Information Classification: **PARTNER CONFIDENTIAL**

2ndQuadrant®
PostgreSQL

2ndQuadrant
pglogical

Version 3.6.33
10 November 2022

pglogical Development Team

Contents

pglogical 3 **9**
Table of Contents 10

Nodes **11**
Node information 11
pglogical.local_node 11
pglogical.node 11
pglogical.node Columns 11
pglogical.node_interface 11
pglogical.node_interface Columns 11
Node management 11
pglogical.create_node 12
pglogical.drop_node 12
pglogical.alter_node_add_interface 12
pglogical.alter_node_drop_interface 13

Replication sets **14**
Behavior of partitioned tables 14
Older versions of PostgreSQL 14
Replication set manipulation interfaces 14
pglogical.create_replication_set 15
pglogical.alter_replication_set 15
pglogical.drop_replication_set 16
pglogical.replication_set_add_table 16
pglogical.replication_set_add_all_tables 18
pglogical.replication_set_remove_table 18
pglogical.replication_set_add_sequence 18
pglogical.replication_set_add_all_sequences 19
pglogical.replication_set_remove_sequence 19
Automatic assignment of replication sets for new tables 20
Additional functions 20
pglogical.synchronize_sequence 20
Row Filtering on Provider 21
Writing safer row filters 21
Changing row filters 22

DDL Replication **23**
Replication set DDL filters manipulation interfaces 23
pglogical.replication_set_add_ddl 23
pglogical.replication_set_remove_ddl 24
Additional functions 24
pglogical.ddl_replication 24
pglogical.ddl_replication Columns 24

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- pglogical.replicate_ddl_command . . . 24
- Restrictions . . . 25
- Considerations with global objects . . . 25
- `pglogical.tables` . . . 26
  - `pglogical.tables` Columns . . . 26
- `pglogical.queue` . . . 26
- `pglogical.queue_truncate` . . . 26

Subscription Overview **27**
- Subscription information . . . 27
- pglogical.stat_subscription . . . 27
- `pglogical.stat_subscription` Columns . . . 27
- `pglogical.stat_relation` . . . 28
- `pglogical.stat_relation` Columns . . . 28
- pglogical.local_sync_status . . . 28
- pglogical.show_workers . . . 28
- SQL interfaces . . . 29
  - pglogical.create_subscription . . . 29
  - `pglogical_create_subscriber` . . . 30
  - pglogical.drop_subscription . . . 31
  - pglogical.alter_subscription_disable . . . 32
  - pglogical.alter_subscription_enable . . . 32
  - pglogical.alter_subscription_interface . . . 32
  - pglogical.alter_subscription_synchronize . . . 33
  - pglogical.alter_subscription_resynchronize_table . . . 33
  - pglogical.show_subscription_status . . . 34
  - pglogical.show_subscription_table . . . 34
  - pglogical.show_subscription_clock_drift . . . 35
  - pglogical.alter_subscription_add_replication_set . . . 35
  - pglogical.alter_subscription_remove_replication_set . . . 35
  - pglogical.wait_for_subscription_sync_complete . . . 36
  - pglogical.wait_for_table_sync_complete . . . 36
  - `pglogical.wait_slot_confirm_lsn(name, pg_lsn)` . . . 37
  - `pglogical.standby_wait_replay_upstream_lsn(pg_lsn)` . . . 38
  - pglogical.alter_subscription_skip_changes_upto . . . 38
  - pglogical.alter_subscription_writer_options . . . 40
  - pglogical.alter_subscription_set_conflict_resolver . . . 41

pglogical writer **44**
- Conflict handling . . . 44
  - Row versioning . . . 44
- Configuration options . . . 44
  - pglogical.conflict_log_level . . . 45
  - pglogical.conflict_ignore_redundant_updates . . . 45

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

pglogical.conflict_check_full_tuple . . . 45
pglogical.batch_inserts . . . 45
config.session_replication_role . . . 46
Restrictions . . . 46
Only one unique index/constraint/PK . . . 46
Deferrable unique indexes . . . 46
Foreign Keys . . . 46
TRUNCATE . . . 47
Triggers . . . 47

SPI writer **48**
Conflicts handling . . . 48
Conflict Logging . . . 48
pglogical.alter_subscription_add_log . . . 48
pglogical.alter_node_remove_log . . . 49
`pglogical.apply_log_summary` . . . 50
`pglogical.apply_log_summary` Columns . . . 50
pglogical.conflict_resolution_to_string . . . 50
pglogical.conflict_type_to_string . . . 50
Configuration options . . . 51
pglogical.conflict_resolution . . . 51
pglogical.batch_inserts . . . 51
Restrictions . . . 51
FOREIGN KEYS . . . 51
TRUNCATE . . . 51
Triggers . . . 52

PostgreSQL settings which affect pglogical **53**
pglogical specific settings . . . 54
`pglogical.synchronous_commit` . . . 54
`pglogical.track_subscription_apply` . . . 54
`pglogical.track_relation_apply` . . . 55
`pglogical.temp_directory` . . . 55
`pglogical.extra_connection_options` . . . 55
`pglogical.synchronize_failover_slot_names` . . . 55
`pglogical.synchronize_failover_slots_drop` . . . 56
`pglogical.synchronize_failover_slots_dsn` . . . 56
`pglogical.standby_slot_names` . . . 56
`pglogical.standby_slots_min_confirmed` . . . 57
`pglogical.writer_input_queue_size` . . . 57
`pglogical.writer_output_queue_size` . . . 57
`pglogical.min_worker_backoff_delay` . . . 57

Postgres-XL **59**

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Failover with pglogical3 60
Provider failover setup 60
Subscriber failover setup 62
Additional functions 63
pglogical.sync_failover_slots() 63
Legacy: Provider failover with pglogical2 using failover slots 63

Restrictions 65
Superuser is required 65
UNLOGGED and TEMPORARY not replicated 65
One database at a time 65
PRIMARY KEY or REPLICA IDENTITY required 65
DDL 65
Sequences 66
PostgreSQL Version differences 66
pglogical.pglogical_version 66
pglogical.pglogical_version_num 66
Database encoding differences 67
Large objects 67
Additional restrictions 67

Troubleshooting 68

Diagnostic views and relations 69
`pglogical.worker_error` 69
`pglogical.worker_tasks` 69
`pglogical.apply_log` and `pglogical.apply_log_summary` 69

Error handling in pglogical 70
Diagnosing and fixing errors 70
Common problems 70
Multiple data source issues 71

Credits and Licence 73

Appendix A: Release Notes for pglogical3 74
pglogical 3.6.33 74
Upgrades 74
Upgrades 74
pglogical 3.6.31 74
Resolved Issues 74
Upgrades 75
pglogical 3.6.30 75
Resolved Issues 75
Upgrades 75

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

pglogical 3.6.29 75
- Resolved Issues 75

pglogical 3.6.28 75
- Resolved Issues 76
- Improvements 76

pglogical 3.6.27 76
- Resolved Issues 76
- Improvements 76

pglogical 3.6.26 77
- Resolved Issues 77
- Other Changes 77
- Upgrades 77

pglogical 3.6.25 77
- Resolved Issues 78
- Other Changes 78

pglogical 3.6.24 78
- Resolved Issues 78

pglogical 3.6.23 78
- Resolved Issues 78
- Other Changes 79

pglogical 3.6.22 79
- Resolved Issues 79
- Improvements 80

pglogical 3.6.21 80
- Resolved Issues 80
- Improvements 81

pglogical 3.6.20 81
- Resolved Issues 81
- Improvements 82

pglogical 3.6.19 82
- Resolved Issues 82
- Improvements 82

pglogical 3.6.18 82
- Improvements 83
- Resolved Issues 83

pglogical 3.6.17 83
- Improvements 83
- Resolved Issues 83
- Support, Diagnostic and Logging Changes 84

pglogical 3.6.16 84

pglogical 3.6.15 84
- Resolved Issues 84
- Improvements 85
- Upgrades 85

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

pglogical 3.6.14 85
Resolved Issues 85
pglogical 3.6.12 86
Improvements 86
Resolved Issues 86
pglogical 3.6.11 86
Improvements 86
Resolved Issues 87
pglogical 3.6.10 87
Improvements 87
Resolved Issues 87
pglogical 3.6.9 88
Improvements 88
pglogical 3.6.8 88
Resolved Issues 88
pglogical 3.6.7.1 88
Resolved Issues 88
pglogical 3.6.7 88
Improvements 89
Resolved Issues 89
Upgrades 89
pglogical 3.6.6 89
Improvements 89
Resolved Issues 90
Upgrades 90
pglogical 3.6.5 90
Improvements 90
Resolved Issues 90
Upgrades 91
pglogical 3.6.4 91
New Features 91
Resolved Issues 91
pglogical 3.6.3 92
New Features 92
Resolved Issues 92
pglogical 3.6.2 92
New Features 92
Resolved Issues 93
pglogical 3.6.1 93
New Features 93
Resolved Issues 93
pglogical 3.6.0.1 93
Resolved Issues 94
pglogical 3.6.0 94

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

New Features 94
Resolved Issues 94
Other Improvements 94

Appendix B: Known Issues **95**

Appendix C: Libraries **96**
LLVM 96
OpenSSL 96
Original SSLeay Licence 97
PostgreSQL License 98

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

pglogical 3

The pglogical 3 extension provides logical streaming replication for PostgreSQL, using a publish/subscribe model. It is based on technology developed as part of the BDR project (https://www.2ndquadrant.com/en/resources/bdr/).

We use the following terms to describe data streams between nodes:

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

These terms have been deliberately reused from the earlier Slony technology.

pglogical is new technology utilizing 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 provider or subscriber side (row_filter)
- Selective replication of table columns at provider 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.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Table of Contents

- Configuration
- Nodes
- Replication sets
- Subscriptions
- pglogical writer
- SPI writer
- Postgres-XL
- DDL
- Restrictions
- Troubleshooting
- Credits

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Nodes

Each database that participates in pglogical replication must be represented by its own node. Each node must have a unique identifier.

Node information

pglogical.local_node

A view containing node information but only for the local node.

`pglogical.node`

This table lists all PGL nodes.

`pglogical.node` Columns

Name	Type	Description
node_id	oid	Id of the node
node_name	name	Name of the node

`pglogical.node_interface`

This is a view that elaborates the information in `pglogical.node`, showing the DSN and node interface information.

`pglogical.node_interface` Columns

Name	Type	Description
if_id	oid	Node Interface ID
if_name	name	Name of the node the interface is for
if_nodeid	oid	ID of the node
if_dsn	text	DSN of the node

Node management

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

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

pglogical.create_node

Creates a node.

Synopsis

```
pglogical.create_node(node_name name, dsn text)
```

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

Removes the node.

Synopsis

```
pglogical.drop_node(node_name name, ifexists bool)
```

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

Adds an interface to a node.

Synopsis

```
pglogical.alter_node_add_interface (
    node_name name,
    interface_name name,
    dsn text
)
```

When a 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.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

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

Remove an existing named interface from a node.

Synopsis

```
pglogical.alter_node_drop_interface(node_name name, interface_name name)
```

Parameters

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

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

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" replication set only replicates INSERTs and is meant for tables that don't have primary key (see Restrictions section for details). The "ddl_sql" replication set is defined to replicate schema changes specified by the `pglogical.replicate_ddl_command`.

Behavior of partitioned tables

From PostgreSQL 11 onwards, pglogical supports partitioned tables transparently. This means that a partitioned table can be added to a replication set and changes to any of the partitions will be replicated downstream.

The partitioning definition on the subscription side can be set up differently to the one on the provider. This means that one can also replicate a partitioned table to a single table, or a single table to a partitioned table, or a partitioned tabled to a differently'partitioned table (repartitioning).

It's also possible to add individual partitions to the replication set, in which case they will be replicated like regular tables (to the table of the same name as the partition on the downstream). This has some performance advantages in case the partitioning definition is same on both provider and subscriber, as the partitioning logic does not have to be executed.

Note: If the root-partitioned table is part of any replication set, memberships of individual partitions are ignored and only the membership of said root table will be taken into account.

Older versions of PostgreSQL

In PostgreSQL 10 and older, pglogical only allows the replication of partitions directly to other partitions. Which means the partitioned table itself cannot be added to a replication set and can't be target of replication on the subscriber either (one can't replicate a normal table to a partitioned table).

Replication set manipulation interfaces

The following functions are provided for managing the replication sets:

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

pglogical.create_replication_set

This function creates a new replication set.

Synopsis

```
pglogical.create_replication_set (
    set_name name,
    replicate_insert boolean,
    replicate_update boolean,
    replicate_delete boolean,
    replicate_truncate boolean,
    autoadd_tables boolean,
    autoadd_sequences boolean,
    autoadd_existing boolean
)
```

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
- `autoadd_tables` - specifies if newly created tables should be automatically added to the new replication set; default false
- `autoadd_sequences` - specifies if newly created sequences should be automatically added to the new replication set; default false
- `autoadd_existing` - this in combination with `autoadd_tables` or `autoadd_sequences` specifies if any existing tables and sequences should be added as well

The autoadd options will ignore tables that are in `information_schema` or `pg_catalog` schemas or are part of an extension.

The autoadd options will also allow automatic removal of tables from the replication set. So there will be no dependency check on replication membership when the table which is part of the autoadd replication set is being dropped.

If you want to replicate tables which are part of some extension, you still have to add them manually.

pglogical.alter_replication_set

This function changes the parameters of the existing replication set.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Synopsis

```
pglogical.alter_replication_set (
    set_name name,
    replicate_inserts boolean,
    replicate_updates boolean,
    replicate_deletes boolean,
    replicate_truncate boolean,
    autoadd_tables boolean,
    autoadd_sequences boolean
)
```

Parameters

- `set_name` - name of the existing replication set
- `replicate_insert` - specifies if `INSERT` is replicated
- `replicate_update` - specifies if `UPDATE` is replicated
- `replicate_delete` - specifies if `DELETE` is replicated
- `replicate_truncate` - specifies if `TRUNCATE` is replicated
- `autoadd_tables` - specifies if newly created tables should be automatically added to the new replication set
- `autoadd_sequences` - specifies if newly created sequences should be automatically added to the new replication set

If any of these replication set parameters is NULL (which is the default value if nothing else is specified), the current setting for that parameter will remain unchanged.

pglogical.drop_replication_set

Removes the replication set.

Synopsis

```
pglogical.drop_replication_set(set_name text)
```

Parameters

- `set_name` - name of the existing replication set

pglogical.replication_set_add_table

Adds a table to a specified existing replication set, optionally requesting resynchronization by subscribers.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Synopsis

```
pglogical.replication_set_add_table (
    set_name name,
    relation regclass,
    synchronize_data boolean,
    columns text[],
    row_filter text
)
```

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 On Provider for more info.

WARNING: Use caution when synchronizing data with a valid row filter. Using `synchronize_data=true` with a valid `row_filter` is like a one-time operation for a table. Executing it again with a modified `row_filter` won't synchronize data to subscriber. Subscribers may need to call `pglogical.alter_subscription_resynchronize_table()` to fix it.

Also, note that if `synchronize_data` is enabled, a synchronization request is scheduled on each subscriber and actioned asynchronously. Adding to the replication set *does not wait for synchronization to complete*.

To wait until the resync has completed, first, on the provider, run:

```
SELECT pglogical.wait_slot_confirm_lsn(NULL, NULL);
```

To ensure each subscriber has received the request, then on each subscriber run:

```
SELECT pglogical.wait_for_subscription_sync_complete('sub_name');
```

NOTE: There is currently no function to alter the row filter or columns of a table's replication set membership (RM#5960). However, you can use a *single transaction* to remove the table from the replication set and then re-add it with the desired row filter and column filter. Make sure to set `synchronize_data := false`. This provides a seamless transition from the old to the new membership and will not skip or lose any rows from concurrent transactions.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

pglogical.replication_set_add_all_tables

Adds all tables in given schemas.

Synopsis

```
pglogical.replication_set_add_all_tables (
    set_name name,
    schema_names text[],
    synchronize_data boolean
)
```

Only existing tables are added; any tables created later will not be added automatically. To see how to automatically add tables to the correct replication set at creation time, 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 the given replication set; default false

pglogical.replication_set_remove_table

Removes a table from a specified existing replication set.

Synopsis

```
pglogical.replication_set_remove_table(set_name name, relation regclass)
```

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

Adds a sequence to a replication set.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Synopsis

```
pglogical.replication_set_add_sequence (
    set_name name,
    relation regclass,
    synchronize_data boolean
)
```

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

Adds all sequences from the given schemas.

Synopsis

```
pglogical.replication_set_add_all_sequences (
    set_name name,
    schema_names text[],
    synchronize_data boolean
)
```

Only existing sequences are added; any sequences created later will not be added automatically.

Parameters

- `set_name` - name of the existing replication set
- `schema_names` - array of names 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

Remove a sequence from a replication set.

Synopsis

```
pglogical.replication_set_remove_sequence(set_name name, relation regclass)
```


Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

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.

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 the `default` replication set.

Additional functions

pglogical.synchronize_sequence

Push sequence state to all subscribers.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Synopsis

```
pglogical.synchronize_sequence(relation regclass)
```

Unlike the subscription and table synchronization function, this function should be run on the provider. It forces an 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

Row Filtering on Provider

On the provider side, row filtering can be done by specifying the `row_filter` parameter for the `pglogical.replication_set_add_table` function. The `row_filter` is a normal PostgreSQL expression with the same limitations as a `CHECK` constraint.

You can see which row filters are active in the `pglogical.tables` view.

The table's column(s) are exposed to the row filter as simple identifiers; there's no qualifier or namespace.

Unlike a CHECK constraint's body, the row-filter is passed as a string which is parsed and checked by pglogical. So to avoid quoting issues you should use PostgreSQL's dollar-quoting, like this:

```
SELECT pglogical.replication_set_add_table(
    'setname', 'tblname'::regclass,
    synchronize_data := false,
    row_filter := $FILTER$ id > 0 $FILTER$
  );
```

A simple `row_filter` would look something like `row_filter := 'id > 0'` which would replicate only those rows where values of column `id` are greater than zero. This *will not affect any already-committed rows pending replication, or any already-replicated rows*.

Important: Caveats apply when re-synchronizing tables with row filters using `replication_set_add_table`. See `pglogical.replication_set_add_table`.

Writing safer row filters

Be very cautious when writing row filter expressions, and keep them as simple as possible. If a row-filter expression raises an error during replication, it is generally necessary to drop and re-create the subscription, resynchronizing *all* tables, not just the table with the problem row-filter. So row filters should be simple and defensively written. A non-exhaustive list of rules for writing filters is that they:

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- *Should* be simple expressions wherever possible. Try to use only built-in PostgreSQL operators and `IMMUTABLE` functions if you can.
- *Must* avoid using any expression that could raise an `ERROR` at runtime, such as casting from `text` to a more strictly validated data type. They must tolerate any value that the table's constraints permit to appear in the table.
- *May* use `VOLATILE` or `STABLE` functions, but any functions must obey the same constraints as the filter expression itself.

E.g. you can call `random()` but not `txid_current()` or `my_audit_log_function()`.

- *May* call user-defined functions written in SQL, Pl/PgSQL, or (with care) C. Use of other languages is untested and not recommended. PL/PgSQL functions *must not* use `EXCEPTION` blocks, and may have other as-yet-undiscovered issues so their use is not recommended. Stick to SQL where possible.
- *Should not* attempt to access any tables. Only the column values should be used.

Direct use of subqueries in the row-filter expression is blocked.

It's possible to call a user-defined function within the filter, and that *can* access table contents. This is *not recommended* and may be subject to surprising behaviour. The function *must* only access tables in `pg_catalog.*` or tables marked with the `user_catalog_table=true` attribute. Accessing other tables will not raise an error, but may cause undefined behaviour, errors, or crashes.

- *Must never* attempt any write operation or anything that assigns a transaction-id. Similar to queries on a read-replica. Attempting writes will break replication.
- *May* safely use columns of the filtered table that are not part of the replication set's column list. Filtering happens on the provider side so non-replicated columns will have their values accessible. This lets you do things like pre-compute complex filter criteria in triggers.
- *Should not* rely on session state, since the `row_filter` is running inside the replication session. Session specific expressions such as `CURRENT_USER` will have values of the replication session and not the session which did the writes. The same is true for GUCs etc.

Changing row filters

To change a row-filter expression on a table, use a single transaction to remove the table from the replication set, then add it again with the new row filter expression. Do not specify data sync and make sure to explicitly repeat the set of replicated columns. You can check the `pglogical.tables` view for the old column set and row filter.

See `pglogical.replication_set_add_table`.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

DDL Replication

DDL replication in pglogical builds on the idea of Replication Sets, similarly to the table replication.

The main difference from table replication is that DDL replication does not replicate the result of the DDL but the statement itself.

To replicate DDL, a DDL replication filter has to be added to the replication set.

The DDL filter can specify a `command_tag` and `role_name` to allow replication of only some DDL statements. The `command_tag` is same as those used by EVENT TRIGGERs. The `role_name` is used for matching against the current role which is executing the command. Both `command_tag` and `role_name` are evaluated as regular expressions which are case sensitive.

Replication set DDL filters manipulation interfaces

The following functions are provided for managing the DDL replication filters using replication sets:

pglogical.replication_set_add_ddl

Adds a DDL replication filter to a replication set.

Synopsis

```
pglogical.replication_set_add_ddl (
    set_name name,
    ddl_filter_name text,
    command_tag text,
    role_name text
)
```

Parameters

- `set_name` - name of the existing replication set
- `ddl_filter_name` - name of the new DDL replication filter
- `command_tag` - regular expression for matching command tags
- `role_name` - regular expression for matching role name

The `command_tag` and `role_name` parameters can be set to `NULL` in which case they will match any command tag or role respectively. They are both regular expressions, so you can use patterns like `'CREATE.*'` or `'(CREATE|DROP).*'`.

The target object identity (oid, name, etc) are not exposed, so you cannot filter on them.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

pglogical.replication_set_remove_ddl

Remove a DDL replication filter from replication set.

Synopsis

```
pglogical.replication_set_remove_ddl(set_name name, ddl_filter_name text)
```

Parameters

- `set_name` - name of the existing replication set
- `ddl_filter_name` - name of the DDL replication filter to be removed from the set

Additional functions

`pglogical.ddl_replication`

This view lists ddl replication configuration as set up by current ddl_filters.

`pglogical.ddl_replication` Columns

Name	Type	Description
set_ddl_name	name	Name of DDL filter
set_ddl_tag	text	Which command tags it applies to (regular expression)
set_ddl_role	text	Which roles it applies to (regular expression)
set_name	name	Name of the replication set for which this filter is defined

pglogical.replicate_ddl_command

This function can be used to explicitly replicate a command as-is using the specified set of replication sets. The command will also be executed locally.

Synopsis

```
pglogical.replicate_ddl_command(command text, replication_sets text[])`
```

Parameters

- `command` - DDL query to execute

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- `replication_sets` - array of replication sets which this command should be associated with; default "{ddl_sql}"

Restrictions

When the DDL replication filter matches a DDL command, it will modify the `search_path` configuration parameter to include only system catalogs. This means that all the user objects referenced in the query have to be fully schema qualified. For example `CREATE TABLE foo...` will not work and has to be written as `CREATE SCHEMA public.foo....`

DDL that matches the DDL replication filter and does not comply with this requirement will fail with an error like this:

```
ERROR:  no schema has been selected to create in
```

The same restriction applies to any command executed using the `pglogical.replicate_ddl_command` function. The function call has the additional restriction that it cannot execute commands which need to be run outside of a transaction. Most notably `CREATE INDEX CONCURRENTLY` will fail if run using `pglogical.replicate_ddl_command` but will work via DDL replication sets.

Considerations with global objects

Because PostgreSQL has objects that exist within one database, objects shared by all databases, and objects that exist outside the catalogs, some care is required when you may potentially replicate a subset of DDL or replicate DDL from more than one database:

- pglogical can capture and replicate DDL that affects global objects like roles, users, groups, etc, but only if the commands are run in a database with pglogical ddl replication enabled. So it's easy to get into inconsistent states if you do something like `CREATE ROLE` in the `postgres` db then `ALTER ROLE` in the `my_pglogical_enabled`. The resulting captured DDL may not apply on the downstream, requiring a transaction to be skipped over or non-replicated DDL to be run on the downstream to create the object that's targeted by the replicated DDL.
- pglogical can also capture and replicate DDL that references global objects that may not exist on the other node(s), such as tablespaces and users/roles. So an `ALTER TABLE ... OWNER TO ...` can fail to apply if the role, a global object, does not exist on the downstream. You may have to create a dummy global object on the downstream or if absolutely necessary, skip some changes from the stream.
- DDL that references local paths like tablespaces may fail to apply on the other end if paths differ.

In general you should run all your DDL via your pglogical-enabled database, and ensure that all global objects *exist* on the provider and all subscribers. This may require the creation of dummy roles, dummy tablespaces, etc.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Information Classification: **PARTNER CONFIDENTIAL**

`pglogical.tables`

This view lists information about table membership in replication sets. If a table exists in multiple replication sets it will appear multiple times in this table.

`pglogical.tables` Columns

Name	Type	Description
relid	oid	The OID of the relation
nspname	name	Name of the schema relation is in
relname	name	Name of the relation
set_name	name	Name of the replication set
set_ops	text[]	List of replicated operations
rel_columns	text[]	List of replicated columns (NULL = all columns) (*)
row_filter	text	Row filtering expression

`pglogical.queue`

DDL can also be queued up with a message to state the replication information. This can be seen in ascending order, on this view.

`pglogical.queue_truncate`

A function that erases all the logging information of the view.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Subscription Overview

A subscription is the receiving side (or downstream) of the pglogical replication setup. Just like on the upstream, the subscription first needs local node to be created (see #Nodes).

Subscription information

pglogical.stat_subscription

Apply statistics for each subscription. Only contains data if the tracking is enabled.

`pglogical.stat_subscription` Columns

Column	Type	Description
sub_name	name	Name of the subscription
subid	oid	Oid of the subscription
nconnect	bigint	Number of times this subscription has connected upstream
ncommit	bigint	Number of commits this subscription did
ninsert	bigint	Number of inserts this subscription did
nupdate	bigint	Number of updates this subscription did
ndelete	bigint	Number of deletes this subscription did
ntruncate	bigint	Number of truncates this subscription did
nddl	bigint	Number of DDL operations this subscription has executed
shared_blks_hit	bigint	Total number of shared block cache hits by the subscription
shared_blks_read	bigint	Total number of shared blocks read by the subscription
shared_blks_dirtied	bigint	Total number of shared blocks dirtied by the subscription
shared_blks_written	bigint	Total number of shared blocks written by the subscription
blk_read_time	double precision	Total time the subscription spent reading blocks, in milliseconds (if `track_io_timing` is enabled, otherwise zero)
blk_write_time	double precision	Total time the subscription spent writing blocks, in milliseconds (if `track_io_timing` is enabled, otherwise zero)

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

`pglogical.stat_relation`

Apply statistics for each relation. Only contains data if the tracking is enabled and something was replicated for a given relation.

`pglogical.stat_relation` Columns

Column	Type	Description
nspname	name	Name of the relation's schema
relname	name	Name of the relation
relid	oid	OID of the relation
total_time	double precision	Total time spent processing replication for the relation
ninsert	bigint	Number of inserts replicated for the relation
nupdate	bigint	Number of updates replicated for the relation
ndelete	bigint	Number of deletes replicated for the relation
ntruncate	bigint	Number of truncates replicated for the relation
shared_blks_hit	bigint	Total number of shared block cache hits for the relation
shared_blks_read	bigint	Total number of shared blocks read for the relation
shared_blks_dirtied	bigint	Total number of shared blocks dirtied for the relation
shared_blks_written	bigint	Total number of shared blocks written for the relation
blk_read_time	double precision	Total time spent reading blocks for the relation, in milliseconds (if `track_io_timing` is enabled, otherwise zero)
blk_write_time	double precision	Total time spent writing blocks for the relation, in milliseconds (if `track_io_timing` is enabled, otherwise zero)
lock_acquire_time	double precision	Total time spent acquiring locks on the relation (if `pglogical.track_apply_lock_timing` is enabled, otherwise zero)

pglogical.local_sync_status

An updated view of the synchronization locally. Columns include subscription ID, sync status and kind.

pglogical.show_workers

A function to bring the user information of the worker PID, role and subscription ID.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

SQL interfaces

pglogical.create_subscription

Creates a subscription from the current node to the provider node. Command does not block, just initiates the action.

Synopsis

```
pglogical.create_subscription (
    subscription_name name,
    provider_dsn text,
    replication_sets text[],
    synchronize_structure boolean,
    synchronize_data boolean,
    forward_origins text[],
    strip_origins boolean,
    apply_delay interval,
    writer name,
    writer_options text[]
)
```

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 the synchronous replication setup.

Subscription setup is asynchronous. `pglogical.create_subscription` returns immediately, before the subscription is up and running. Use `pglogical.wait_for_subscription_sync_complete` to wait until the subscription is up and has completed any requested schema and/or data sync.

`synchronize_structure` internally uses `pg_dump` and `pg_restore` to copy schema definitions. If more than one upstream is being subscribed to, only use `synchronize_data` on the first one, because it cannot de-duplicate schema definitions.

`synchronize_structure` internally uses `COPY` to unload and load the data from the provider.

If both `synchronize_structure` and `synchronize_data` are used, take care to create table definitions, then copy data, and only create indexes etc. at the end.

Note: An alternative to `pglogical.create_subscription` is the `pglogical_create_subscriber` tool, which takes a `pg_basebackup` or uses a pre-existing streaming replica of the provider node and converts it into a new logical replica. It's often much faster where network bandwidth is sufficient, but cannot filter the initial dump to exclude some databases/tables/etc.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

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
- `strip_origins` - determines whether to remove origin names from forwarded data, making it look like the data originate from local node, and allowing to forward the data to a subscription in the same instance (default is "false" which keeps origin info). The negative effect is it makes it impossible to redirect the subscription to the first node.
- `writer` - which writer to use for writing the data from the replication stream. Available writers currently are `local`, `HeapWriter` and `SPIWriter`; the local is an alias that automatically selects either `HeapWriter` or `SPIWriter` based on the version of PostgreSQL being used.
- `writer_options` - writer-specific options as an array of keys and values

`pglogical_create_subscriber`

`pglogical_create_subscriber` isn't a SQL function, it's a standalone command that provides an alternative way to create a subscriber. By default it will take a `pg_basebackup` of the provider node and convert that into a `pglogical` subscriber.

This can be a lot faster than `pglogical.create_subscription` where network and disk bandwidth is sufficient. However, it cannot filter out individual tables or table subsets, and it copies all databases whether or not they are intended for use with pglogical. It does not respect replication sets for the initial data copy. Unlike `pglogical.create_subscription`, it copies indexes rather than rebuilding them on the subscriber side.

It may be necessary to specify a customized `postgresql.conf`and/or `pg_hba.conf` for the copied node. In particular, you *must* copy the provider's `postgresql.conf` and edit it to change the `port` if you plan on creating a subscriber on the same host, where the port number would otherwise conflict.

`pglogical_create_subscriber` may also be used to convert an existing, running streaming replica of the provider into a subscriber. This lets the user clone the provider using alternative methods like `pg_start_backup()`, `rsync`, and `pg_stop_backup()`, or from a SAN snapshot. This conversion is done automatically when the target data directory is non-empty and instead contains a suitable PostgreSQL streaming replica.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Synopsis

```
pglogical_create_subscriber [OPTION]...
```

Options

General Options

- `-D, --pgdata=DIRECTORY` - data directory to be used for new node; can be either empty/non-existing directory, or directory populated using pg_basebackup -X stream command
- `--databases` - optional list of databases to replicate
- `-n, --subscriber-name=NAME` - name of the newly created subscriber
- `--subscriber-dsn=CONNSTR` - connection string to the newly created subscriber
- `--provider-dsn=CONNSTR` - connection string to the provider
- `--replication-sets=SETS` - comma separated list of replication set names
- `--apply-delay=DELAY` - apply delay in seconds (by default 0)
- `--drop-slot-if-exists` - drop replication slot of conflicting name
- `-s, --stop` - stop the server once the initialization is done
- `-v` - increase logging verbosity
- `--extra-basebackup-args` - additional arguments to pass to pg_basebackup. Safe options: `-T, -c, --xlogdir/--waldir`

Configuration Files Override

- `--hba-conf` - path to the new pg_hba.conf
- `--postgresql-conf` - path to the new postgresql.conf

WARNING: pglogical will always overwrite the `recovery.conf`, this behavior will be fixed in the next release.

pglogical.drop_subscription

Disconnects the subscription and removes it from the catalog.

Synopsis

```
pglogical.drop_subscription (
    subscription_name name,
    ifexists bool
)
```


Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

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

Disables a subscription and disconnects it from the provider.

Synopsis

```
pglogical.alter_subscription_disable (
    subscription_name name,
    immediate bool
)
```

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 the current transaction; default is false

pglogical.alter_subscription_enable

Enables disabled subscription.

```
pglogical.alter_subscription_enable(subscription_name name, immediate bool)
```

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

Switch the subscription to use a different interface to connect to the provider node.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Synopsis

```
pglogical.alter_subscription_interface (
    subscription_name name,
    interface_name name
)
```

Parameters

- `subscription_name` - name of an existing subscription
- `interface_name` - name of an existing interface of the current provider node

pglogical.alter_subscription_synchronize

All unsynchronized tables in all sets are synchronized in a single operation.

Synopsis

```
pglogical.alter_subscription_synchronize (
    subscription_name name,
    truncate bool
)
```

Tables are copied and synchronized one by one. Command does not block, just initiates the action.

Use `pglogical.wait_for_subscription_sync_complete('sub_name')` to wait for the resynchronization to complete.

Parameters

- `subscription_name` - name of the existing subscription
- `truncate` - if true, tables will be truncated before copy; default false

pglogical.alter_subscription_resynchronize_table

Asynchronously resynchronize one existing table.

WARNING: This function will truncate the table first. The table will be visibly empty to transactions between when the resync is scheduled and when it completes.

Use `pglogical.wait_for_subscription_sync_complete('sub_name')` to wait for all pending resynchronizations to complete, or `pglogical.wait_for_table_sync_complete` for just the named table.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Synopsis

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

Parameters

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

pglogical.show_subscription_status

Shows status and basic information about a subscription.

```
pglogical.show_subscription_status (subscription_name name)
```

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

Shows the synchronization status of a table.

Synopsis

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

Parameters

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

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

pglogical.show_subscription_clock_drift

Shows clock drift between provider and subscriber.

On the subscriber at apply time, we track the commit timestamp received from the provider and the current local timestamp. When the above function is invoked, we generate a diff (interval) of these values. A negative value will indicate clock drift.

```
pglogical.show_subscription_clock_drift (subscription_name name)
```

Parameters

- `subscription_name` - optional name of the existing subscription; when no name is provided, the function will show clock drift information for all subscriptions on the local node

pglogical.alter_subscription_add_replication_set

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

Synopsis

```
pglogical.alter_subscription_add_replication_set (
    subscription_name name,
    replication_set name
)
```

Parameters

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

pglogical.alter_subscription_remove_replication_set

Removes one replication set from a subscriber.

Synopsis

```
 pglogical.alter_subscription_remove_replication_set (
    subscription_name name,
    replication_set name
)
```


Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Parameters

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

pglogical.wait_for_subscription_sync_complete

Wait on the subscriber side until the named subscription is fully synchronized. The function waits for both the initial schema and data syncs (if any) and any currently outstanding individual table resyncs.

To ensure that this function sees and waits for pending resynchronizations triggered by provider-side replication set changes, make sure to `pglogical.wait_slot_confirm_lsn(NULL, NULL)` on the provider after any replication set changes.

Synopsis

```
pglogical.wait_for_subscription_sync_complete(
  subscription_name name
)
```

Parameters

- `subscription_name` - name of the existing subscription to wait for

pglogical.wait_for_table_sync_complete

Same as `pglogical.wait_for_subscription_sync_complete`, except that it waits for the subscription to be synced and for exactly one named table, which must exist on the downstream. You can use this variant to wait for a specific table resync to complete while ignoring other pending resyncs.

Synopsis

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

Parameters

- `subscription_name` - name of the existing subscription to wait for
- `relid` - possibly schema-qualified relation name (cast to regclass if needed) for the relation to wait for sync completion of.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

`pglogical.wait_slot_confirm_lsn(name, pg_lsn)`

On a pglogical provider, wait for the specified replication slot(s) to pass all the requested WAL position.

Note that to wait for a subscriber this function should be called on the *provider*, not the subscriber.

Waits for one specified slot if named explicitly, or all logical slots that use the pglogical output plugin if the slot name is null.

If no position is supplied the current WAL write position on the Pg instance this function is called on is used.

No timeout is offered, use a `statement_timeout`.

This function can only wait for physical slots and for logical slots with output plugins other than 'pglogical' if specified as a single named slot argument.

For physical slots the LSN waited for is the `restart_lsn`, because physical slots don't have the same two-phase advance as logical slots and they have a NULL `confirmed_flush_lsn`. Because physical standbys guarantee durability (flush) before visibility (replay), if you want to ensure transactions are actually visible you should call `pglogical.standby_wait_replay_upstream_lsn` on the standby instead.

Waiting with default (null) position can cause delays on idle systems because the slot position may not advance until the next standby status update if there are no further txns to replay. If you can ensure there will be are no concurrent transactions you can instead capture `pg_current_wal_insert_lsn()` after the writes you are interested in but before you commit the transaction, then wait for that. Ideally commit would report the commit lsn, and you could wait for that, but Pg doesn't do that yet. Doing this may lead to waits ending prematurely if there are concurrent txns, so only do it on test harness setups that do only one thing at a time.

Synopsis

```
SELECT pglogical.wait_slot_confirm_lsn(
  slotname name,
  target_lsn pg_lsn
);
```

Typically it's sufficient to use:

```
SELECT pglogical.wait_slot_confirm_lsn(NULL, NULL);
```

to wait until all pglogical (and bdr3) subscriber replication slots' `confirmed_flush_lsns` have confirmed a successful flush to disk of all WAL that was written on the provider as of the start of the `pglogical.wait_slot_confirm_lsn` call.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Parameters

- `slotname` - name of the replication slot to wait for, or NULL for all pglogical slots
- `target_lsn` - xlog position to wait for slots to confirm, or NULL for current xlog insert location.

`pglogical.standby_wait_replay_upstream_lsn(pg_lsn)`

On a physical streaming replica (hot standby), wait for the standby to replay WAL from the upstream up to or past the specified lsn before returning.

Does not support an explicit timeout. Use a `statement_timeout`.

ERRORs if called on a non-standby, or when a standby is promoted while waiting.

Use this where you need to guarantee that changes are replayed and visible on a replica, not just safe on disk. The sender-side function `pglogical.wait_slot_confirm_lsn()` only ensures durability, not visibility, when applied to physical replicas, because there's no guarantee the flushed WAL is replayed and commits become visible before the flush position is reported to the upstream.

This is effectively a convenience function for a loop over `pg_last_wal_replay_lsn()` for use in testing.

pglogical.alter_subscription_skip_changes_upto

Because logical replication can replicate across versions, doesn't replicate global changes like roles, and can replicate selectively, sometimes the logical replication apply process can encounter an error and stop applying changes.

Wherever possible such problems should be fixed by making changes to the subscriber side. `CREATE`ing any missing table that's blocking replication, `CREATE` a needed role, `GRANT` a necessary permission, etc. But occasionally a problem can't be fixed that way and it may be necessary to skip entirely over a transaction.

There's no support in pglogical for skipping over only parts of a transaction, i.e. subscriber-side filtering. Changes are skipped as entire transactions, all or nothing. To decide where to skip to, use log output to find the commit LSN, per the example below, or peek the change stream with the logical decoding functions.

Unless a transaction only made one change, it's often necessary to manually apply the transaction's effects on the downstream side, so it's important to save the problem transaction whenever possible. See the example below.

It's possible to skip over changes without `pglogical.alter_subscription_skip_changes_upto` by using `pg_catalog.pg_logical_slot_get_binary_changes` to skip to the LSN of interest, so this is really a convenience function. It does do a faster skip; however, it may bypass some kinds of errors in logical decoding.

This function only works on disabled subscriptions.

The usual sequence of steps is:

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- identify the problem subscription and LSN of the problem commit
- disable the subscription
- save a copy of the transaction(s) using `pg_catalog.pg_logical_slot_peek_changes` *on the provider* (if possible)
- `pglogical.alter_subscription_skip_changes_upto` on the subscriber
- apply repaired or equivalent changes on the subscriber manually if necessary
- re-enable the subscription

WARNING: It's easy to make problems worse when using this function. Don't do anything unless you're really, really sure it's the only option.

Synopsis

```
pglogical.alter_subscription_skip_changes_upto(
  subname text,
  skip_upto_and_including pg_lsn
);
```

Example

Apply of a transaction is failing with an ERROR, and you've determined that lower-impact fixes such as changes to the subscriber side will not resolve this issue. You determine that you must skip the transaction.

In the error logs, find the commit record LSN to skip to, as in this artificial example:

```
ERROR:  55000: pglogical target relation "public.break_me" does not exist
CONTEXT:  during apply of INSERT in commit before 0/1B28848, xid 670 committed
                                 ^^^^^^^^^^^^^^^^^^^^^^^^^^
                                                  this LSN
at 2018-07-03 14:28:48.58659+08 (action #2) from node replorigin 1
```

and if needed use the `pglogical.subscriptions` view to map the origin back to a subscription name, e.g.:

```
SELECT subscription_name, slot_name
FROM pglogical.subscriptions s
WHERE replication_origin_id = 1
```

Next, disable the subscription so the apply worker doesn't try to connect to the replication slot:

```
SELECT pglogical.alter_subscription_disable('the_subscription');
```


Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Note that you cannot skip only parts of the transaction, it's all or nothing. So it's strongly recommended that you save a record of it by `COPY`ing it out on the provider side first, using the subscription's slot name (as obtained above).

```
\copy (SELECT * FROM pg_catalog.pg_logical_slot_peek_changes('the_slot_name',
    'the_target_lsn', NULL, 'min_proto_version', '1', 'max_proto_version', '1',
    'startup_params_format', '1', 'proto_format', 'json')
 TO 'transaction_to_drop.csv' WITH (FORMAT csv);
```

(Note that the example is broken into multiple lines for readability, but it should be issued in a single line because `\copy` does not support multi-line commands)

Now you can skip the change by changing "peek" to "get" above, but `pglogical.skip_changes_upto` does a faster skip that avoids decoding and outputting all the data:

```
SELECT pglogical.alter_subscription_skip_changes_upto('subscription_name',
    'the_target_lsn');
```

If necessary or desired, apply the same changes (or repaired versions of them) manually to the subscriber, using the dumped transaction contents as a guide.

Finally, re-enable the subscription:

```
SELECT pglogical.alter_subscription_enable('the_subscription');
```

pglogical.alter_subscription_writer_options

Change the writer options first addressed when `writer_name` and `writer_options` are clarified with pglogical.create_subscription.

Synopsis

```
pglogical.alter_subscription_writer_options(
subscription_name name,
writer_name name,
writer_options text[] = '{}'

);
```


Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Example

Find the subscription you want to alter and use that as the `subscription_name` and possibly the `writer_name` if chosen (shown first). Then the DML with the `writer_options` text array.

```
SELECT pglogical.alter_subscription_writer_options(sub_name, sub_name, '{}') FROM pglogical.sub
```

Grant all writer options to `writer_name` super; array has to be an even number of elements.

```
SELECT pglogical.alter_subscription_writer_options(sub_name, 'super', '{UPDATE,INSERT,DELETE,''
```

pglogical.alter_subscription_set_conflict_resolver

Change the conflict resolver of given conflict type for the given subscription.

Synopsis

```
pglogical.alter_subscription_set_conflict_resolver(
    sub_name text,
    conflict_type text,
    conflict_resolver text
    )
```

Conflict type can be one of:

- `insert_exists` - the row being inserted exists locally
- `update_differing` - the origin has updated a different version of row that the local has
- `update_missing` - the row being updated does not exist locally
- `delete_missing` - the row being deleted does not exist locally
- `update_origin_change` - the row being updated was updated on a different origin
- `target_table_missing` - the table corresponding to the change does not exist locally
- `target_column_missing` - the column being updated or inserted to does not exist locally
- `source_column_missing` - a column that exists locally is not available in the updated or inserted row replicated
- `update_recently_deleted` - the row being updated was deleted locally recently
- `delete_recently_updated` - the row being deleted was updated locally recently
- `update_pkey_exists` - the updated primary key exists locally
- `apply_error` - an error occured while applying the change locally
- `apply_error_trigger` - an error occured while firing a trigger locally after applying the change
- `apply_error_ddl` - an error occured during applying a DDL that was replicated
- `apply_error_dml` - an error occured while applying a DML that was

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Note that apply_error, apply_error_trigger, apply_error_ddl and apply_error_dml are never raised right now. They may be used in future.

Conflict resolver can be one of:

- error - the replication will stop on error if conflict is detected; manual action is then required for resolution.
- skip - keep the local version of the data and ignore the conflicting change that is coming from the remote node. This is same as keep_local which is now deprecated.
- update - always apply the upstream change that's conflicting with local data. This is same as apply_remote, which is now deprecated.
- update_if_newer - the version of data with the newest commit timestamp will be kept (this can be either the local or the remote version). This is same as last_update_wins which is now deprecated.
- update_if_older - the version of the data with the oldest timestamp will be kept (this can be either the local or the remote version). This is same as first_update_wins which is now deprecated.
- insert_or_skip - if the row being updated is missing and the downstream can verify that the updated row was none of the ones that exist the new row will be inserted. Otherwise the change will be skipped.
- insert_or_error - if the row being updated is missing and the downstream can verify that the updated row was none of the ones that exist the new row will be inserted. Otherwise the replication will stop on error.
- ignore - if the updated or inserted column is missing, it will be ignored while applying the upstream change
- ignore_or_error - if the updated or inserted column is missing, it will be ignored if it the new value is NULL. Otherwise replication will stop on error
- use_default_value - if a column is present locally but is not available on the source, a default value will be used for that column.

The available settings and defaults depend on the version of PostgreSQL and other settings.

The skip, update_if_newer and first_update_wins settings require the track_commit_timestamp PostgreSQL setting to be enabled. Those can not be used with PostgreSQL 9.4 as track_commit_timestamp is not available in there.

Some conflict resolvers can not be used with some conflict types e.g. resolver update_if_newer can not be used with conflict type target_table_missing. error is the only resolved available to handle conflict types apply_error, apply_error_trigger, apply_error_ddl, or apply_error_dml. The function throws an error when an incompatible resolver is used.

Example

Find the subscription you want to change the conflict resolver for and use that as the sub_name.

```
SELECT pglogical.alter_subscription_set_conflict_resolver(`sub_name`, 'insert_exists', 'update_
```


Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Changes the conflict resolver of conflict type `insert_exists` for subscription `sub_name` to `update_if_newer`. If the row specified by INSERT change on subscription `sub_name` already exists locally, out of the two rows, the one with the newest commit will be kept.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

pglogical writer

The pglogical writer (or HeapWriter) is the standard way of writing into a local PostgreSQL instance when using pglogical subscription. This is the default writer used when no writer is specified in `pglogical.create_subscription()`.

The pglogical writer is using low-level APIs to write the data into local tables and sequences. It supports conflict detection and resolution, has full support for `REPLICA IDENTITY`, invokes constraints with the exception of foreign keys (see Foreign Keys for details) and row triggers marked as `REPLICA` (see Triggers).

Changes are applied as the table owning-user, thus security concerns are similar to the use of triggers by table owners.

Conflict handling

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 using pglogical.alter_subscription_set_conflict_resolver().

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

Row versioning

To ease reasoning about different versions of a row, it can be helpful for it to carry a row version. PGLogical provides the helper trigger `pglogical.inc_row_version` to simplify this task. It requires a user provided integer column of any bitwidth (usually, `SMALLINT` is enough) and needs to be added to a table as follows (assuming a table `my_table` with an integer column `row_version`):

```
CREATE TRIGGER my_row_version_trigger
    BEFORE UPDATE ON my_table
    FOR EACH ROW
    EXECUTE PROCEDURE pglogical.inc_row_version('row_version');
```

This approach resembles Lamport timestamps and - in combination with `REPLICA IDENTITY FULL` and `check_full_tuple` (see below) - fully prevents the ABA problem for conflict detection.

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`.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

pglogical.conflict_log_level

Sets the log level for reporting detected conflicts.

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

Possible values are the same as for PostgreSQL `log_min_messages` parameter.

The default is `LOG`.

pglogical.conflict_ignore_redundant_updates

In case the subscriber retrieves an INSERT or UPDATE to a locally pre-existing and equivalent tuple, it is simply ignored without invoking any conflict handler or logging on the subscriber side, if this option is turned on.

To be used in combination with `REPLICA IDENTITY FULL`.

The default is `false`.

pglogical.conflict_check_full_tuple

This option controls the detection of UPDATE-UPDATE conflicts. By default, the origin of the existing tuple is compared to the expected origin - every mismatch is considered a conflict and initiates conflict handling. This is a low-overhead conflict detection mechanism and is therefore the default. However, it can lead to false positives and invoke conflict handlers inadvertently.

With this option turned on, the expected tuple, as it was before the update on the provider, is compared to the existing tuple on the subscriber. This allows for a better conflict detection mechanism and (in combination with a row version column) can mitigate all false positives.

Due to the requirement to know the full old tuple, this option only ever affects relations that are set to `REPLICA IDENTITY FULL`.

The default is `false`.

pglogical.batch_inserts

This tells pglogical writer to use the batch insert mechanism if possible. The 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 perform many inserts into one table. pglogical will switch to batch mode when the transaction performed than 5 INSERTs, or 5 rows within a COPY.

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.

The default is `true`.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

config.session_replication_role

This tells pglogical writer what `session_replication_role` to use. This can be useful mainly in case when it's desirable to enforce `FOREIGN KEY` constraints.

The default is `replica` which ignores foreign keys when writing changes to the database.

WARNING: Use with caution. This option changes trigger execution behavior as documented in PostgreSQL documentation. If set to `origin` or `local` this will fire **normal** triggers in the database which can leadi to the trigger being executed both on the upstream and on the downstream!

Restrictions

There are some additional restrictions imposed by pglogical writer over the standard set of restrictions.

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).

Deferrable unique indexes

Deferrable unique indexes are supported; however initially deferred unique indexes might result in apply retries, as the conflicts might not be detected on first try due to the deferred uniqueness check.

Note that deferred `PRIMARY KEY` cannot be used as `REPLICA IDENTITY` - PostgreSQL will throw an error if this is attempted. As a result a table withi a deferred `PRIMARY KEY` does not have `REPLICA IDENTITY` unless another `REPLICA IDENTITY` is explicitly set. Replicated tables without `REPLICA IDENTITY` cannot receive `UPDATEs` or `DELETEs`.

Foreign Keys

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

This behavior can be changed using `config.session_replication_role` writer option.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

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.

Triggers

Trigger behavior depends on the `config.session_replication_role` setting of the writer. By default it's set to `replica`, which means that `ENABLE REPLICA` and `ENABLE ALWAYS` triggers will be fired. When it's set to `origin` or `local`, it will trigger normal triggers.

Only row triggers are fired. Statement triggers are ignored as there are no statements executed by the writer. Per-column UPDATE triggers are ignored.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

SPI writer

The SPI writer is alternative writer for writing into local PostgreSQL instances when using a pglogical subscription.

This writer will use SQL statements to write the data locally. This means that there is no conflict detection or resolution support. Constraints on tables will be executed with the exception of foreign keys (see [#Foreign Keys] for details), and both row and statement triggers which are marked as `REPLICA` are triggered. It also fully supports `REPLICA IDENTITY`.

This writer is also used by default when a subscription is created on Postgres-XL.

Conflicts handling

Conflicts are not generally detected in SPI writer. The behavior during application of conflicting remote changes depends on which change is being replicated. For `INSERT`s, the replication will throw an error if there is a unique constraint violation. The `UPDATE` will simply be executed as a normal `UPDATE` without any information about the conflict occurring, and `DELETE` will skip the missing row.

Conflict Logging

To make diagnosis and handling of the conflicts easier, Pglogical will, by default, log every conflict into the PostgreSQL log file. This behavior can be changed with more granularity with the following functions.

pglogical.alter_subscription_add_log

Add a named conflict logging configuration for a node.

Synopsis

```
pglogical.alter_subscription_add_log(sub_name text,
                              log_name text,
                              log_to_file bool DEFAULT true,
                              log_to_table regclass DEFAULT NULL,
                              conflict_type text[] DEFAULT NULL,
                              conflict_resolution text[] DEFAULT NULL)
```

Parameters

- `sub_name` - the subscription for which is being changed
- `log_name` - name of the logging configuration

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- `log_to_file` - whether to log to the server log file
- `log_to_table` - whether to log to a table, and which table should be the target; NULL (the default) means do not log to a table
- `conflict_type` - which conflict types to log; NULL (the default) means all
- `conflict_resolution` - which conflict resolutions to log; NULL (the default) means all

pglogical.alter_node_remove_log

Remove an existing conflict logging configuration from a node.

Synopsis

```
pglogical.alter_node_remove_log(subscription text,
                                log_config_name text)
```

Parameters

- `subscription` - name of the subscription that is being changed
- `log_config_name` - name of the logging configuration to be removed

Logging to a Table

Conflicts will be logged to a table if `log_to_table` is set to a non-NULL value. The target table can be any user table which contains any of recognized columns. The pre-existing table pglogical.apply_log contains all the recognized columns, so this table can be used as parameter for `log_to_table` without needing any additional configuration

The user conflict log table can be any regular table which contains any of the following columns (the column matching is done using column name and type so these need to be exact):

- `sub_id` of type `oid` - which subscription has produced this conflict; can be joined to `pglogical.subscription` table
- `local_xid` of type `xid` - local transaction of the replication process at the time of conflict
- `local_lsn` of type `pg_lsn` - local lsn of the replication process at the time of conflict
- `local_time` of type `timestamptz` - local time of the conflict
- `remote_xid` of type `xid` - transaction which produced the conflicting change on the remote node (a peer)
- `remote_commit_lsn` of type `pg_lsn` - commit lsn of the transaction which produced the conflicting change on the remote node (a peer)
- `remote_commit_time` of type `timestamptz` - commit timestamp of the transaction which produced the conflicting change on the remote node (a peer)
- `conflict_type` of type `integer` - detected type of the conflict (see [Conflict Types] below)

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- `conflict_resolution` of type `integer` - conflict resolution chosen (see [Conflict Resolutions] below)
- `conflict_index` of type `regclass` - conflicting index (only valid if the index wasn't dropped since)
- `nspname` of type `text` - name of the schema for the relation on which the conflict has occurred
- `relname` of type `text` - name of the relation on which the conflict has occurred
- `key_tuple` of type `json` - json representation of the key used for matching the row
- `remote_tuple` of type `json` - json representation of an incoming conflicting row
- `local_tuple` of type `json` - json representation of the local conflicting row
- `apply_tuple` of type `json` - json representation of the resulting (the one that has been applied) row
- `local_tuple_xmin` of type `xid` - transaction which produced the local conflicting row (if `local_tuple` is set and the row is not frozen)
- `local_tuple_node_id` of type `oid` - node which produced the local conflicting row (if `local_tuple` is set and the row is not frozen)
- `local_tuple_commit_time` of type `timestamptz` - last known change timestamp of the local conflicting row (if `local_tuple` is set and the row is not frozen)

Any of the columns above may be omitted from the table in which case the information associated with that column won't be saved.

Please note that any of the values for these columns may be NULL with the exception of `sub_id`.

`pglogical.apply_log_summary`

This view is contains user-readable details of row conflict.

`pglogical.apply_log_summary` Columns

```
Name              |            Type             | Description
```

—————————-+—————————-+———— schema | text | Name of the schema table | text | Name of the table local_tuple_commit_time | timestamp with time zone | Time of local commit remote_commit_time | timestamp with time zone | Time of remote commit conflict_type | text | Type of conflict conflict_resolution | text | Resolution adopted

pglogical.conflict_resolution_to_string

Transforms the conflict resolution from oid to text.

The view pglogical.apply_log_summary uses it to give user-friendly information on the conflict resolution.

pglogical.conflict_type_to_string

Transforms the conflict type from oid to text.

The view pglogical.apply_log_summary uses it to give user-friendly information on the conflict type.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

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 to resolve the conflict

pglogical.batch_inserts

Tells pglogical writer to use the batch insert mechanism if possible. The batch mechanism uses PostgreSQL internal batch insert mode which is also used by `COPY`.

The batch inserts will improve replication performance of transactions that did many inserts into one table. pglogical will switch to batch mode when the transaction performed 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.

The default is `true`.

Restrictions

There are some additional restrictions imposed by SPI writer over the standard set of restrictions.md.

FOREIGN KEYS

Foreign key 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.

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.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Triggers

The 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.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

PostgreSQL settings which affect pglogical

Several PostgreSQL configuration options may need adjusting for pglogical to work.

PostgreSQL must be configured for logical replication:

```
wal_level = 'logical'
```

The pglogical library need to be loaded at server start, so the parameter `shared_preload_libraries` must contain pglogical, e.g.:

```
shared_preload_libraries = 'pglogical'
```

As pglogical uses additional worker processes to maintain state and apply the replicated changes, enough worker process slots need to be present:

```
max_worker_processes = 10
```

The formula for computing the correct value of `max_worker_processes` is: one for instance + one per database on the provider (upstream), one for instance + one per database + two per subscription on the subscriber (downstream).

The replication slots and origins are used so enough slots for those need to exist; both replication slots and origins are controlled by same configuration option:

```
max_replication_slots = 10
```

One per subscription on both provider and subscriber is needed.

The replication data is sent using walsender (just like physical replication):

```
max_wal_senders = 10
```

There is one walsender needed for every subscriber (on top of any standbys or backup streaming connections).

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

```
track_commit_timestamp = on
```

Also `pg_hba.conf` has to allow replication connections from the subscribers.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

pglogical specific settings

There are additional pglogical specific configuration options. Some generic options are mentioned below, but most of the configuration options depend on which writer is used and are documented as part of the individual writer documentation.

`pglogical.synchronous_commit`

This controls whether pglogical apply worker should use synchronous commit. By default this is off. Turning it on has performance implications - the maximum replication throughput will be much lower. However in low TPS environments which use `synchronous_commit = remote_apply` on the provider, turning this option on can improve the transaction latency. This guidance may change in later releases.

The pglogical.synchronous_commit setting for a subscription determines what happens to the things that the subscription's apply worker writes locally. The subscription's apply worker operates much like a normal client backend, and whatever it writes and commits is subject to its current pglogical.synchronous_commit setting.

In most cases, pglogical.synchronous_commit off is the best setting because it avoids the flushing work at commit time, and it is safe because in case of a crash the data can be re-obtained from the publishing server.

But if you use synchronous replication on the publishing server, then the publishing server will wait for the subscribing server to send feedback messages when the sent data has been flushed to disk on the subscribing server (depending on the particular setting). If the subscriber has pglogical.synchronous_commit off, then the flushing happens at some random later time, and then the upstream publisher has to wait for that to happen. In order to speed that up, you need to make the subscriber flush stuff faster, and the way to do that is to set pglogical.synchronous_commit to a value other than off on the subscriber.

Also if you have standbys connected to this subscriber server then you can set the value of pglogical.synchronous_commit to wait for confirmation from its standbys.

NOTE As per design, if on, this configuration will always wait for the local flush confirmation, even if the `synchronous_standby_names` would point to any physical standby/s.

The default is off.

`pglogical.track_subscription_apply`

This controls whether to track per subscription apply statistics. If this is on, the `pglogical.stat_subscription` view will contain performance statistics for each subscription which has received any data, otherwise the view is empty.

Collecting statistics requires additional CPU resources on the subscriber.

The default is on.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

`pglogical.track_relation_apply`

This controls whether to track per table apply statistics. If this is on, the `pglogical.stat_relation` view will contain performance statistics for each subscribed relation which has received any data, otherwise the view is empty.

Collecting statistics requires additional CPU resources on the subscriber.

The default is off.

`pglogical.temp_directory`

This defines system path for where to put temporary files needed for schema synchronization. This path needs to exist and be writeable by users running Postgres.

The default is empty, which tells pglogical to use the default temporary directory based on environment and operating system settings.

`pglogical.extra_connection_options`

This option 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`.

`pglogical.synchronize_failover_slot_names`

This standby option allows setting which logical slots should be synchronized to this physical standby. It's comma separated list of slot filters.

Slot filter is defined as `key:value` pair (separated by colon) where `key` can be one of:

- `name` - specifies to match exact slot name
- `name_like` - specifies to match slot name against SQL `LIKE` expression
- `plugin` - specifies to match slot plugin name agains the value

The `key` can be omitted and will default to `name` in that case.

For example `'my_slot_name,plugin:pglogical_output,plugin:pglogical'` will synchronize slot named "my_slot_name" and any pglogical slots.

If this is set to empty string, no slots will be synchronized to this physical standby.

Default value is `'plugin:pglogical,plugin:pglogical_output'` meaning pglogical slots will be synchronized.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

`pglogical.synchronize_failover_slots_drop`

This standby option controls what happens to extra slots on standby that are not found on primary using pglogical.synchronize_failover_slot_names filter. If it's set to true, they will be dropped, otherwise they will be kept.

The default value is `true`.

`pglogical.synchronize_failover_slots_dsn`

A standby option for specifying which connection string to use to connect to primary when fetching slot information.

If empty (and default) is to use same connection string as `primary_conninfo`.

Note that `primary_conninfo` cannot be used if there is a `password` field in the connection string because it gets obfuscated by PostgreSQL and pglogical can't actually see the password. In this case the `pglogical.synchronize_failover_slots_dsn` must be used.

`pglogical.standby_slot_names`

This option is typically used in failover configurations to ensure that the failover-candidate streaming physical replica(s) for this pglogical provider have received and flushed all changes before they ever become visible to any subscribers. That guarantees that a commit cannot vanish on failover to a standby for the provider.

Replication slots whose names are listed in the comma-separated `pglogical.standby_slot_names` list are treated specially by the walsender on a pglogical provider.

pglogical's logical replication walsenders will ensure that all local changes are sent and flushed to the replication slots in `pglogical.standby_slot_names` before the provider sends those changes to any other pglogical replication clients. Effectively it provides a synchronous replication barrier between the named list of slots and all pglogical replication clients.

Any replication slot may be listed in `pglogical.standby_slot_names`; both logical and physical slots work, but it's generally used for physical slots.

Without this safeguard, two anomalies are possible where a commit can be received by a subscriber then vanish from the provider on failover because the failover candidate hadn't received it yet:

- For 1+ subscribers, the subscriber may have applied the change but the new provider may execute new transactions that conflict with the received change, as it never happened as far as the provider is concerned;

and/or

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- For 2+ subscribers, at the time of failover, not all subscribers have applied the change.The subscribers now have inconsistent and irreconcilable states because the subscribers that didn't receive the commit have no way to get it now.

Setting `pglogical.standby_slot_names` will (by design) cause subscribers to lag behind the provider if the provider's failover-candidate replica(s) are not keeping up. Monitoring is thus essential.

If pglogical.standby_slot_names is not set and a physical standby is configured; failover to this standby will have data consistency issues as described above. However, the replica could just be a simple read replica. In any case, we warn on the replica about the potential data corruption/divergence that could result if failover is desired to such a standby.

Note that this setting is generally not required for BDR3 nodes (which are based on pglogical). Unlike base pglogical3, BDR3 is capable of reconciling lost changes from surviving peer nodes.

`pglogical.standby_slots_min_confirmed`

Controls how many of the `pglogical.standby_slot_names` have to confirm before we send data to pglogical subscribers.

`pglogical.writer_input_queue_size`

This option is used to specify the size of the shared memory queue used by the receiver to send data to the writer process. If the writer process is stalled or making slow progress, then the queue might get filled up, stalling the receiver process too. So it's important to provide enough shared memory for this queue. The default is 1MB and the maximum allowed size is 1GB. While any storage size specifier can be used to set the GUC, the default is kB.

`pglogical.writer_output_queue_size`

This option is used to specify the size of the shared memory queue used by the receiver to receive data from the writer process. Since the writer is not expected to send a large amount of data, a relatively smaller sized queue should be enough. The default is 1MB and the maximum allowed size is 1MB. While any storage size specifier can be used to set the GUC, the default is kB.

`pglogical.min_worker_backoff_delay`

Rate limit pglogical background worker launches by preventing a given worker from being relaunched more often than every `pglogical.min_worker_backoff_delay` milliseconds. Time-unit suffixes are supported.

The default is 0, meaning no rate limit. The delay is a time limit applied from launch-to-launch, so a value of `'500ms'` would limit all types of workers to at most 2 (re)launches per second.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

If the backoff delay setting is changed and the PostgreSQL configuration is reloaded then all current backoff waits will be reset. Additionally, the `pglogical.worker_task_reset_backoff_all()` function is provided to allow the administrator to force all backoff intervals to immediately expire.

A tracking table in shared memory is maintained to remember the last launch time of each type of worker. This tracking table is not persistent; it is cleared by PostgreSQL restarts, including soft-restarts during crash recovery after an unclean backend exit.

The view `pglogical.worker_tasks` may be used to inspect this state so the administrator can see any backoff rate-limiting currently in effect.

For rate limiting purposes, workers are classified by "task". This key consists of the worker role, database oid, subscription id, subscription writer id, extension library name and function name, extension-supplied worker name, and the remote relation id for sync writers. `NULL` is used where a given classifier does not apply, e.g. manager workers don't have a subscription ID and receivers don't have a writer id.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Postgres-XL

Postgres-XL can act as a subscriber in pglogical, but not a provider. The replication to Postgres-XL is optimized for larger data loads (bulk copy) and those will perform better than with regular Postgres.

The minimum supported version of Postgres-XL is 9.5r1.5.

The subscription on Postgres-XL will always use SPI Writer.

Note that when replicating DDL, the Postgres-XL syntax rules will apply, and trying to replicate commands which are not supported by Postgres-XL may stop replication.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Failover with pglogical3

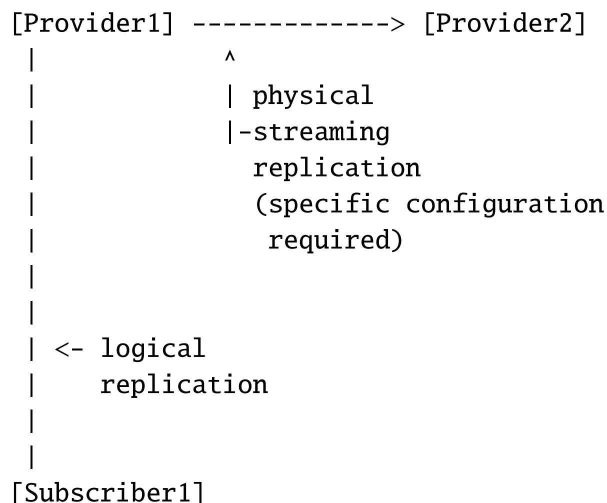
pglogical has support for following failover of both the provider (logical master) and subscriber (logical replica) if the conditions described in the following sections are met.

Only failover to streaming physical replicas is supported. pglogical subscribers cannot switch from replication from the provider to replicating from another peer subscriber.

Provider failover setup

With appropriate configuration of the provider and the provider's physical standby(s), pglogical subscriber(s) can follow failover of the provider to a promoted physical streaming replica of the provider.

Given a topology like this:



```
[Provider1] -------------> [Provider2]
 |            ^
 |            | physical
 |            |-streaming
 |              replication
 |              (specific configuration
 |               required)
 |
 |
 | <- logical
 |    replication
 |
 |
[Subscriber1]
```

On failure of Provider1 and promotion of Provider2 to replace it, pglogical on Subscriber1 can consistently follow the failover and promotion if:

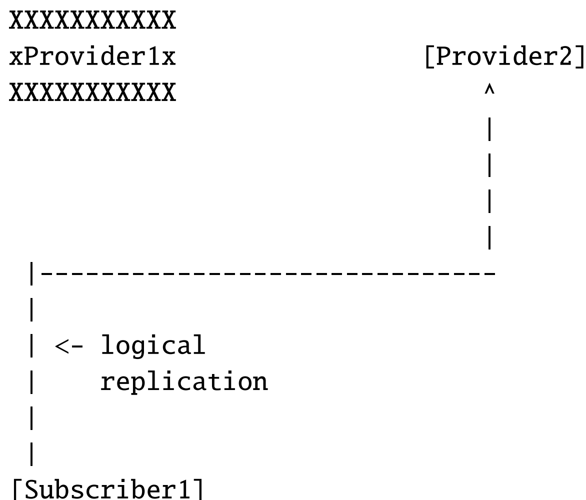
- Provider1 and Provider2 run PostgreSQL 10 or newer
- The connection between Provider1 and Provider2 uses streaming replication with hot standby feedback and a physical replication slot. It's OK if WAL archiving and a `restore_command` is configured as a fallback.
- Provider2 has:
- `recovery.conf`:
  - `primary_conninfo` pointing to Provider1
  - `primary_slot_name` naming a physical replication slot on Provider1 to be used only by Provider2

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- `postgresql.conf`:
  - `pglogical` in its `shared_preload_libraries`
  - `hot_standby = on`
  - `hot_standby_feedback = on`
  - `pglogical.synchronize_failover_slot_names` can be modified to specify which slots should be synchronized (default is all pglogical/bdr slots)
- Provider1 has:
- `postgresql.conf`:
  - `pglogical.standby_slot_names` lists the physical replication slot used for Provider2's `primary_slot_name`. Promotion will still work if this is not set, but subscribers may be inconsistent per the linked documentation on the setting.
- Provider2 has had time to sync and has created a copy of Subscriber1's logical replication slot. pglogical3 creates master slots on replicas automatically once the replica's resource reservations can satisfy the master slot's requirements, so just check that all pglogical slots on the master exist on the standby, and have `confirmed_flush_lsn` set.
- Provider2 takes over Provider1's IP address or hostname *or* Subscriber1's existing subscription is reconfigured to connect to Provider2 using `pglogical.alter_node_add_interface` and `pglogical.alter_subscription_interface`.

It is not necessary for Subscriber1 to be aware of or able to connect to Provider2 until it is promoted.

The post-failover topology is:



The reason pglogical must run on the provider's replica, and the provider's replica must use a physical replication slot, is due to limitations in PostgreSQL itself.

Normally when a PostgreSQL instance is replaced by a promoted physical replica of the same instance, any replication slots on that node are lost. Replication slot status is not itself replicated along physical

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

replication connections and does not appear in WAL. So if the failed-and-replaced node was the upstream provider of any logical subscribers, those subscribers stop being able to receive data and cannot recover. Physical failover breaks logical replication connections.

To work around this, pglogical3 running on the failover-candidate replica syncs the state of the master provider's logical replication slot(s) to the replica. It also sends information back to the master to ensure that those slots guarantees' (like `catalog_xmin`) are respected by the master. That synchronization requires a physical replication slot to avoid creating excessive master bloat and to ensure the reservation is respected by the master even if the replication connection is broken.

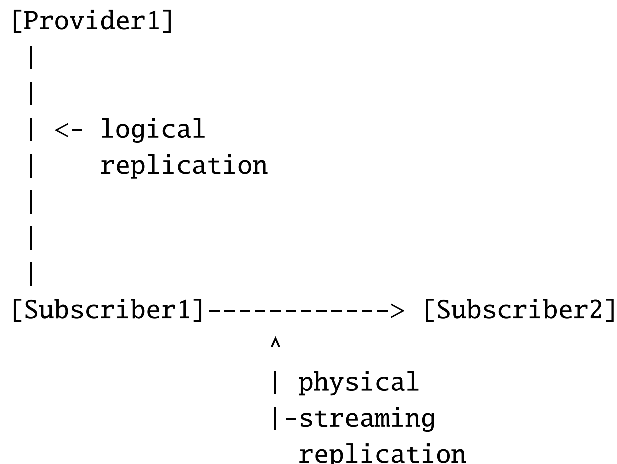
Subscriber failover setup

pglogical automatically follows failover of a subscriber to a streaming physical replica of the subscriber. No additional configuration is required.

WARNING: At present it's possible for the promoted subscriber to lose some transactions that were committed on the failed subscriber and confirmed-flushed to the provider, but not yet replicated to the new subscriber at the time of promotion. That's because the provider will silently start replication at the greater of the position the subscriber sends from its replication origin and the position the master has recorded in its slot's `confirmed_flush_lsn`.

Where possible you should execute a planned failover by stopping the subscription on Subscriber1 and waiting until Subscriber2 is caught up to Subscriber1 before failing over.

Given the server topology:



```
[Provider1]
  |
  |
  | <- logical
  |    replication
  |
  |
  |
[Subscriber1]------------> [Subscriber2]
                 ^
                 | physical
                 |-streaming
                   replication
```

Upon promotion of Subscriber2 to replace a failed Subscriber1, logical replication will resume normally. It doesn't matter whether Subscriber2 has the same IP address or not.

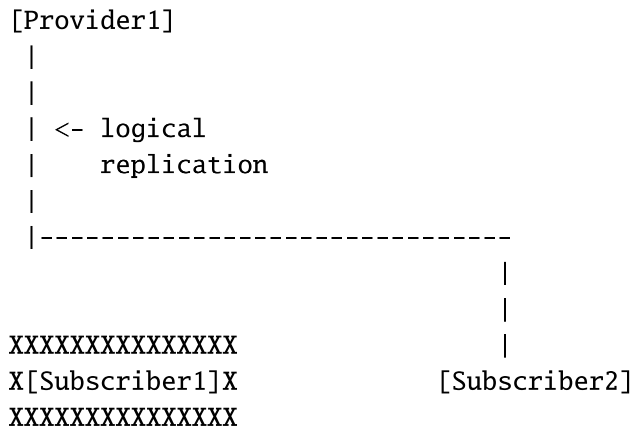
For replication to resume promptly it may be necessary to explicitly terminate the walsender for Subscriber1 on Provider1 if the connection failure is not detected promptly by Provider1. pglogical enables

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

TCP keepalives by default so in the absence of manual action it should exit and release the slot automatically in a few minutes.

It is important that Subscriber1 be fenced or otherwise conclusively terminated before Subscriber2 is promoted. Otherwise Subscriber1 can interfere with Subscriber2's replication progress tracking on Provider1 and create gaps in the replication stream.

After failover the topology is:



```
[Provider1]
 |
 |
 | <- logical
 |    replication
 |
 |-------------------------------
                               |
                               |
XXXXXXXXXXXXXXX                |
X[Subscriber1]X           [Subscriber2]
XXXXXXXXXXXXXXX
```

Note: at this time it is possible that there can be a small window of replicated data loss around the window of failover. pglogical on Subscriber1 may send confirmation of receipt of data to Provider1 before ensuring that Subscriber2 has received and flushed that data.

Additional functions

pglogical.sync_failover_slots()

Signal the supervisor to restart the mecanism to synchronize the failover slots specifyed in the `pglogical.synchronize_failover_slot_names`

Synopsis

pglogical.syncfailover_slots();

This function should be run on the subscriber.

Legacy: Provider failover with pglogical2 using failover slots

An earlier effort to support failover of logical replication used the "failover slots" patch to PostgreSQL 9.6. This patch is carried in 2ndQPostgres 9.6 (only), but did not get merged into any community PostgreSQL version. pglogical2 supports using 2ndQPostgres and failover slots to follow provider failover.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

The failover slots patch is neither required nor supported by pglogical3. pglogical3 only supports provider failover on PostgreSQL 10 or newer, since that is the first PostgreSQL version that contains support for sending `catalog_xmin` in hot standby feedback and for logical decoding to follow timeline switches.

This section is retained to explain the change in failover models and reduce any confusion that may arise when updating from pglogical2 to pglogical3.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Restrictions

pglogical currently has the following restrictions or missing functionality. These might be addressed in future releases.

Superuser is required

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

UNLOGGED and TEMPORARY not replicated

`UNLOGGED` and `TEMPORARY` tables will not and cannot be replicated, similar to physical streaming replication.

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.

PRIMARY KEY or REPLICA IDENTITY required

When replicating `UPDATE`s and `DELETE`s for tables that lack a `PRIMARY KEY`, the `REPLICA IDENTITY` must be set to `FULL`. However it's important to note that without `PRIMARY KEY` every `UPDATE` or `DELETE` will produce a sequential scan on a table which will have severe detrimental effect on performance of replication and subsequently the replication lag.

Note: On regular PostgreSQL nodes it's only possible to set the `REPLICA IDENTITY` to `FULL` via `ALTER TABLE`, however on pglogical nodes tables can be created with `REPLICA IDENTITY FULL` directly using the following syntax:

```
CREATE TABLE name (column_a int) WITH (replica_identity = full);
```

See http://www.postgresql.org/docs/current/static/sql-altertable.html#SQL-CREATETABLE-REPLICA-IDENTITY for details on replica identity.

DDL

There are several limitations of DDL replication in pglogical, for details check the DDL Replication chapter.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Sequences

The state of sequences added to replication sets is replicated periodically and not in real-time. A dynamic buffer is used for the value being replicated so that the subscribers actually receive the future state of the sequence. This minimizes the chance of the subscriber's notion of the 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 a given sequence after "big events" in the database such as data loading or during the online upgrade.

The types bigserial and bigint are recommended for sequences on multi-node systems as smaller sequences might reach the end of the sequence space fast.

Users who want to have independent sequences on the provider and subscriber can avoid adding sequences to replication sets and create sequences with a step interval equal to or greater than the number of nodes, and then set 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.

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 the same version of PostgreSQL or from lower version to a higher version.

Replicating between different minor versions makes no difference at all.

pglogical.pglogical_version

This function retrieves the textual representation of the PGL version that is currently in use.

```
SELECT pglogical.pglogical_version();
```

pglogical.pglogical_version_num

This function retrieves a numerical representation of the PGL version that is currently in use. Version numbers are monotonically increasing, allowing this value to be used for less-than and greater-than comparisons.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Database encoding differences

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

Large objects

PostgreSQL's logical decoding facility does not support decoding changes to large objects, so pglogical cannot replicate Large Objects. This does not restrict the use of large values in normal columns.

Additional restrictions

Please node that additional restrictions may apply depending on which writers.md is being used and which version of PostgreSQL is being used. These additional restrictions are documented in their respective sections (ie., every writer documents it's own additional restrictions).

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Troubleshooting

The main tool for troubleshooting is the PostgreSQL log file.

On the upstream side, monitoring uses the views:

```
pg_catalog.pg_replication_slots
pg_catalog.pg_stat_replication
```

On the subscriber side, the main point of reference is:

```
SELECT * FROM pglogical.subscriptions
```

along with the other information functions documented above and the usual tools such as:

```
pg_catalog.pg_stat_activity
pg_catalog.pg_locks
```

Although the logs should generally be your main reference, the following view is extremely useful in getting a quick overview of recent problems:

```
pglogical.worker_error
```

Statistics are reported by:

```
SELECT * FROM pglogical.stat_relation;
SELECT * FROM pglogical.stat_subscription;
```

Other views provide logs and details:

```
SELECT * FROM pglogical.local_sync_status;
SELECT * FROM pglogical.show_subscription_status();
SELECT * FROM pglogical.sub_history;
SELECT * FROM pglogical.sub_log;
SELECT * FROM pglogical.worker_error;
SELECT * FROM pglogical.apply_log;
SELECT * FROM pglogical.apply_log_summary;
SELECT * FROM pglogical.show_workers();
SELECT * FROM pglogical.worker_tasks;

SELECT * FROM pg_catalog.pg_stat_activity;
SELECT * FROM pg_catalog.pg_locks;
SELECT * FROM pg_catalog.pg_replication_origin_status;
```

The relation `pglogical.worker_error_summary` is particularly important for getting a quick overview of recent problems, though the logs should generally be your main reference.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Diagnostic views and relations

pglogical.worker_error

This relation shows the last error reported by each kind of pglogical worker. Only the most recent error is retained for each distinct worker task. Receiver workers are tracked separately to their writer(s), as are any writer(s) used for table (re)sync purposes.

walsender workers cannot record errors in `pglogical.worker_error`. Their errors are only available in the log files.

pglogical.worker_tasks

The `pglogical.worker_tasks` view shows pglogical's current worker launch rate limiting state as well as some basic statistics on background worker launch and registration activity.

Unlike the other views listed here, it is not specific to the current database and pglogical node; state for all pglogical nodes on the current PostgreSQL instance is shown. Join on the current database to filter it.

`pglogical.worker_tasks` does not track walsenders and output plugins.

See the configuration option `pglogical.min_worker_backoff_delay` for rate limit settings and overrides.

pglogical.apply_log and pglogical.apply_log_summary

The `pglogical.apply_log_summary` view summarizes the record of apply worker events kept in `pglogical.apply_log`. This records human-readable information about conflicts and errors that arose during apply.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Error handling in pglogical

When pglogical workers encounter an error condition during operation they report the error to the PostgreSQL log file, record the error to the `pglogical.worker_error` table if possible, and exit.

Unlike normal PostgreSQL user backends they do not attempt to recover from most errors and resume normal operation. Instead the worker in question will be relaunched soon and will resume operations at the last recoverable point. In the case of apply workers and walsenders that generally means restarting the last uncommitted transaction from the beginning.

This is an intentional design choice to make error handling and recovery simpler and more robust.

For example, if an apply worker tries to apply an `UPDATE` and the new row violates a secondary unique constraint on the target table, the apply worker will report the unique violation error and exit. The error information will be visible in the `pglogical.worker_error` table. The walsender worker on the peer end will exit automatically as well. The apply worker will be relaunched by the manager worker for the database in a few seconds and will retry the failed transaction from the beginning. If the conflicting row has since been removed the transaction will apply normally and replication will resume. If not, the worker will error again and the cycle will repeat until the cause of the error is fixed. In this case the fix would typically be for another subscription or a local application write to replicate a change that clears the unhandled conflict condition or for the administrator to intervene to change the conflicting row.

Diagnosing and fixing errors

It's important to first check that your schema and deployment don't violate any of the restrictions imposed by pglogical. Also check the additional writer-specific restrictions from the pglogical writer you are using, most likely the HeapWriter.

Common problems

Some issues that arise when operating pglogical include:

- Incorrect or changed provider address or hostname. Update the interface definition for the subscription.

Use `pglogical.alter_node_add_interface(...)` and `pglogical.alter_subscription_interface(...)` to change the subscriber's recorded address for the provider.

- Incorrect `pg_hba.conf` on provider disallowing subscriber from connecting. The subscriber must be able to connect in both replication and ordinary non-replication mode.

Correct the `pg_hba.conf` on the provider and `SELECT pg_reload_conf();` on the provider.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- Incompatible schema definitions on provider and subscriber caused by schema changes being made without DDL replication enabled and without use of `pglogical.replicate_ddl_command`. For example, missing columns on subscriber that haven't been excluded by a column filter, differing data types for columns between provider and subscriber, etc.

(Some data type differences are actually permitted, but care must be taken that the text representations are compatible. Do not use differing data types for PostgreSQL built-in data types. See restrictions.)

- Incorrectly defined `ENABLE REPLICA` or `ENABLE ALWAYS` triggers firing on apply on the subscriber and causing errors.
- Heap writers configured to fire normal triggers and foreign key validation triggers (using writer option `config.session_replication_role`). Problems arise when not all triggers have been checked to ensure they'll work correctly with row-replication and without statement triggers being fired as well. Or when FK violations or check constraint violations are created by replication set configuration such as row and column filters or by referenced tables not being replicated along with the referencing tables.
- Inconsistent versions of PostgreSQL or extensions between provider and subscriber where the version difference affects the behaviour or limits of a data type being replicated.

pglogical explicitly supports replicating between different versions of PostgreSQL, so a version difference alone is not a problem. But the data being replicated must be valid on the subscriber's PostgreSQL version.

For example, apply errors may occur when replicating data from PostGIS 3.0 to PostGIS 2.5 where not all the 3.0 data is understood by 2.5. Similarly, replicating from a PostgreSQL configured without integer datetimes to one with integer datetimes may result in errors if there are non-integer datetimes with values outside the somewhat narrower range permitted by integer datetimes support.

Multiple data source issues

Additional classes of error tend to arise with any sort of multiple-data-source configuration, i.e. multiple subscriptions to different providers for the same tables and/or local writes to tables that are also part of a subscription. Some of these affect BDR3 as well.

These include:

- Tables with multiple unique constraints may cause unique violation errors during apply if the table receives writes from multiple sources.
- Updating the PRIMARY KEY value for rows, or deleting a key then inserting the same key again soon afterwards. This may cause unique violation errors during apply if the table receives writes from more than one source, i.e. multiple providers and/or local writes.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- Any sort of multiple data source use where `pglogical.conflict_resolution` is set to `error`. Use a multi-master-compatible option like `apply_remote` or `last_update_wins` or use BDR3 for more powerful conflict handling, detection and resolution.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Credits and Licence

pglogical has been designed, developed and tested by the 2ndQuadrant team:

- Petr Jelinek
- Craig Ringer
- Simon Riggs
- Peter Eisentraut
- Tomas Vondra
- Pallavi Sontakke
- Nikhil Sontakke
- Pavan Deolasee
- Umair Shahid
- Markus Wanner

Copyright (c) 2015-2020 2ndQuadrant Ltd

pglogical3 is provided under the terms of the 2ndQuadrant Product Usage License.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Appendix A: Release Notes for pglogical3

pglogical 3.6.33

This is a maintenance release for PGLogical 3.6 which includes fixes for issues identified previously.

- Don't replicate TRUNCATE as global message (BDR-2821, RT87453)
  The TRUNCATE command now takes the replication set into account.

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.6.23 and higher
- 2.4.0 and 2.4.1
- 2.3.3 and 2.3.4 ## pglogical 3.6.32

This is a maintenance release for PGLogical 3.6. It is equivalent to 3.6.31, but still gets a release and a version bump to match the BDR version number.

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.6.23 and higher
- 2.4.0 and 2.4.1
- 2.3.3 and 2.3.4

pglogical 3.6.31

This is a maintenance release for PGLogical 3.6 which includes fixes for issues identified previously.

Resolved Issues

- Keep the `lock_timeout` as configured on non-CAMO-partner BDR nodes (BDR-1916)
  A CAMO partner uses a low `lock_timeout` when applying transactions from its origin node. This was inadvertently done for all BDR nodes rather than just the CAMO partner, which may have led to spurious `lock_timeout` errors on pglogical writer processes on normal BDR nodes.

- Prevent walsender processes spinning when facing lagging standby slots (RT80295, RT78290)
  Correct signaling to reset a latch so that a walsender process does consume 100% of a CPU in case one of the standby slots is lagging behind.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.6.23 and higher
- 2.4.0 and 2.4.1
- 2.3.3 and 2.3.4

pglogical 3.6.30

This is a maintenance release for PGLogical 3.6 which includes fixes for issues identified previously.

Resolved Issues

- Push snapshot in SPI writer for every transaction (RT76368)
  This is required in newer versions of Postgres.

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.6.23 and higher
- 2.4.0 and 2.4.1
- 2.3.3 and 2.3.4

pglogical 3.6.29

This is a maintenance release for PGLogical 3.6 which includes fixes for issues identified previously.

Resolved Issues

- Stop replication in case of a CAMO misconfiguration (RT74906, BDR-1724)
  In case a normal BDR node was treated as a CAMO partner, but it itself is not configured as such via `bdr.camo_partner_of`, the node applied all transactional changes, but did not ever commit them. Correct this to throw a FATAL error and halt replication instead of silently ignoring transactions. This will allow for the node to resume replication once configured properly.

pglogical 3.6.28

This is a maintenance release for pglogical 3.6 which includes fixes for issues identified previously.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Resolved Issues

- Don't wait on own replication slot when waiting for standby_slot_names (RT74036)
  The walsenders that use slots named in standby_slot_names should not wait for anything, otherwise we might wait forever.
- Close partitions when get replication info about tables in repsets (RT74658)
  The partitions are skipped as the replication is handeld from the parent table, these partitions were not closed, thus issuing the reference leak warning.
- Enabling async conflict resolution for explicit 2PC (BDR-1609, RT71298)
  Continue applying the transaction using the async conflict resolution for explicit two phase commit.

Improvements

- Allow upgrades from pglogical 2.4.1

pglogical 3.6.27

This is a maintenance release for pglogical 3.6 which includes fixes for issues identified previously.

Resolved Issues

- Don't materialize remote tuple slot for conflict reporting (BDR-734, RT71005)
  Otherwise we might fill in defaults for any missing columns instead of keeping the existing value.
- Don't drop temporary synchronization replication slots which may still be needed by the synchronization connection (BDR-647, RT70760, RT68455, RT68352)
  Instead of doing cleanup periodically in the background, make the walsender that creates the slot responsible for the cleanup. Similar to how native temporary slots work in newer versions of PostgreSQL.
- Fix memory leak in the pglogical COPY handler (BDR-1219, RT72091)
  This fixes memory leak when synchronizing large tables.
- Fix snapshot handling around our internal executor processing (BDR-904)
  Recent changes in PostgreSQL uncovered minor issues in snapshot handling in row filtering and slot manipulation. This is mostly to improve compatibility with latest minor version of PostgreSQL.

Improvements

- Allow upgrades from pglogical 2.4.0
- Allow binary and internal protocol on more hardware combinations. This currently only affects internal testing.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

pglogical 3.6.26

This is a security and maintenance release for pglogical 3.6 which includes fixes for issues identified previously.

Resolved Issues

- Fix pg_dump/pg_restore execution (CVE-2021-3515) Correctly escape the connection string for both pg_dump and pg_restore so that exotic database and user names are handled correctly. Reported by Pedro Gallegos
- Fix potential divergence after physical failover (BDR-365, RT68894, RM19886)
  The `pglogical.standby_slot_names` setting now also protects subscriber standbys using those slots from being behind. Previously the slot on provider could move ahead of subscriber's standby causing data loss on failover of subscriber to that standby. Configuring `pglogical.standby_slot_names` on the subscriber for stanbys that are promotion targets prevents this issue now.
- Fix writer crash caused by concurrent relcache invalidation (RT70549) This crash is caused by a race with concurrent relcache invalidation deliveries. This was triggered by concurrent execution of commands that invalidate whole relcache (for example `VACUUM FULL`).
- Don't re-enter worker error handling loop recursive
  This should help make what happens clearer in any cases where we do encounter errors during error processing.

Other Changes

- Document `pglogical.alter_subscription_set_conflict_resolver`
  This functions allows configuration of conflict resolution used for the subscription.

Upgrades

This release supports upgrading from following versions of pglogical:

- 2.3.3
- 2.3.4
- 3.6.23 and higher

pglogical 3.6.25

This is a security and maintenance release for pglogical 3.6 which includes fixes for issues identified previously.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Resolved Issues

- Correctly set verbosity of `pg_ctl` command when executed from `pglogical_create_subscriber` The `pg_ctl` was always executed in silent mode even when `-v` option was given to `pglogical_create_subscriber` in previous versions of pglogical.

This allows for getting more meaningful troubleshooting information when analyzing issues with `pglogical_create_subscriber`.

Other Changes

- Optimize utility command processing (RT69617)
  For commands that won't affect any DB objects and don't affect pglogical we can skip the processing early without reading any pgl or system catalogs or calling to DDL replication plugin interfaces.

This is optimization for systems with large number of such utility command calls (for example using pglogical in transaction pooling).

pglogical 3.6.24

This is a security and maintenance release for pglogical 3.6 which includes fixes for issues identified previously.

Resolved Issues

- Correct cleanup of dead synchronization slots (RT69227)
  Instead of statistics, use the internal process list to reliably find backends for a given xmin.

pglogical 3.6.23

This is a security and maintenance release for pglogical 3.6 which includes fixes for issues identified previously.

Resolved Issues

- Don't replicate DDL on temporary objects (RM19491, RT69170) Don't replicate CREATE or DROP statements on objects(table, function, procedure, type and sequence) if they are on the "pg_temp" schema.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Other Changes

- Make smart shutdown of writer timeout after half of wal_receiver_timeout (RM19310) Otherwise it might get stuck forever when waiting in Postgres internal code.
- Add initial support for upgrades from 2.3.3
- Drop support for upgrading from long deprecated version 3.2, 3.3, 3.4 and 3.5

pglogical 3.6.22

This is a security and maintenance release for pglogical 3.6 which includes minor features as well as fixes for issues identified previously.

Resolved Issues

- Ensure that `pglogical.standby_slot_names` takes effect when `pglogical.standby_slots_min_confirmed` is at the default value of -1.

On 3.6.21 and older `pglogical.standby_slot_names` was ignored if `pglogical.standby_slot_names` is set to zero (RM19042).

Clusters satisfying the following conditions may experience inter-node data consistency issues after a provider failover:

- Running pglogical 3.0.0 through to 3.6.21 inclusive;
- Using pglogical subscriptions/or providers directly (BDR3-managed subscriptions between pairs of BDR3 nodes are unaffected);
- Have a physical standby (streaming replica) of a pglogical provider intended as a failover candidate;
- Have `pglogical.standby_slot_names` on the provider configured to list that physical standby;
- Have left `pglogical.standby_slots_min_confirmed` unconfigured or set it explicitly to zero;

This issue can cause inconsistencies between pglogical provider and subscriber and/or between multiple subscribers when a provider is replaced using physical replication based failover. It's possible for the subscriber(s) to receive transactions committed to the pre-promotion original provider that will not exist on the post-promotion replacement provider. This causes provider/subscriber divergence. If multiple subscribers are connected to the provider, each subscriber could also receive a different subset of transactions from the pre-promotion provider, leading to inter-subscriber divergence as well.

The `pglogical.standby_slots_min_confirmed` now defaults to the newly permitted value `-1`, meaning "all slots listed in `pglogical.standby_slot_names`". The default of 0 on previous releases was intended to have that effect, but instead effectively disabled physical-before-logical replication.

To work around the issue on older versions the operator is advised to set `pglogical.standby_slots_min_confirmed = 100` in `postgresql.conf`. This has no effect unless `pglogical.standby_slot_names` is also set.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

No action is generally required for this issue on BDR3 clusters. BDR3 has its own separate protections to ensure consistency during promotion of replicas.

- Fix very rare replication set cache invalidation race condition which could cause crash of a backend or walsender (RM19043, RM19244)
- Fix very rare writer relation cache invalidation race condition which could cause crash of the writer (RM19037)

Improvements

- Add `pglogical.replication_origin_status` view which allows `pglogical_superuser` role to see the status of replication origins.
  This is normally visible only to superuser in PostgreSQL itself.
- Improve `wal_receiver_timeout` handling introduced in 3.6.21
  Don't timeout on nodes that are doing table resynchronization but are oterwise idle.

pglogical 3.6.21

This is a security and maintenance release for pglogical 3.6 which includes minor features as well as fixes for issues identified previously.

Resolved Issues

- Fix segmentation fault encountered when adding a partitioned table with many partitions to replication set (RM15733, RT68352)
  The segmentation fault was caused by a cache entry of one of the partitions invalidated during copying data to the subscriber. This has been fixed by using a valid cache entry for this purpose.
- Fix a failure encountered when adding a large table to replication set (RM18154, RT68455)
  pglogical.replication_set_add_table() may fail to add a large table containing millions of rows to a replication set. Similar failure may be seen if the replication takes longer say due to a slow network between publisher and subscriber. Server logs of the subscriber node will indicate START_REPLICATION SLOT command failing with ERROR 42704 "replication slot does not exist". This failure has been fixed by correcting the logic which periodically removes unused replication slots created for synchronizing table being added to the replication set.
- Fix crash when running replica triggers on partitions (RM18252)
- Prohibit INSERT ON CONFLICT and MERGE (2ndQPostgres) commands on tables without replica identity (RM17323, RT68146)
  These commands might end up doing UPDATE or DELETE which would break replication when table does not have any replica identity.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- Fix memory leak in executor state cache during the initial data COPY (RM17668)
  This was particularly problematic when adding large tables to replication set.
- Fix memory leak in writer INSERT processing (RM17668)
  Resulted in unusually large memory use when applying of INSERTs that affected many rows.
- Fix race condition in invalidation handling of local_sync_status (RM17929)
  This could result in receiver waiting forever for the table resynchronization triggered by `pglogical.alter_subscription_resynchronize_table()` to finish.
- Fix rare race condition where reported flush lsn would be ahead of apply lsn (RM18044)
  This would mostly cause monitoring queries on provider to show odd values.

Improvements

- Document limitations of using `primary_conninfo` for slot synchronization to a standby (RM14612, RT67443)
- Make PGL receiver respect `wal_receiver_timeout` (RM13805, RT67066)
  After an unclean disconnect, the receiver process now terminates once the `wal_receiver_timeout` is exceeded. This allows it to be restarted and then attempt to reconnect. Prior to this release, the TCP expiration time of the OS applied.

pglogical 3.6.20

This is a security and maintenance release for pglogical 3.6 which includes minor features as well as fixes for issues identified previously.

Resolved Issues

- Only process keepalives in writer if they come outside of transaction (RT67858)
  Keepalives sent in middle of forwarded transactions could move wrong origin forward resulting in skipping future transactions from that origin.
- Use timestamp of slot snapshot for initial copy transaction (RM16396)
  We can't set commit timestamp of individual rows correctly when doing logical copy during subscription initialization because that would be too slow (every row would have to have separate transaction). But we can use the knowledge that each row had to be committed at or before the snapshot which we use to read the data was taken. So we find the last commit in that snapshot and use the timestamp of that commit.
  This helps with time base conflict resolution against the existing data copied during the subscription initialization.
- Keep open the connection until `pglogical_create_subscriber` finishes (RM13649)
  Set `idle_in_transaction_session_timeout` to 0 so we avoid any user setting that could close the connection and invalidate the snapshot.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Improvements

- CentOS 8 is now supported, starting with this release.

pglogical 3.6.19

This is a security and maintenance release for pglogical 3.6 which includes minor features as well as fixes for issues identified previously.

Resolved Issues

- SECURITY: Set search_path to empty for internal PGLogical SQL statements (RM15373)
  Also, fully qualify all operators used internally. PGLogical is now protected from attack risks identified in CVE-2018-1058, when the user application avoids the insecure coding practices identified there.
- Correct parsing of direct WAL messages (RT67762)
  Custom WAL messages emitted by PGLogical (or plugins building on top of it) can be broadcast or direct types. Decoding of the latter was incorrect and could in rare cases (depending on the node name) lead to “insufficient data left in message” or memory allocation errors. Decoding of such direct WAL messages has been corrected.
- Add pglogical.sync_failover_slots() function (RM14318)
  Signal the supervisor process to restart the mechanism to synchronize the failover slots specifyed in the “pglogical.synchronize_failover_slot_name”.
- Fix the `--extra-basebackup-args` argument passed to pg_basebackup (RM14808)
  Corrects how the `pglogical_create_subscriber` tool passes on such extra arguments to pg_backbackup.

Improvements

- Add more diagnostic information to pglogical.queue message (RM15292)
  A new key `info` has been added to `pglogical.queue` providing additional information about a queued DDL operation.

pglogical 3.6.18

This is a maintenance release for pglogical 3.6 which includes minor features as well as fixes for issues identified previously.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Improvements

- Warn about failover issues if standby_slot_names is not set (RT66767, RM12973) If pglogical.standby_slot_names is not set and a physical standby is configured; failover to this standby will have data consistency issues as per our documentation. However, the replica could just be a simple read replica. In any case, we now warn on the replica about the potential data corruption/divergence that could result if failover is desired to such a standby.
- Check repsets in create_subscription for pgl2 upstreams also.
- Various improvements to systemtap integration.

Resolved Issues

- Prevent a hang in case of an early error in the PGL writer (RT67433, RM14678)
- Allow postgres to start with pglogical library loaded but activity suspended
  Add start_workers commandline-only GUC to facilitate this.

pglogical 3.6.17

This is a maintenance release for pglogical 3.6 which includes minor features as well as fixes for issues identified previously.

Improvements

- Make the slot synchronization to standby more configurable (RM13111)
  Added several new configuration parameters which tune the behavior of the synchronization of logical replication slots from a primary to a standby PostgreSQL servers. This allows for better filtering, inclusion of non-pglogical replication sets and also using different connection string than physical replication uses (useful when different user or database should be used to collect information about slots).

Resolved Issues

- Fix issue with UPDATEs on partitions with different physical row representation than partition root (RM13539, RT67045)
  The partitions must have same logical row as partition root they can have different physical representation (primarily due to dropped columns). UPDATEs on such partitions need to do special handling to remap everything correctly otherwise constraints and not-updated TOAST columns will refer to wrong incoming data.
- Fix truncation of `_tmp` slot names in sync slots
  Long slot names could previously cause the temporary slot to be suffixed by `_tm` rather than the expected `_tmp` suffix.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Support, Diagnostic and Logging Changes

These changes don't directly change existing behaviour or add new user-facing features. They are primarily of interest to 2ndQuadrant support operations and for advanced diagnostic analysis.

- Expand non-invasive tracing (SystemTap, linux-perf, DTrace, etc) support to cover inspection of the pglogical receiver's input protocol stream, walsender output plugin protocol stream, and other useful events. (RM13517)
- Add a test and debug utility that decodes captured pglogical protocol streams into human-readable form (RM13538)
- Improve error context logging in the pglogical writer to show more information about the transaction being applied and its origin.
- Fix incorrectly reported commit lsn in errcontext messages from the pglogical heap writer (RM13796). This fix only affects logging output. The writer would report the lsn of the original forwarded transaction not the lsn of the immediate source transaction.
- Add subscription, local node and peer node names to heap writer errcontext log output.

pglogical 3.6.16

This is the sixteenth minor release of the Pglogical 3.6 series. This release includes mostly just enables BDR 3.6.16 without any significant changes to pglogical.

pglogical 3.6.15

This is the fifteenth minor release of the Pglogical 3.6 series. This release includes fixes for issues identified previously.

Resolved Issues

- Fix backwards-compatibility to PGLogical 2 (RM13333, RT66919)
  Recent releases performed additional checks during `create_subscription`, which are fine against other PGLogical 3 installations, but not backwards-compatible. This release corrects the check to account for backwards-compatibility.
- Correct a warning about GUC nest level not being reset (EE) (RM13375)
  The addition of the `lock_timeout` in 3.6.14 led to a warning being issued for CAMO and Eager All Node transaction ("GUC nest level = 1 at transaction start"). With this release, GUC nest levels are properly managed and the warning no longer occurs.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Improvements

- Add a new `pglogical.worker_tasks` view that tracks and records pglogical's background worker use. The view exposes information about the number of times a given type of worker has been restarted, how long it has been running, whether it accomplished any useful work, and more. This offers administrators more insight into pglogical's internal activity when diagnosing problems, especially when joined against the `pglogical.worker_error` table.
- Add support for rate-limiting pglogical background worker (re)launches. The new `pglogical.min_worker_backof` configuration option sets a minimum delay between launches of all types of pglogical background workers so that rapid respawning of workers cannot fill the log files and or excessive load on the system that affects other operations.

For example, if configured with `pglogical.min_worker_backoff_delay = '500ms'`, pglogical will not retry any given background worker startup more often than twice per second (`1000/500 = 2`).

A simple fixed-rate factor was deemed to be the most predictable and production-safe initial approach. Future enhancements may add a heuristic delay factor based on worker type, time from start to exit, number of recent launches, etc.

The launch backoff delay defaults to 0 (off) to prevent surprises for upgrading users.

A setting of `pglogical.min_worker_backoff_delay = '5s'` or similar is a reasonable starting point, and may become the default in a future release.

Upgrades

The PostgreSQL Global Development Group has phased out support for PostgreSQL 9.4 on all Debian based distributions. Following that, this release covers only PostgreSQL 9.5 and newer. We advise to upgrade to a newer version.

For RedHat based distributions, this release is still available for PostgreSQL 9.4.

pglogical 3.6.14

This is the fourteenth minor release of the Pglogical 3.6 series. This release includes fixes for issues identified previously.

Resolved Issues

- Resolve deadlocked CAMO or Eager transactions (RM12903, RM12910)
  Add a `lock_timeout` as well as an abort feedback to the origin node to resolve distributed deadlocking due to conflicting primary key updates. This also prevents frequent restarts and retries of the PGL writer process for Eager All Node and sync CAMO transactions.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

pglogical 3.6.12

This is the twelveth minor release of the Pglogical 3.6 series. This release includes fixes for issues identified previously.

Improvements

- Add infrastructure for `check_full_row` in DELETE operations used by BDR 3.6.12 (RT66493)
- Validate requested replication sets at subscribe time (RM12020, RT66310)
  `pglogical.create_subscription()` now checks that all requested replication sets actually exist on the provider node before returning. If any are missing it will raise an ERROR like:
  `ERROR: replication set(s) "nonexistent_repset" requested by subscription are missing on provider`

with a DETAIL message listing the full sets requested, etc.

On prior releases subscriptions with missing repsets would fail after `pglogical.create_subscription(...)` returned, during initial sync. The failure would only be visible in the logs where it is much less obvious to the user. Or if schema sync was not enable they could appear to succeed but not populate the initial table contents.

Resolved Issues

- Fix a potential deadlock at CAMO partner startup. (RM12187)
  After a restart, the CAMO partner resends all confirmations for recent CAMO protected transactions. In case these fill the internal queue between the receiver and writer processes, a deadlock was possible. This release ensures the receiver consumes pending feedback messages allowing the writer to make progress.

pglogical 3.6.11

This is the eleventh minor release of the Pglogical 3.6 series. This release includes fixes for issues identified previously.

Improvements

- Implement remote_commit_flush for CAMO. (RM11564)
  Additional level of robustness for CAMO, only replying when xact is known committed and flushed on partner node.
- Make receiver-writer shared queues of configurable size. (RM11779)
  Two new GUCs are introduced: pglogical.writer_input_queue_size (default 1MB) pglogical.writer_output_queue_size (default 1MB)

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- Add a warning when user tries to set update_origin_change to skip
- Add callback to request replay progress update. (RM6747)

Resolved Issues

- Send TimeZone GUC when replicating DDL (RT66019)
  To ensure that timezone dependent expressions in DDL get evaluated to same value on all nodes.
- Only use isvalid indexes when searching for conflicts (RT66036)
  Indexes currently being created or failed index creations will be ignored, to prevent concurrency issues with change apply and `CREATE INDEX CONCURRENTLY`.
- Fix crash when replication invalidations arrive outside a transaction (RM11159)
- Make the receiver apply the queue before shutting down (RM11778)
  Upon smart shutdown, the PGL writer no longer terminates immediately, requiring queued transactions to be resent, but applies already received transactions prior to shutting down.

pglogical 3.6.10

This is the tenth minor release of the Pglogical 3.6 series. This release includes fixes for issues identified previously.

Improvements

- Add support for a CAMO remote_write mode (RM6749)

Resolved Issues

- COMMIT after initial sync of a table. This avoids treating the first catchup xact as if it was part of the initial COPY, which could lead to strange errors or false conflicts. (RM11284).
- Remove the 4 billion row limit during the initial subscription synchronization (RT66050).
- Cleanup table replication cache when replication set configuration changes.
  Previously we could use stale cache on multiple calls for table replication info on same connection if user changed the configuration in meantime. This could result in initial sync missing replicated table if the configuration was changed while the subscription was being created.
- Remember repsets when caching table replication info.
  If the client calls the table replication info with different parameters, we need to remember them otherwise we might return cached value for wrong replication sets. This could result in initial sync copying data from table which were not supposed to be replicated.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

pglogical 3.6.9

This is the ninth minor release of the Pglogical 3.6 series. This release includes minor improvements.

Improvements

- Add support for local, remote_apply and remote_write. (RM11069, RT65801)
  We now accept the use of all the values that PostgreSQL accepts when configuring the "pglogical.synchronous_commit".
- Immediately forward all messages from the PGL receiver back to origin (BDR CAMO)
  Confirmations for CAMO protected transactions flow from the PGL writer applying the transaction back to origin node via the PGL receiver. This process used to consume only one confirmation message per iteration. It now consumes all pending confirmations from the PGL writer and immediately sends them back to the origin. It also decreases latency for BDR CAMO transactions in case confirmations queue up.

pglogical 3.6.8

This is the eigth minor release of the Pglogical 3.6 series. This release includes fixes for issues identified previously.

Resolved Issues

- Use RelationGetIndexAttrBitmap to get pkey columns. (RT65676, RT65797)
  No need to try to fetch pkey columns from index itself, we have relcache interface that does exactly what we need and does so in more performant way.

pglogical 3.6.7.1

This is a hot-fix release on top of 3.6.7.

Resolved Issues

- Fix a protocol violation after removal of an origin. (RT65671, RM10605)
  Removal of a replication subscription may lead to a walsender trying to forward data for unknown origins. Prevent emission of an invalid message in that case.

pglogical 3.6.7

pglogical 3.6.7 is the seventh minor release of the pglogical 3.6 series. This release includes minor new features as well as fixes for issues identified earlier.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Improvements

- Replicate TRUNCATE on a partition if only parent table is published in replication set (RT65335)
  Previously, we'd skip such TRUNCATE unless the partition was also published.
- Generate `target_table_missing` for TRUNCATE which is executed against non-existing table (RT10291)
  Allows for user-configured decision if it should be a replication-stopping issue or not.
- Improve performance of repeated UPDATEs and DELETEs executed on origin node by caching the replication configuration of tables in a user session.
- Reduce CPU usage of receiver worker when writer queue is full (RM10370).

Resolved Issues

- Fix partition replication set membership detection for multi-level partitioned tables
  Replicate changes correctly for multi-level partitioned tables, where only the intermediate partition is part of replication set (not root or leaf partitions).
- Support replication `TRUNCATE CASCADE` on tables referenced by `FOREIGN KEY` (RT65518)
  Previously this would throw error on the subscriber. This will only work if all tables on subscriber which have `FOREIGN KEY` on the table being `TRUNCATEd` are replicated tables. Also it's only supported on PostgreSQL 11 and higher.
- Flush writer between data copy and constraint restore
  Otherwise there could in some rare cases still be unapplied changes when creating constraints during initial synchronization of a subscription, potentially causing deadlocks.
- Fix potential writer queue corruption on very wide (1000+ columns) tables

Upgrades

This release supports upgrading from following versions of pglogical:

- 2.2.0
- 2.2.1
- 2.2.2
- 3.2.0 and higher

pglogical 3.6.6

pglogical 3.6.6 is the sixth minor release of the pglogical 3.6 series. This release includes minor new features as well as fixes for issues identified earlier.

Improvements

- New conflict type `update_pkey_exists` (RM9976)
  Allows resolving conflicts when a `PRIMARY KEY` was updated to one which already exists on the node which is applying the change.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- Add `pglogical.apply_log_summary` (RM6596)
  View over `pglogical.apply_log` which shows the human-readable conflict type and resolver string instead of internal id.

- Improve logging during both the initial data synchronization of a subscription and the individual table resynchronization.

Resolved Issues

- Make sure writer flushes changes after initial data copy (RT65185)
  Otherwise depending on timing and I/O load the subscription might not update positioning info and get data both via initial copy and replication stream catchup that follows.

Upgrades

This release supports upgrading from following versions of pglogical:

- 2.2.0
- 2.2.1
- 2.2.2
- 3.2.0 and higher

pglogical 3.6.5

pglogical 3.6.5 is the fifth minor release of the pglogical 3.6 series. This release includes minor new features as well as fixes for issues identified in 3.6.4.

Improvements

- Improve tuple lock waits during apply for deletes (RM9569)
  This should improve performance of replication of deletes and updates in contentious situation.

Resolved Issues

- Use consistent table list in initial data copy (RM9651/RT64809) To prevent issues during initial data copy and concurrent table drop.
- Cleanup worker_dsm_handle on worker detach (internal)
  Otherwise we could leave dangling DSM segment handle for a worker after a crash, which could confuse plugins using this API.
- Fix handling of empty eager transactions (RM9550)
  In case no relevant change remains to be applied on a replica node, the prepare of such an empty transaction now works just fine.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- Fix the replication sets output in `pglogical.pglogical_node_info()`
  Previously it could be garbled.
- Reduce log level for messages when resolving ERRCODE_T_R_SERIALIZATION_FAILUREs (RM9439)

Upgrades

This release supports upgrading from following versions of pglogical:

- 2.2.0
- 2.2.1
- 2.2.2
- 3.2.0 and higher

Note that upgrades from 2.2.x are only supported on systems with `pglogical.conflict_resolution` set to `last_update_wins`.

pglogical 3.6.4

pglogical 3.6.4 is the fourth minor release of the pglogical 3.6 series. This release includes minor new features as well as fixes for issues identified in 3.6.3.

New Features

- Apply statistics tracking (RM9063)
  We now track statistics about replication and resource use for individual subscriptions and relations and make them available in `pglogical.stat_subscription` and `pglogical.stat_relation` views. The tracking can be configured via `pglogical.stat_track_subscription` and `pglogical.stat_track_relation` configuration parameters.
- The `replicate_inserts` option now affects initial COPY
  We now do initial copy of data only if the table replicates inserts.

Resolved Issues

- Fix initial data copy of multi-level partitioned tables (RT64809)
  The initial data copy used to support only single level partitioning, multiple levels of partitioning are now supported.
- Don't try to copy initial data twice for partitions in some situations (RT64809)
  The initial data copy used to try to copy data from all tables that are in replication sets without proper regard to partitioning. This could result in partition data to be copied twice if both root partition and individual partitions were published via replication set. This is now solved, we only do the initial copy on the root partition if it's published.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- Fix handling of indexes when replicating INSERT to a partition (RT64809)
  Close the indexes correctly in all situations.
- Improve partitioning test coverage (RM9311)
  In light of the partitioning related issues, increase the amount of automated testing done against partitioned tables.
- Fix a leak in usage of the relation cache (RT64935)
- Fix a potential queue deadlock between writer and receiver (RT64935, RT64714)

pglogical 3.6.3

pglogical 3.6.3 is the third minor release of the pglogical 3.6 series. This release includes minor new features as well as fixes for issues identified in 3.6.2.

New Features

- Support `DoNotReplicateId` special origin
  This allows correct handling of "do not replicate" origin which allows skipping replication of some changes. Primarily needed internally for other features.
- Persist the last_xact_replay_timestamp (RT63881)
  So that it's visible even if the subscription connection is down.
- Rework documentation build procedure for better consistency between HTML and PDF documentation
  This mainly changes the way docs are structured into chapters so that there is single source of chapter list and ordering for both PDF and HTML docs.

Resolved Issues

- Invalidate local cache when adding new invalidation
  Fixes visibility of changes in the catalog cache view of the transaction which did those changes. Not triggered yet by any code but will be in the future releases.
- Open indexes after partition routing
  Otherwise we might be opening indexes of the root table rather than the partition, causing issues with handling conflicts for `INSERT` operation replication.

pglogical 3.6.2

pglogical 3.6.2 is the second minor release of the pglogical 3.6 series. This release includes minor new features as well as fixes for issues identified in 3.6.1.

New Features

- Support DEFERRED UNIQUE indexes
  They used to work only in limited cases before this release.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

- Support covering UNIQUE indexes (RT64650)
  The use of covering UNIQUE indexes could result in ambiguous error messages in some cases before.
- Add `--log-file` option to `pglogical_create_subscriber` (RT64129) So that log can be saved somewhere other than the current working directory

Resolved Issues

- Fix error message when the database name in the connection string in `pglogical_create_subscriber` is missing (RT64129) The previous message was ambiguous.
- Raise error when unknown parameter was specified for `pglogical_create_subscriber` (RT64129)
  Otherwise mistakes in command line arguments could be silently ignored.
- Solve timing issue with workers exiting while another one tries to start using same worker slot
  Before, we could corrupt the worker information causing the newly starting worker to crash (and having to start again later), this will no longer happen.
- Set statement time on start of every transaction in pglogical workers (RT64572)
  Fixes reporting of `xact_start` in `pg_stat_activity`

pglogical 3.6.1

pglogical 3.6.1 is the first minor release of the pglogical 3.6 series. This release includes minor new features and fixes including all the fixes from 3.6.0.1.

New Features

- Add slot failover documentation
- Add `pglogical.get_sub_progress_timestamp` for retrieving origin timestamp of the last committed change by the subscription

Resolved Issues

- Stop retrying subscription synchronization after unrecoverable error (RT64463)
  If the schema synchronization failed (which is an unrecoverable error) don't keep retrying forever. Instead mark the subscription synchronization as failed and disable the subscription.
- Improve handling and messaging with missing replication sets in output plugin (RT64451)
  Report all missing and found sets and make sure the sets are looked up using current snapshot.

pglogical 3.6.0.1

The pglogical 3.6.0.1 is the first bug-fix release in the pglogical 3.6 series.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Resolved Issues

- Improve synchronous `remote_write` replication performance (RT64397)
- Re-add support for binary protocol

pglogical 3.6.0

The version 3.6 of pglogical is a major update which brings performance improvements, better conflict handling, bug fixes and infrastructure necessary for BDR 3.6.

New Features

- Significant replication performance improvement
- Cache table synchronization state
- Only send keepalives when necessary
- Only do flush when necessary
- Serialize transactions in fewer cases in walsender (2ndQPostgres)
- Improved replication position reporting which is more in line with how physical streaming replication reports it
- Conflict detection and resolution improvements
- Add new types of conflicts (like `target_table_missing`)
- Add new types of conflict resolvers
- Make conflict resolution configurable by subscription and conflict type
- Improve conflict detection for updates

Resolved Issues

- Don't try to replicate REINDEX on temporary indexes

Other Improvements

- Fix potential message parsing error for two-phase commits
- Make initial COPY of data interruptible

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Appendix B: Known Issues

In this section we list a number of known issues that have not been addressed yet, each marked with an unique identifier.

- (RM17354, RM17554) Very large values can break pglogical replication. 2ndQPostgres v11.9r1.6.21 includes a fix for this issue; therefore BDR EE 3.6.21 or newer is not affected. The problem still exists when using community PostgreSQL, as in BDR SE.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Appendix C: Libraries

In this section we list the libraries used by pglogical3, with the corresponding licenses.

Library	License
LLVM	BSD (3-clause)
OpenSSL	SSLeay License AND OpenSSL License
Libpq	PostgreSQL License

LLVM

Copyright (c) 1994 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

OpenSSL

==

Copyright (c) 1998-2004 The OpenSSL Project. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgment: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org.
5. Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project.
6. Redistributions of any form whatsoever must retain the following acknowledgment: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/)"

THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

==

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com).

Original SSLeay Licence

Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com) All rights reserved.

This package is an SSL implementation written by Eric Young (eay@cryptsoft.com). The implementation was written so as to conform with Netscapes SSL.

This library is free for commercial and non-commercial use as long as the following conditions are aheared to. The following conditions apply to all code found in this distribution, be it the RC4, RSA, lhash, DES, etc., code; not just the SSL code. The SSL documentation included with this distribution is covered by the same copyright terms except that the holder is Tim Hudson (tjh@cryptsoft.com).

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

Copyright remains Eric Young's, and as such any Copyright notices in the code are not to be removed. If this package is used in a product, Eric Young should be given attribution as the author of the parts of the library used. This can be in the form of a textual message at program startup or in documentation (online or textual) provided with the package.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement: "This product includes cryptographic software written by Eric Young (eay@cryptsoft.com)" The word 'cryptographic' can be left out if the rouines from the library being used are not cryptographic related :-).
4. If you include any Windows specific code (or a derivative thereof) from the apps directory (application code) you must include an acknowledgement: "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"

THIS SOFTWARE IS PROVIDED BY ERIC YOUNG "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The licence and distribution terms for any publically available version or derivative of this code cannot be changed. i.e. this code cannot simply be copied and put under another distribution licence [including the GNU Public Licence.]

PostgreSQL License

PostgreSQL Database Management System (formerly known as Postgres, then as Postgres95)

Portions Copyright © 1996-2020, The PostgreSQL Global Development Group

Portions Copyright © 1994, The Regents of the University of California

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN “AS IS” BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Copyright © 2015-2020, 2ndQuadrant Limited. Copyright in these materials belongs to 2ndQuadrant Limited and no permissions or licences in relation to these materials are granted. No part of these materials may be reproduced in any form, stored in a retrieval system or transmitted in any way or by any means without the written permission of 2ndQuadrant Limited.