglogical`
- PostgreSQL 14: `yum install postgresql14-pglogical`

You may be prompted to accept the repository GPG key for package signing:

```
Retrieving key from file:///etc/pki/rpm-gpg/RPM-GPG-KEY-2NDQ-DL-DEFAULT Importing GPG␣
↪key 0xD6BAF0C3: Userid : "Public repository signing key 2ndQuadrant <ci@2ndquadrant.
↪com>" Fingerprint: 8565 305c ea7d 0b66 4933 d250 9904 cd4b d6ba f0c3 From : /etc/pki/
↪rpm-gpg/RPM-GPG-KEY-2NDQ-DL-DEFAULT Is this ok [y/N]:
```

If so, accept the key (if it matches the above) by pressing 'y' then enter. (It's signed by the 2ndQuadrant master packaging key, if you want to verify that.)

3.1.4 Installing pglogical with APT

The instructions below are valid for Debian and all Linux flavors based on Debian (e.g. Ubuntu).

3.1.5 Pre-requisites

You can install the "2ndQuadrant's General Public" repository by running the following instructions as root on the destination Linux server: `curl https://techsupport.enterprisedb.com/api/repository/dl/default/release/deb | bash`

- Add the http://apt.postgresql.org/ repository. See the site for instructions.

3.1.6 Installation

Once pre-requisites are complete, installing pglogical is simply a matter of executing the following for your version of PostgreSQL:

- PostgreSQL 9.4: `sudo apt-get install postgresql-9.4-pglogical`
- PostgreSQL 9.5: `sudo apt-get install postgresql-9.5-pglogical`
- PostgreSQL 9.6: `sudo apt-get install postgresql-9.6-pglogical`
- PostgreSQL 10: `sudo apt-get install postgresql-10-pglogical`
- PostgreSQL 11: `sudo apt-get install postgresql-11-pglogical`
- PostgreSQL 12: `sudo apt-get install postgresql-12-pglogical`
- PostgreSQL 13: `sudo apt-get install postgresql-13-pglogical`
- PostgreSQL 14: `sudo apt-get install postgresql-14-pglogical`

3.2 From source code

Source code installs are the same as for any other PostgreSQL extension built using PGXS.

Make sure the directory containing `pg_config` from the PostgreSQL release is listed in your `PATH` environment variable. You might have to install a `-dev` or `-devel` package for your PostgreSQL release from your package manager if you don't have `pg_config` .

Then run `make` to compile, and `make install` to install. You might need to use `sudo` for the install step.

e.g. for a typical Fedora or RHEL 7 install, assuming you're using the yum.postgresql.org packages for PostgreSQL:

```
sudo dnf install postgresql95-devel
PATH=/usr/pgsql-9.5/bin:$PATH make clean all
sudo PATH=/usr/pgsql-9.5/bin:$PATH make install
```

CHAPTER

FOUR

USAGE

This section describes basic usage of the pglogical replication extension.

4.1 Quick setup

First the PostgreSQL server has to be properly configured to support logical decoding:

```
wal_level = logical
max_worker_processes = 10   # one per database needed on provider node
                            # one per node needed on subscriber node
max_replication_slots = 10  # one per node needed on provider node
max_wal_senders = 10        # one per node needed on provider node
shared_preload_libraries = pglogical
```

If you are using PostgreSQL 9.5+ (this won't work on 9.4) and want to handle conflict resolution with last/first update wins (see

Conflicts), you can add this additional option to postgresql.conf:

```
track_commit_timestamp = on # needed for last/first update wins conflict resolution
                            # property available in PostgreSQL 9.5+
```

`pg_hba.conf` has to allow logical replication connections from localhost. Up until PostgreSQL 9.6, logical replication connections are managed using the `replication` keyword in `pg_hba.conf` . In PostgreSQL 10 and later, logical replication connections are treated by `pg_hba.conf` as regular connections to the provider database.

Next the `pglogical` extension has to be installed on all nodes:

```
CREATE EXTENSION pglogical;
```

If using PostgreSQL 9.4, then the `pglogical_origin` extension also has to be installed on that node:

```
CREATE EXTENSION pglogical_origin;
```

Now create the provider node:

```
SELECT pglogical.create_node(
    node_name := provider1,
    dsn := host=providerhost port=5432 dbname=db
);
```

Add all tables in `public` schema to the `default` replication set.

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

Optionally you can also create additional replication sets and add tables to them (see *Replication sets*).

It's usually better to create replication sets before subscribing so that all tables are synchronized during initial replication setup in a single initial transaction. However, users of bigger databases may instead wish to create them incrementally for better control.

Once the provider node is setup, subscribers can be subscribed to it. First the subscriber node must be created:

```
SELECT pglogical.create_node(
    node_name := subscriber1,
    dsn := host=thishost port=5432 dbname=db
);
```

And finally on the subscriber node you can create the subscription which will start synchronization and replication process in the background:

```
SELECT pglogical.create_subscription(
    subscription_name := subscription1,
    provider_dsn := host=providerhost port=5432 dbname=db
);

SELECT pglogical.wait_for_subscription_sync_complete(subscription1);
```

4.2 Creating subscriber nodes with base backups

In addition to the SQL-level node and subscription creation, pglogical also supports creating a subscriber by cloning the provider with `pg_basebackup` and starting it up as a pglogical subscriber. This is done with the `pglogical_create_subscriber` tool; see the `--help` output.

Unlike `pglogical.create_subscription` 's data sync options, this clone ignores replication sets and copies all tables on all databases. However, it's often much faster, especially over high-bandwidth links.

4.3 Node management

Nodes can be added and removed dynamically using the SQL interfaces.

- `pglogical.create_node(node_name name, dsn text)` Creates a node.

Parameters:

- `node_name` - name of the new node, only one node is allowed per database
- `dsn` - connection string to the node, for nodes that are supposed to be providers, this should be reachable from outside
- `pglogical.drop_node(node_name name, ifexists bool)` Drops the pglogical node.

Parameters:

- `node_name` - name of an existing node
- `ifexists` - if true, error is not thrown when subscription does not exist, default is false

- `pglogical.alter_node_add_interface(node_name name, interface_name name, dsn text)` Adds additional interface to a node.

When node is created, the interface for it is also created with the `dsn` specified in the `create_node` and with the same name as the node. This interface allows adding alternative interfaces with different connection strings to an existing node.

Parameters:

- `node_name` - name of an existing node
- `interface_name` - name of a new interface to be added
- `dsn` - connection string to the node used for the new interface
- `pglogical.alter_node_drop_interface(node_name name, interface_name name)` Remove existing interface from a node.

Parameters:

- `node_name` - name of and existing node
- `interface_name` - name of an existing interface

4.4 Subscription management

`pglogical.create_subscription(subscription_name name, provider_dsn text, replication_sets text[], synchronize_structure boolean, synchronize_data boolean, forward_origins text[], apply_delay interval)` Creates a subscription from current node to the provider node. Command does not block, just initiates the action.

Parameters:

- `subscription_name` - name of the subscription, must be unique
- `provider_dsn` - connection string to a provider
- `replication_sets` - array of replication sets to subscribe to, these must already exist, default is "{default,default_insert_only,ddl_sql}"
- `synchronize_structure` - specifies if to synchronize structure from provider to the subscriber, default false
- `synchronize_data` - specifies if to synchronize data from provider to the subscriber, default true
- `forward_origins` - array of origin names to forward, currently only supported values are empty array meaning don't forward any changes that didn't originate on provider node (this is useful for two-way replication between the nodes), or "{all}" which means replicate all changes no matter what is their origin, default is "{all}"
- `apply_delay` - how much to delay replication, default is 0 seconds
- `force_text_transfer` - force the provider to replicate all columns using a text representation (which is slower, but may be used to change the type of a replicated column on the subscriber), default is false

The `subscription_name` is used as `application_name` by the replication connection. This means that it's visible in the `pg_stat_replication` monitoring view. It can also be used in `synchronous_standby_names` when pglogical is used as part of

> *Synchronous Replication* setup.

Use `pglogical.wait_for_subscription_sync_complete(sub_name)` to wait for the subscription to asynchronously start replicating and complete any needed schema and/or data sync.

- `pglogical.drop_subscription(subscription_name name, ifexists bool)` Disconnects the subscription and removes it from the catalog.

Parameters:

- `subscription_name` - name of the existing subscription
- `ifexists` - if true, error is not thrown when subscription does not exist, default is false
- `pglogical.alter_subscription_disable(subscription_name name, immediate bool)` Disables a subscription and disconnects it from the provider.

Parameters:

- `subscription_name` - name of the existing subscription
- `immediate` - if true, the subscription is stopped immediately, otherwise it will be only stopped at the end of current transaction, default is false
- `pglogical.alter_subscription_enable(subscription_name name, immediate bool)` Enables disabled subscription.

Parameters:

- `subscription_name` - name of the existing subscription
- `immediate` - if true, the subscription is started immediately, otherwise it will be only started at the end of current transaction, default is false
- `pglogical.alter_subscription_interface(subscription_name name, interface_name name)` Switch the subscription to use different interface to connect to provider node.

Parameters:

- `subscription_name` - name of an existing subscription
- `interface_name` - name of an existing interface of the current provider node
- `pglogical.alter_subscription_synchronize(subscription_name name, truncate bool)` All unsynchronized tables in all sets are synchronized in a single operation. Tables are copied and synchronized one by one. Command does not block, just initiates the action. Use `pglogical.wait_for_subscription_sync_complete` to wait for completion.

Parameters:

- `subscription_name` - name of the existing subscription
- `truncate` - if true, tables will be truncated before copy, default false
- `pglogical.alter_subscription_resynchronize_table(subscription_name name, relation regclass)`

Resynchronize one existing table. The table may not be the target of any foreign key constraints.

Does not block, use `pglogical.wait_for_table_sync_complete` to wait for completion.

Parameters:

- `subscription_name` - name of the existing subscription
- `relation` - name of existing table, optionally qualified

`pglogical.wait_for_subscription_sync_complete(subscription_name name)`

Wait for a subscription or to finish synchronization after a `pglogical.create_subscription` or `pglogical.alter_subscription_synchronize` .

This function waits until the subscription's initial schema/data sync, if any, are done, and until any tables pending individual resynchronisation have also finished synchronising.

For best results, run `SELECT pglogical.wait_slot_confirm_lsn(NULL, NULL)` on the provider after any replication set changes that requested resyncs, and only then call `pglogical.wait_for_subscription_sync_complete` on the subscriber.

`pglogical.wait_for_table_sync_complete(subscription_name name, relation regclass)`

Same as `pglogical.wait_for_subscription_sync_complete` , but waits only for the subscription's initial sync and the named table. Other tables pending resynchronisation are ignored.

- `pglogical.wait_slot_confirm_lsn`

`SELECT pglogical.wait_slot_confirm_lsn(NULL, NULL)`

Wait until all replication slots on the current node have replayed up to the xlog insert position at time of call on all providers. Returns when all slots' `confirmed_flush_lsn` passes the `pg_current_wal_insert_lsn()` at time of call.

Optionally may wait for only one replication slot (first argument). Optionally may wait for an arbitrary LSN passed instead of the insert lsn (second argument). Both are usually just left null.

This function is very useful to ensure all subscribers have received changes up to a certain point on the provider.

- `pglogical.show_subscription_status(subscription_name name)` Shows status and basic information about subscription.

Parameters:

`subscription_name` - optional name of the existing subscription, when no name was provided, the function will show status for all subscriptions on local node

- `pglogical.show_subscription_table(subscription_name name, relation regclass)`

Shows synchronization status of a table.

Parameters:

- `subscription_name` - name of the existing subscription
- `relation` - name of existing table, optionally qualified
- `pglogical.alter_subscription_add_replication_set(subscription_name name, replication_set name)`

Adds one replication set into a subscriber. Does not synchronize, only activates consumption of events.

Parameters:

- `subscription_name` - name of the existing subscription
- `replication_set` - name of replication set to add
- `pglogical.alter_subscription_remove_replication_set(subscription_name name, replication_set name)` Removes one replication set from a subscriber.

Parameters:

- `subscription_name` - name of the existing subscription
- `replication_set` - name of replication set to remove

There is also a `postgresql.conf` parameter, `pglogical.extra_connection_options` , that may be set to assign connection options that apply to all connections made by pglogical. This can be a useful place to set up custom keepalive options, etc.

pglogical defaults to enabling TCP keepalives to ensure that it notices when the upstream server disappears unexpectedly. To disable them add `keepalives = 0` to `pglogical.extra_connection_options` .

4.5 Replication sets

Replication sets provide a mechanism to control which tables in the database will be replicated and which actions on those tables will be replicated.

Each replicated set can specify individually if `INSERTs` , `UPDATEs` , `DELETEs` and `TRUNCATEs` on the set are replicated. Every table can be in multiple replication sets and every subscriber can subscribe to multiple replication sets as well. The resulting set of tables and actions replicated is the union of the sets the table is in. The tables are not replicated until they are added into a replication set.

There are three preexisting replication sets named "default", "default_insert_only" and "ddl_sql". The "default" replication set is defined to replicate all changes to tables in it. The "default_insert_only" only replicates INSERTs and is meant for tables that don't have primary key (see *Limitations and restrictions* section for details). The "ddl_sql" replication set is defined to replicate schema changes specified by `pglogical.replicate_ddl_command`

The following functions are provided for managing the replication sets:

- `pglogical.create_replication_set(set_name name, replicate_insert bool, replicate_update bool, replicate_delete bool, replicate_truncate bool)` This function creates a new replication set.

Parameters:

- `set_name` - name of the set, must be unique
- `replicate_insert` - specifies if `INSERT` is replicated, default true
- `replicate_update` - specifies if `UPDATE` is replicated, default true
- `replicate_delete` - specifies if `DELETE` is replicated, default true
- `replicate_truncate` - specifies if `TRUNCATE` is replicated, default true
- `pglogical.alter_replication_set(set_name name, replicate_inserts bool, replicate_updates bool, replicate_deletes bool, replicate_truncate bool)` This function changes the parameters of the existing replication set.

Parameters:

- `set_name` - name of the existing replication set
- `replicate_insert` - specifies if `INSERT` is replicated, default true
- `replicate_update` - specifies if `UPDATE` is replicated, default true
- `replicate_delete` - specifies if `DELETE` is replicated, default true
- `replicate_truncate` - specifies if `TRUNCATE` is replicated, default true
- `pglogical.drop_replication_set(set_name text)` Removes the replication set.

Parameters:

- `set_name` - name of the existing replication set
- `pglogical.replication_set_add_table(set_name name, relation regclass, synchronize_data boolean, columns text[], row_filter text)` Adds a table to replication set.

Parameters:

- `set_name` - name of the existing replication set
- `relation` - name or OID of the table to be added to the set
- `synchronize_data` - if true, the table data is synchronized on all subscribers which are subscribed to given replication set, default false
- `columns` - list of columns to replicate. Normally when all columns should be replicated, this will be set to NULL which is the default
- `row_filter` - row filtering expression, default NULL (no filtering), see *Row Filtering* for more info. **WARNING:Usecautionwhensynchronizingdatawithavalidrowfilter.** Using `synchronize_data=true` with a valid `row_filter` is like a one-time operation for a table. Executing it again with modified `row_filter` won't synchronize data to subscriber. Subscribers may need to call `pglogical.alter_subscription_resynchronize_table()` to fix it.
- `pglogical.replication_set_add_all_tables(set_name name, schema_names text[], synchronize_data boolean)` Adds all tables in given schemas. Only existing tables are added, table that will be created in future will not be added automatically. For how to ensure that tables created in future are added to correct replication set, see *Automatic assignment of replication sets for new tables* .

Parameters:

- `set_name` - name of the existing replication set
- `schema_names` - array of names name of existing schemas from which tables should be added
- `synchronize_data` - if true, the table data is synchronized on all subscribers which are subscribed to given replication set, default false
- `pglogical.replication_set_remove_table(set_name name, relation regclass)` Remove a table from replication set.

Parameters:

- `set_name` - name of the existing replication set
- `relation` - name or OID of the table to be removed from the set
- `pglogical.replication_set_add_sequence(set_name name, relation regclass, synchronize_data boolean)` Adds a sequence to a replication set.

Parameters:

- `set_name` - name of the existing replication set
- `relation` - name or OID of the sequence to be added to the set
- `synchronize_data` - if true, the sequence value will be synchronized immediately, default false
- `pglogical.replication_set_add_all_sequences(set_name name, schema_names text[], synchronize_data boolean)` Adds all sequences from the given schemas. Only existing sequences are added, any sequences that will be created in future will not be added automatically.

Parameters:

- `set_name` - name of the existing replication set
- `schema_names` - array of names name of existing schemas from which tables should be added
- `synchronize_data` - if true, the sequence value will be synchronized immediately, default false
- `pglogical.replication_set_remove_sequence(set_name name, relation regclass)` Remove a sequence from a replication set.

Parameters:

- `set_name` - name of the existing replication set
- `relation` - name or OID of the sequence to be removed from the set

You can view the information about which table is in which set by querying the `pglogical.tables` view.

4.5.1 Automatic assignment of replication sets for new tables

The event trigger facility can be used for describing rules which define replication sets for newly created tables.

Example:

```
CREATE OR REPLACE FUNCTION pglogical_assign_repset()
RETURNS event_trigger AS $$
DECLARE obj record;
BEGIN
    FOR obj IN SELECT * FROM pg_event_trigger_ddl_commands()
    LOOP
        IF obj.object_type = table THEN
            IF obj.schema_name = config THEN
                PERFORM pglogical.replication_set_add_table(configuration, obj.objid);
            ELSIF NOT obj.in_extension THEN
                PERFORM pglogical.replication_set_add_table(default, obj.objid);
            END IF;
        END IF;
    END LOOP;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER pglogical_assign_repset_trg
    ON ddl_command_end
    WHEN TAG IN (CREATE TABLE, CREATE TABLE AS)
    EXECUTE PROCEDURE pglogical_assign_repset();
```

The above example will put all new tables created in schema `config` into replication set `configuration` and all other new tables which are not created by extensions will go to `default` replication set.

4.6 Additional functions

- `pglogical.replicate_ddl_command(command text, replication_sets text[])` Execute locally and then send the specified command to the replication queue for execution on subscribers which are subscribed to one of the specified `replication_sets` .

Parameters:

- `command` - DDL query to execute
- `replication_sets` - array of replication sets which this command should be associated with, default "{ddl_sql}"
- `pglogical.synchronize_sequence(relation regclass)` Push sequence state to all subscribers. Unlike the subscription and table synchronization function, this function should be run on provider. It forces update of the tracked sequence state which will be consumed by all subscribers (replication set filtering still applies) once they replicate the transaction in which this function has been executed.

Parameters:

- `relation` - name of existing sequence, optionally qualified

4.7 Row Filtering

PGLogical allows row based filtering both on provider side and the subscriber side.

4.7.1 Row Filtering on Provider

On the provider the row filtering can be done by specifying `row_filter` parameter for the `pglogical.replication_set_add_table` function. The `row_filter` is normal PostgreSQL expression which has the same limitations on what's allowed as the `CHECK` constraint.

Simple `row_filter` would look something like `row_filter := 'id > 0'` which would ensure that only rows where values of `id` column is bigger than zero will be replicated.

It's allowed to use volatile function inside `row_filter` but caution must be exercised with regard to writes as any expression which will do writes will throw error and stop replication.

It's also worth noting that the `row_filter` is running inside the replication session so session specific expressions such as `CURRENT_USER` will have values of the replication session and not the session which did the writes.

4.7.2 Row Filtering on Subscriber

On the subscriber the row based filtering can be implemented using standard `BEFORE TRIGGER` mechanism.

It is required to mark any such triggers as either `ENABLE REPLICA` or `ENABLE ALWAYS` otherwise they will not be executed by the replication process.

CHAPTER

FIVE

SYNCHRONOUS REPLICATION

Synchronous replication is supported using same standard mechanism provided by PostgreSQL for physical replication.

The `synchronous_commit` and `synchronous_standby_names` settings will affect when `COMMIT` command reports success to client if pglogical subscription name is used in `synchronous_standby_names` . Refer to PostgreSQL documentation for more info about how to configure these two variables.

CHAPTER

SIX

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` setting.

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.

CHAPTER

SEVEN

CONFIGURATION OPTIONS

Some aspects of PGLogical can be configured using configuration options that can be either set in `postgresql.conf` or via `ALTER SYSTEM SET` .

- `pglogical.conflict_resolution` Sets the resolution method for any detected conflicts between local data and incoming changes.

Possible values:

```
-   `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 available settings and defaults depend on version of PostgreSQL and other settings.

The default value in PostgreSQL is `apply_remote` .

The `keep_local` , `last_update_wins` and `first_update_wins` settings require `track_commit_timestamp` PostgreSQL setting to be enabled. As `track_commit_timestamp` is not available in PostgreSQL 9.4 `pglogical.conflict_resolution` can only be `apply_remote` or `error` .

In Postgres-XL, the only supported value and the default is `error` .

- `pglogical.conflict_log_level` Sets the log level for reporting detected conflicts when the `pglogical.conflict_resolution` is set to anything else than `error` .

Main use for this setting is to suppress logging of conflicts.

Possible values are same as for `log_min_messages` PostgreSQL setting.

The default is `LOG` .

- `pglogical.batch_inserts` Tells PGLogical to use batch insert mechanism if possible. Batch mechanism uses PostgreSQL internal batch insert mode which is also used by `COPY` command.

The batch inserts will improve replication performance of transactions that did many inserts into one table. PGLogical will switch to batch mode when transaction did more than 5 INSERTs.

It's only possible to switch to batch mode when there are no `INSTEAD OF INSERT` and `BEFORE INSERT` triggers on the table and when there are no defaults with volatile expressions for columns of the table. Also the batch mode will only work when `pglogical.conflict_resolution` is set to `error` .

The default is `true` .

- `pglogical.use_spi` Tells PGLogical to use SPI interface to form actual SQL (`INSERT` , `UPDATE` , `DELETE`) statements to apply incoming changes instead of using internal low level interface.

This is mainly useful for Postgres-XL and debugging purposes.

The default in PostgreSQL is `false` .

This can be set to `true` only when `pglogical.conflict_resolution` is set to `error` . In this state, conflicts are not detected.

In Postgres-XL the default and only allowed setting is `true` .

- `pglogical.temp_directory` Defines system path where to put temporary files needed for schema synchronization. This path need to exist and be writable by user running Postgres.

Default is empty, which tells PGLogical to use default temporary directory based on environment and operating system settings.

CHAPTER

EIGHT

LIMITATIONS AND RESTRICTIONS

8.1 Superuser is required

Currently pglogical replication and administration requires superuser privileges. It may be later extended to more granular privileges.

8.2 `UNLOGGED` and `TEMPORARY` not replicated

`UNLOGGED` and `TEMPORARY` tables will not and cannot be replicated, much like with physical streaming replication.

8.3 One database at a time

To replicate multiple databases you must set up individual provider/subscriber relationships for each. There is no way to configure replication for all databases in a PostgreSQL install at once.

8.4 PRIMARY KEY or REPLICA IDENTITY required

`UPDATE` s and `DELETE` s cannot be replicated for tables that lack a `PRIMARY KEY` or other valid replica identity such as using an index, which must be unique, not partial, not deferrable, and include only columns marked NOT NULL. Replication has no way to find the tuple that should be updated/deleted since there is no unique identifier. `REPLICA IDENTITY FULL` is not supported yet.

8.5 Only one unique index/constraint/PK

If more than one upstream is configured or the downstream accepts local writes then only one `UNIQUE` index should be present on downstream replicated tables. Conflict resolution can only use one index at a time so conflicting rows may `ERROR` if a row satisfies the `PRIMARY KEY` but violates a `UNIQUE` constraint on the downstream side. This will stop replication until the downstream table is modified to remove the violation.

It's fine to have extra unique constraints on an upstream if the downstream only gets writes from that upstream and nowhere else. The rule is that the downstream constraints must *not be more restrictive* than those on the upstream(s).

Partial secondary unique indexes are permitted, but will be ignored for conflict resolution purposes.

8.6 Unique constraints must not be deferrable

On the downstream end pglogical does not support index-based constraints defined as `DEFERRABLE` . It will emit the error

```
ERROR: pglogical doesnt support index rechecks needed for deferrable indexes
DETAIL: relation "public"."test_relation" has deferrable indexes: "index1", "index2"
```

if such an index is present when it attempts to apply changes to a table.

8.7 DDL

Automatic DDL replication is not supported. Managing DDL so that the provider and subscriber database(s) remain compatible is the responsibility of the user.

pglogical provides the `pglogical.replicate_ddl_command` function to allow DDL to be run on the provider and subscriber at a consistent point.

8.8 No replication queue flush

There's no support for freezing transactions on the master and waiting until all pending queued xacts are replayed from slots. Support for making the upstream read-only for this will be added in a future release.

This means that care must be taken when applying table structure changes. If there are committed transactions that aren't yet replicated and the table structure of the provider and subscriber are changed at the same time in a way that makes the subscriber table incompatible with the queued transactions replication will stop.

Administrators should either ensure that writes to the master are stopped before making schema changes, or use the `pglogical.replicate_ddl_command` function to queue schema changes so they're replayed at a consistent point on the replica.

Once multi-master replication support is added then using `pglogical.replicate_ddl_command` will not be enough, as the subscriber may be generating new xacts with the old structure after the schema change is committed on the publisher. Users will have to ensure writes are stopped on all nodes and all slots are caught up before making schema changes.

8.9 FOREIGN KEYS

Foreign keys constraints are not enforced for the replication process - what succeeds on provider side gets applied to subscriber even if the `FOREIGN KEY` would be violated.

8.10 TRUNCATE

Using `TRUNCATE ... CASCADE` will only apply the `CASCADE` option on the provider side.

(Properly handling this would probably require the addition of `ON TRUNCATE CASCADE` support for foreign keys in PostgreSQL).

`TRUNCATE ... RESTART IDENTITY` is not supported. The identity restart step is not replicated to the replica.

8.11 Sequences

The state of sequences added to replication sets is replicated periodically and not in real-time. Dynamic buffer is used for the value being replicated so that the subscribers actually receive future state of the sequence. This minimizes the chance of subscriber's notion of sequence's `last_value` falling behind but does not completely eliminate the possibility.

It might be desirable to call `synchronize_sequence` to ensure all subscribers have up to date information about given sequence after "big events" in the database such as data loading or during the online upgrade.

It's generally recommended to use `bigserial` and `bigint` types for sequences on multi-node systems as smaller sequences might reach end of the sequence space fast.

Users who want to have independent sequences on provider and subscriber can avoid adding sequences to replication sets and create sequences with step interval equal to or greater than the number of nodes. And then setting a different offset on each node. Use the `INCREMENT BY` option for `CREATE SEQUENCE` or `ALTER SEQUENCE` , and use `setval(...)` to set the start point.

8.12 Triggers

Apply process and the initial COPY process both run with `session_replication_role` set to `replica` which means that `ENABLE REPLICA` and `ENABLE ALWAYS` triggers will be fired.

8.13 PostgreSQL Version differences

PGLogical can replicate across PostgreSQL major versions. Despite that, long term cross-version replication is not considered a design target, though it may often work. Issues where changes are valid on the provider but not on the subscriber are more likely to arise when replicating across versions.

It is safer to replicate from an old version to a newer version since PostgreSQL maintains solid backward compatibility but only limited forward compatibility. Initial schema synchronization is only supported when replicating between same version of PostgreSQL or from lower version to higher version.

Replicating between different minor versions makes no difference at all.

8.14 Database encoding differences

PGLogical does not support replication between databases with different encoding. We recommend using `UTF-8` encoding in all replicated databases.

8.15 Large objects

PostgreSQL's logical decoding facility does not support decoding changes to large objects, so pglogical cannot replicate large objects.

8.16 Postgres-XL

Minimum supported version of Postgres-XL is 9.5r1.5.

Postgres-XL is only supported as subscriber (cannot be a provider). For workloads with many small transactions the performance of replication may suffer due to increased write latency. On the other hand large insert (or bulkcopy) transactions are heavily optimized to work very fast with Postgres-XL.

Also any DDL limitations apply so extra care need to be taken when using `replicate_ddl_command()` .

Postgres-XL changes defaults and available settings for `pglogical.conflict_resolution` and `pglogical.use_spi` configuration options.