Information Classification: **PARTNER CONFIDENTIAL**

2ndQuadrant®
PostgreSQL

EnterpriseDB
Postgres-BDR

Version 3.6.33 Enterprise Edition
28 October 2022

BDR Development Team

Contents

Postgres-BDR

BDR (short for Bi-Directional Replication) is a PostgreSQL extension that provides a solution for building multi-master clusters with mesh topology. This means that you can write to any server and the changes will be sent sent row-by-row to all the other servers that are part of the same BDR group.

BDR version 3 ("BDR3") is built on the pglogical3 extension. However, everything you need to know about BDR3 is included here and it is unnecessary (and potentially confusing) to refer to pglogical docs.

This documentation refers only to BDR3, not to earlier architectures, referred to as BDR1 and BDR2. There are significant and important differences in BDR3, and you should not refer to earlier docs or rely on anything stated within them.

BDR3 comes in two variants:

- Standard Edition (BDR-SE), which runs on PostgreSQL 10+.
- Enterprise Edition (BDR-EE), which requires 2ndQPostgres 11 (2QPG11+) to run.

To provide very high availability, avoid data conflicts, and to cope with more advanced usage scenarios, the Enterprise Edition provides the following extensive additional features:

- Eager Replication - synchronizes between the nodes of the cluster before committing a transaction to provide conflict free replication
- Commit At Most Once - a consistency feature helping an application to commit each transaction only once, even in the presence of node failures
- Conflict-free Replicated Data Types - additional data types which provide mathematically proven consistency in asynchronous multi-master update scenarios
- Column Level Conflict Resolution - ability to use per column last-update wins resolution so that UPDATEs on different fields can be "merged" without losing either of them
- Transform Triggers - triggers that are executed on the incoming stream of data providing ability to modify it or to do advanced programmatic filtering
- Conflict triggers - triggers that are called when conflict is detected, providing a way to use custom conflict resolution techniques
- Timestamp-based Snapshots - providing consistent reads across multiple nodes for retrieving data as they appeared or will appear at a given time

This documentation is for the Enterprise Edition of BDR3.

BDR3 was developed by 2ndQuadrant Ltd and is used by many customers.

Architectural Overview

BDR provides loosely-coupled multi-master logical replication using a mesh topology. This means that you can write to any server and the changes will be sent directly, row-by-row to all the other servers that are part of the same BDR group.

Figure 1: node diagram

By default BDR uses asynchronous replication, applying changes on the peer nodes only after the local commit. An optional eager all node replication is available in the Enterprise Edition.

Basic Architecture

Multiple Groups

A BDR node is a member of at least one **Node Group**, and in the most basic architecture there is a single node group for the whole BDR cluster.

Multiple Masters

Each node (database) participating in a BDR group both receives changes from other members and can be written to directly by the user.

This is distinct from Hot or Warm Standby, where only one master server accepts writes, and all the other nodes are standbys that replicate either from the master or from another standby.

You do not have to write to all the masters, all of the time; it is a frequent configuration to direct writes mostly to just one master. However, if you just want one-way replication, the use of pglogical may be more appropriate.

Asynchronous, by default

Changes made on one BDR node are not replicated to other nodes until they are committed locally. As a result, the data is not exactly the same on all nodes at any given time; some nodes will have data that has not yet arrived at other nodes. PostgreSQL's block-based replication solutions default to asynchronous replication as well. In BDR, because there are multiple masters and as a result multiple data streams, data on different nodes might differ even when `synchronous_commit` and `synchronous_standby_names` are used.

Mesh Topology

BDR is structured around a mesh network where every node connects to every other node and all nodes exchange data directly with each other. There is no forwarding of data within BDR except in special circumstances such as node addition and node removal. Data may arrive from outside the BDR cluster or be sent onwards using pglogical or native PostgreSQL logical replication.

Logical Replication

Logical replication is a method of replicating data rows and their changes, based upon their replication identity (usually a primary key). We use the term *logical* in contrast to *physical* replication, which uses exact block addresses and byte-by-byte replication. Index changes are not replicated, thereby avoiding write amplification and reducing bandwidth.

Logical replication starts by copying a snapshot of the data from the source node. Once that is done, later commits are sent to other nodes as they occur in real time. Changes are replicated without re-executing SQL, so the exact data written is replicated quickly and accurately.

Nodes apply data in the order in which commits were made on the source node, ensuring that transactional consistency is guaranteed for the changes from any single node. Changes from different nodes are applied independently of other nodes to ensure the rapid replication of changes.

High Availability

Each master node can be protected by one or more standby nodes, so any node that goes down can be quickly replaced and continue. Each standby node can be a either a logical or a physical standby node.

Replication continues between currently connected nodes, even if one or more nodes are currently unavailable. When the node recovers, replication can restart from where it left off without missing any changes.

Nodes can run different release levels, negotiating the required protocols to communicate. As a result, BDR clusters can use rolling upgrades, even for major versions of database software.

DDL is automatically replicated across nodes by default. DDL execution can be user controlled to allow rolling application upgrades, if desired.

Limits

BDR has been tested with up to 99 master nodes in one cluster, but it is currently designed for use with up to 32 master nodes. Each master node can be protected by multiple physical or logical standby nodes. There is no specific limit on the number of standby nodes, but typical usage would be to have 2-3 standbys per master, with a typical maximum of 32 standbys per master.

When using timeshard sequences, BDR assumes there will be no more than 1024 nodes (counting both master nodes and logical standbys for the total), not counting nodes that have been previously removed (parted/dropped) from a group.

BDR places a limit that at most 10 databases in any one PostgreSQL instance can be BDR nodes across different BDR node groups. BDR does not support multiple nodes/databases within one instance being part of the same BDR node group.

Deployment

BDR3 is intended to be deployed in one of a small number of known-good configurations, using either TPAexec or a 2ndQuadrant-approved configuration management approach and deployment architecture.

Manual deployment is not recommended and may not be supported.

Please refer to the `TPAexec Architecture User Manual` for your architecture.

Log messages and documentation are currently available only in English.

Clocks and Timezones

BDR has been designed to operate with nodes in multiple timezones, allowing a truly worldwide database cluster. Individual servers do not need to be configured with matching timezones, though we do recommend using log_timezone = UTC to ensure the human-readable server log is more accessible and comparable.

Server clocks should be synchronized using NTP or other solutions.

Clock synchronization is not critical to performance, as is the case with some other solutions. Clock skew can impact Origin Conflict Detection, though BDR provides controls to report and manage any skew that exists. BDR also provides Row Version Conflict Detection, as described in Conflict Detection.

Application Usage

This chapter looks at BDR from an application or user perspective.

Setting up nodes is discussed in a later chapter, as is replication of DDL, and various options for controlling replication using replication sets.

Application Behavior

BDR will, by default, replicate all changes from INSERTs, UPDATEs, DELETEs and TRUNCATEs from the source node to other nodes. Only the final changes will be sent, after all triggers and rules have been processed. For example, INSERT ... ON CONFLICT UPDATE will send either an INSERT or an UPDATE depending on what occurred on the origin. If an UPDATE or DELETE affects zero rows, then no changes will be sent.

INSERTs can be replicated without any pre-conditions.

For UPDATEs and DELETEs to be replicated on other nodes, we must be able to identify the unique rows affected. BDR requires that a table have either a PRIMARY KEY defined, a UNIQUE constraint or have a REPLICATION IDENTITY defined. If one of those is not defined, a WARNING will be generated, and later UPDATEs or DELETEs will be explicitly blocked to prevent nodes from ending with differing contents, a situation which we describe as *divergence*. BDR has many mechanisms designed to protect against divergence, and these are described later in the documentation.

TRUNCATE can be used even without a defined replication identity. Replication of TRUNCATE commands is supported, but some care must be taken when truncating groups of tables connected by foreign keys. When replicating a truncate action, the subscriber will truncate the same group of tables that was truncated on the origin, either explicitly specified or implicitly collected via CASCADE, except in cases where replication sets are defined, see Replication Sets chapter for further details and examples. This will work correctly if all affected tables are part of the same subscription. But if some tables to be truncated on the subscriber have foreign-key links to tables that are not part of the same (or any) replication set, then the application of the truncate action on the subscriber will fail. TRUNCATE requires some form of locking or lock avoidance to avoid divergent data errors, see TRUNCATE Conflicts.

Row-level locks taken implicitly by INSERT, UPDATE and DELETE commands will be replicated as the changes are made. Table-level locks taken implicitly by INSERT, UPDATE, DELETE and TRUNCATE commands will also be replicated. Explicit row-level locking (SELECT ... FOR UPDATE/FOR SHARE) by user sessions is not replicated, nor are advisory locks.

Sequences need special handling, described in the Sequences chapter.

Binary data in BYTEA columns is replicated normally, allowing “blobs” of data up to 1GB in size. Data stored in the PostgreSQL “Large object” facility is not replicated.

Replication is only possible from base tables to base tables. That is, the tables on the source and target on the subscription side must be tables, not views, materialized views, or foreign tables. Attempts to replicate tables other than base tables will result in an error.

Partitioned tables are supported by BDR3, but only on PostgreSQL 11+ or 2ndQPostgres 11+ because of differences in internal APIs utilized. It is possible to replicate between tables with dissimilar partitioning definitions, such as a source which is a normal table replicating to a partitioned table, including support for updates that change partitions on the target. It can be faster if the partitioning definition is the same on the source and target since dynamic partition routing need not be executed at apply time. Further details are available in the chapter on Replication Sets.

By default, triggers execute only on the origin node. For example, an INSERT trigger executes on the origin node and is ignored when we apply the change on the target node. You can specify that triggers should execute on both the origin node at execution time and on the target when it is replicated ("apply time") by using `ALTER TABLE ... ENABLE ALWAYS TRIGGER`, or use the `REPLICA` option to execute only at apply time, `ALTER TABLE ... ENABLE REPLICA TRIGGER`.

Some types of trigger are not executed on apply, even if they exist on a table and are currently enabled. Trigger types not executed are

- Statement-level triggers (FOR EACH STATEMENT)
- Per-column UPDATE triggers (UPDATE OF column_name [, ...])

BDR replication apply uses the system-level default search_path. Replica triggers, stream triggers and index expression functions may assume other search_path settings which will then fail when they execute on apply. To ensure this does not occur, resolve object references clearly using either the default search_path only, always use fully qualified references to objects, e.g. schema.objectname, or set the search path for a function using `ALTER FUNCTION ... SET search_path = ...` for the functions affected.

Note that BDR assumes that there are no issues related to text or other collatable datatypes, i.e. all collations in use are available on all nodes and the default collation is the same on all nodes. Replication of changes uses equality searches to locate Replica Identity values, so this will not have any effect except where unique indexes are explicitly defined with non-matching collation qualifiers. Row filters might be affected by differences in collations if collatable expressions were used.

BDR handling of very-long "toasted" data within PostgreSQL is transparent to the user. Note that the TOAST "chunkid" values will likely differ between the same row on different nodes, but that does not cause any problems.

BDR cannot work correctly if Replica Identity columns are marked as "external".

PostgreSQL allows CHECK() constraints that contain volatile functions. Since BDR re-executes CHECK() constraints on apply, any subsequent re-execution that doesn't return the same result as previously will cause data divergence.

BDR does not restrict the use of Foreign Keys; cascading FKs are allowed.

BDR does not currently support the use of non-ASCII schema or relation names. Later versions will remove this restriction.

Transaction Handling

BDR supports all standard transaction options, including savepoints.

All transaction isolation levels are supported, including SERIALIZABLE. SERIALIZABLE level applies only to transactions executed on the local node; transactions executing in SERIALIZABLE mode on other nodes have no effect on transactions executing locally, so some transactions may be allowed for which there is no global/cluster-wide serializable ordering. Workloads that require this should be executed together on one BDR node at a time.

BDR does not restrict the use of temporary tables, though PostgreSQL does prevent temporary tables being used with prepared statements (aka 2PC, XA).

There are no defined limits on the duration or size of transactions with BDR.

BDR applies transactions once they have been committed on the origin node. Larger transactions will take longer to transmit and apply, so can induce replication lag for other transactions. Users are recommended to limit the size of transactions by breaking larger data loads into smaller chunks, or using procedures that commit regularly. Note that the duration of a transaction has no impact, only the number of changes and/or the total data size of the changes has effect.

Non-replicated statements

None of the following user commands are replicated by BDR, so their effects occur on the local/origin node only:

- Cursor operations (DECLARE, CLOSE, FETCH)
- Execution commands (DO, CALL, PREPARE, EXECUTE, EXPLAIN)
- Session management (DEALLOCATE, DISCARD, LOAD)
- Parameter commands (SET, SHOW)
- Constraint manipulation (SET CONSTRAINTS)
- Locking commands (LOCK)
- Table Maintenance commands (VACUUM, ANALYZE, CLUSTER)
- Async operations (NOTIFY, LISTEN, UNLISTEN)

Note that since the `NOTIFY` SQL command and the `pg_notify()` functions are not replicated, notifications are *not* reliable in case of failover. This means that notifications could easily be lost at failover if a transaction is committed just at the point the server crashes. Applications running `LISTEN` may miss notifications in case of failover. This is regrettably true in standard PostgreSQL replication and BDR does not yet improve on this. CAMO and Eager replication options do not allow the `NOTIFY` SQL command or the `pg_notify()` function.

Replicating between different release levels

BDR is designed to replicate between nodes that have different major versions of PostgreSQL. This is a feature designed to allow major version upgrades without downtime.

BDR is also designed to replicate between nodes that have different versions of BDR software. This is a feature designed to allow version upgrades and maintenance without downtime.

However, while it's possible to join a node with a major version in a cluster, you can not add a node with a minor version if the cluster uses a newer protocol version, this will return error.

Both of the above features may be affected by specific restrictions; any known incompatibilities will be described in the release notes.

Replicating between nodes with differences

By default, DDL will automatically be sent to all nodes. This can be controlled manually, as described in DDL Replication, which could be used to create differences between database schemas across nodes. BDR is designed to allow replication to continue even while minor differences exist between nodes. These features are designed to allow application schema migration without downtime, or to allow logical standby nodes for reporting or testing.

Currently, replication requires the same table name on all nodes. A future feature may allow a mapping between different table names.

We can only replicate between tables with the same column names. If a column has the same name but a different datatype, we attempt to cast from the source type to the target type, if casts have been defined that allow that.

By default, all columns are replicated. BDR supports replicating between tables that have different number of columns.

If the target has missing column(s) from the source then BDR will raise a target_column_missing conflict, for which the default conflict resolver is ignore_if_null. This will throw an ERROR if a non-NULL value arrives. Alternatively, a node can also be configured with a conflict resolver of ignore. This setting will not throw an ERROR, just silently ignore any additional columns.

If the target has additional column(s) not seen in the source record then BDR will raise a source_column_missing conflict, for which the default conflict resolver is use_default_value. Replication will proceed if the additional columns have a default, either NULL (if nullable) or a default expression, but will throw an ERROR and halt replication if not.

Transform triggers can also be used on tables to provide default values or alter the incoming data in various ways before apply.

If the source and the target have different constraints, then replication will be attempted, but it might fail if the rows from source cannot be applied to the target. Row filters may help here.

Replicating data from one schema to a more relaxed schema won't cause failures. Replicating data from a schema to a more restrictive schema will be a source of potential failures. The right way to solve this is to place a constraint on the more relaxed side, so bad data is prevented from being entered. That way, no bad data ever arrives via replication, so it will never fail the transform into the more restrictive schema. For example, if one schema has a column of type TEXT and another schema defines the same column as XML, add a CHECK constraint onto the TEXT column that enforces that the text is XML.

A table may be defined with different indexes on each node. By default, the index definitions will be replicated. Refer to DDL Replication to specify how to create an index only on a subset of nodes, or just locally.

Storage parameters, such as fillfactor and toast_tuple_target, may differ between nodes for a table without problems. An exception to that is the value of a table's storage parameter `user_catalog_table` must be identical on all nodes.

A table being replicated should be owned by the same user/role on each node. Refer to Security and Roles for further discussion.

Roles may have different passwords for connection on each node, though by default changes to roles are replicated to each node. Refer to DDL Replication to specify how to alter a role password only on a subset of nodes, or just locally.

Timing Considerations and Synchronous Replication

Being asynchronous by default, peer nodes may lag behind making it's possible for a client connected to multiple BDR nodes or switching between them to read stale data. A queue wait function is provided for clients or proxies to prevent such stale reads.

In addition, BDR provides multiple variants for more synchronous replication. Please refer to the Durability & Performance Options chapter for an overview and comparison of all variants available and its different modes.

Application Testing

BDR applications can be tested using the following programs, in addition to other techniques.

- TPAexec
- pgbench with CAMO/Failover options
- isolationtester with multi-node access

TPAexec

TPAexec is the system used by 2ndQuadrant to deploy reference TPA architectures, including those based on Postgres-BDR.

TPAexec includes test suites for each reference architecture; it also simplifies creating and managing a local collection of tests to be run against a TPA cluster, using a syntax as in the following example:

```
tpaexec test mycluster mytest
```

We strongly recommend that developers write their own multi-node suite of TPAexec tests which verify the main expected properties of the application.

pgbench with CAMO/Failover options

pgbench has been extended to allow users to run failover tests while using CAMO or regular BDR deployments. The following new options have been added:

```
-m, --mode=regular|camo|failover
mode in which pgbench should run (default: regular)

--retry
retry transactions on failover
```

in addition to the above options, the connection information about the peer node for failover must be specified in DSN form.

- Use `-m camo` or `-m failover` to specify the mode for pgbench. The `-m failover` specification can be used to test failover in regular BDR deployments.
- Use `--retry` to specify whether transactions should be retried when failover happens with `-m failover` mode. This is enabled by default for `-m camo` mode.

Here's an example invocation in a CAMO environment:

```
pgbench -m camo -p $node1_port -h $node1_host bdrdemo \
    "host=$node2_host␣user=postgres␣port=$node2_port␣dbname=bdrdemo"
```

The above command will run in `camo` mode. It will connect to `node1` and run the tests; if the connection to `node1` connection is lost, then pgbench will connect to `node2`. It will query `node2` to get the status of in-flight transactions. Aborted and in-flight transactions will be retried in `camo` mode.

In `failover` mode, if `--retry` is specified then in-flight transactions will be retried. In this scenario there is no way to find the status of in-flight transactions.

isolationtester with multi-node access

isolationtester has been extended to allow users to run tests on multiple sessions and on multiple nodes. This is used for internal BDR testing, though it is also available for use with user application testing.

```
$ isolationtester \
     --outputdir=./iso_output \
     --create-role=logical \
     --dbname=postgres \
     --server 'd1=dbname=node1' \
     --server 'd2=dbname=node2' \
     --server 'd3=dbname=node3'
```

Isolation tests are a set of tests run for examining concurrent behaviors in PostgreSQL. These tests require running multiple interacting transactions, which requires management of multiple concurrent

connections, and therefore can't be tested using the normal `pg_regress` program. The name "isolation" comes from the fact that the original motivation was to test the serializable isolation level; but tests for other sorts of concurrent behaviors have been added as well.

It is built using PGXS as an external module. On installation, it creates isolationtester binary file which is run by `pg_isolation_regress` to perform concurrent regression tests and observe results.

`pg_isolation_regress` is a tool similar to `pg_regress`, but instead of using psql to execute a test, it uses isolationtester. It accepts all the same command-line arguments as `pg_regress`. It has been modified to accept multiple hosts as parameters. It then passes these host conninfo's along with server names to isolationtester binary. Isolation tester compares these server names with the names specified in each session in the spec files and runs given tests on respective servers.

To define tests with overlapping transactions, we use test specification files with a custom syntax, which is described in the next section. To add a new test, place a spec file in the specs/ subdirectory, add the expected output in the expected/ subdirectory, and add the test's name to the Makefile.

Isolationtester is a program that uses libpq to open multiple connections, and executes a test specified by a spec file. A libpq connection string specifies the server and database to connect to; defaults derived from environment variables are used otherwise.

Specification consists of five parts, tested in this order:

`server "<name>"`

This defines the name of the servers that the sessions will run on. There can be zero or more server "" specifications. The conninfo corresponding to the names is provided via the command to run isolationtester. This is described in `quickstart_isolationtest.md`. This part is optional.

`setup { <SQL> }`

The given SQL block is executed once, in one session only, before running the test. Create any test tables or other required objects here. This part is optional. Multiple setup blocks are allowed if needed; each is run separately, in the given order. (The reason for allowing multiple setup blocks is that each block is run as a single PQexec submission, and some statements such as VACUUM cannot be combined with others in such a block.)

`teardown { <SQL> }`

The teardown SQL block is executed once after the test is finished. Use this to clean up in preparation for the next permutation, e.g dropping any test tables created by setup. This part is optional.

`session "<name>"`

There are normally several "session" parts in a spec file. Each session is executed in its own connection. A session part consists of three parts: setup, teardown and one or more "steps". The per-session setup and teardown parts have the same syntax as the per-test setup and teardown described above, but they are executed in each session. The setup part typically contains a "BEGIN" command to begin a transaction.

Additionally, a session part also consists of `connect_to` specification. This points to server name specified in the beginning which indicates the server on which this session runs.

`connect_to "<name>"`

Each step has the syntax

`step "<name>" { <SQL> }`

where <name> is a name identifying this step, and SQL is a SQL statement (or statements, separated by semicolons) that is executed in the step. Step names must be unique across the whole spec file.

`permutation "<step name>"`

A permutation line specifies a list of steps that are run in that order. Any number of permutation lines can appear. If no permutation lines are given, the test program automatically generates all possible orderings of the steps from each session (running the steps of any one session in order). Note that the list of steps in a manually specified "permutation" line doesn't actually have to be a permutation of the available steps; it could for instance repeat some steps more than once, or leave others out.

Lines beginning with a # are considered comments.

For each permutation of the session steps (whether these are manually specified in the spec file, or automatically generated), the isolation tester runs the main setup part, then per-session setup parts, then the selected session steps, then per-session teardown, then the main teardown script. Each selected step is sent to the connection associated with its session.

To run isolation tests in a BDR3 environment thats ran all prerequisite make commands, follow the below steps,

1. Run `make isolationcheck-install` to install the isolationtester submodule
2. You can run isolation regression tests using either of the following commands from the bdr-private repo

`make isolationcheck-installcheck` `make isolationcheck-makecheck`

A. To run isolationcheck-installcheck, you need to have two or more postgresql servers running. Pass the conninfo's of servers to `pg_isolation_regress` in BDR 3.0 Makefile. Ex: `pg_isolation_regress --server 'd1=host=myhost dbname=mydb port=5434' --server 'd2=host=myho`

Now, add a .spec file containing tests in specs/isolation directory of bdr-private/ repo. Add .out file in expected/isolation directory of bdr-private/ repo.

Then run `make isolationcheck-installcheck`

B. Isolationcheck-makecheck currently supports running isolation tests on a single instance by setting up BDR between multiple databases.

You need to pass appropriate database names, conninfos of bdr instances to `pg_isolation_regress` in BDR Makefile as follows: `pg_isolation_regress --dbname=db1,db2 --server 'd1=dbname=db1' --server`

Then run `make isolationcheck-makecheck`

Each step may contain commands that block until further action has been taken (most likely, some other session runs a step that unblocks it or causes a deadlock). A test that uses this ability must manually

specify valid permutations, i.e. those that would not expect a blocked session to execute a command. If a test fails to follow that rule, isolationtester will cancel it after 300 seconds. If the cancel doesn't work, isolationtester will exit uncleanly after a total of 375 seconds of wait time. Testing invalid permutations should be avoided because they can make the isolation tests take a very long time to run, and they serve no useful testing purpose.

Note that isolationtester recognizes that a command has blocked by looking to see if it is shown as waiting in the `pg_locks` view; therefore, only blocks on heavyweight locks will be detected.

Performance Testing & Tuning

BDR allows you to issue write transactions onto multiple master nodes. Bringing those writes back together onto each node has a cost in performance that you should be aware of.

First, replaying changes from another node has a CPU cost, an I/O cost and it will generate WAL records. The resource usage is usually less than in the original transaction since CPU overheads are lower as a result of not needing to re-execute SQL. In the case of UPDATE and DELETE transactions there may be I/O costs on replay if data isn't cached.

Second, replaying changes holds table-level and row-level locks that can produce contention against local workloads. The CRDTs and CLCD features ensure you get the correct answers even for concurrent updates, but they don't remove the normal locking overheads. If you get locking contention, try to avoid conflicting updates and/or keep transactions as short as possible. A heavily updated row within a larger transaction will cause a bottleneck on performance for that transaction. Complex applications require some thought to maintain scalability.

If you think your are having performance problems, you are encouraged to develop performance tests using the benchmarking tools above. pgbench allows you to write custom test scripts specific to your use case so you can understand the overheads of your SQL and measure the impact of concurrent execution.

So if "BDR is running slow", then we suggest the following:

1. Write a custom test script for pgbench, as close as you can make it to the production system's problem case.
2. Run the script on one node to give you a baseline figure.
3. Run the script on as many nodes as occurs in production, using the same number of sessions in total as you did on one node. This will show you the effect of moving to multiple nodes.
4. Increase the number of sessions for the above 2 tests, so you can plot the effect of increased contention on your application.
5. Make sure your tests are long enough to account for replication delays.
6. Ensure that replication delay isn't growing during your tests.

Use all of the normal Postgres tuning features to improve the speed of critical parts of your application.

Assessing Suitability

BDR is compatible with PostgreSQL, but not all PostgreSQL applications are suitable for use on distributed databases. Most applications are already, or can be easily modified to become, BDR compliant. Users can undertake an assessment activity in which they can point their application to a BDR-enabled setup. BDR provides a few knobs which can be set during the assessment period. These will aid in the process of deciding suitability of their application in a BDR-enabled environment.

Assessing updates of Primary Key/Replica Identity

BDR cannot currently perform conflict resolution where the PRIMARY KEY is changed by an UPDATE operation. It is permissible to update the primary key, but you must ensure that no conflict with existing values is possible.

When used with 2ndQPostgres 11+, BDR provides the following configuration parameter to assess how frequently the primary key/replica identity of any table is being subjected to update operations.

Note that these configuration parameters must only be used for assessment only. They can be used on a single node BDR instance, but must not be used on a production BDR cluster with two or more nodes replicating to each other. In fact, a node may fail to start or a new node will fail to join the cluster if any of the assessment parameter is set to anything other than `IGNORE`.

```
bdr.assess_update_replica_identity = IGNORE (default) | LOG | WARNING | ERROR
```

By enabling this parameter during the assessment period, one can log updates to the key/replica identity values of a row. One can also potentially block such updates, if desired. E.g.

```
CREATE TABLE public.test(g int primary key, h int);
INSERT INTO test VALUES (1, 1);

SET bdr.assess_update_replica_identity TO 'error';
UPDATE test SET g = 4 WHERE g = 1;
ERROR:  bdr_assess: update of key/replica identity of table public.test
```

Apply worker processes will always ignore any settings for this parameter.

Assessing use of LOCK on tables or in SELECT queries

Because BDR writer processes operate much like normal user sessions, they are subject to the usual rules around row and table locking. This can sometimes lead to BDR writer processes waiting on locks held by user transactions, or even by each other.

When used with 2ndQPostgres 11+, BDR provides the following configuration parameter to assess if the application is taking explicit locks.

```
bdr.assess_lock_statement = IGNORE (default) | LOG | WARNING | ERROR
```

Two types of locks that can be tracked are:

- explicit table-level locking (LOCK TABLE . . .) by user sessions
- explicit row-level locking (SELECT . . . FOR UPDATE/FOR SHARE) by user sessions

By enabling this parameter during the assessment period, one can track (or block) such explicit locking activity. E.g.

```
CREATE TABLE public.test(g int primary key, h int);
INSERT INTO test VALUES (1, 1);

SET bdr.assess_lock_statement TO 'error';
SELECT * FROM test FOR UPDATE;
ERROR:  bdr_assess: "SELECT␣FOR␣UPDATE" invoked on a BDR node

SELECT * FROM test FOR SHARE;
ERROR:  bdr_assess: "SELECT␣FOR␣SHARE" invoked on a BDR node

SET bdr.assess_lock_statement TO 'warning';
LOCK TABLE test IN ACCESS SHARE MODE;
WARNING:  bdr_assess: "LOCK␣STATEMENT" invoked on a BDR node
```

PostgreSQL Configuration for BDR

There are several PostgreSQL configuration parameters that affect BDR.

PostgreSQL Settings for BDR

BDR requires certain PostgreSQL settings to be set to appropriate values:

- `wal_level` - Must be set to `logical` for logical decoding (which BDR uses) to work.
- `shared_preload_libraries` - This must contain `pglogical,bdr` (in that order).
- `max_worker_processes` - BDR uses background workers for replication and maintenance tasks, so there need to be enough worker slots for it to work correctly. The formula for the correct minimal number of workers is: one per PostgreSQL instance + one per database on that instance + two per BDR-enabled database + two per peer node in the BDR group for each database.
- `max_wal_senders` - Two needed per every peer node.
- `max_replication_slots` - Same as `max_wal_senders`.
- `track_commit_timestamp` - Must be set to 'on' for conflict resolution to work.
- `wal_sender_timeout` and `wal_receiver_timeout` - Determine how quickly an origin considers its CAMO partner as disconnected or reconnected; see CAMO Failure Scenarios for details.
- `synchronous_commit` - affects the durability and performance of BDR replication in a similar way to physical replication.

Note that in normal running for a group with N peer nodes, BDR will require N slots/walsenders. During synchronization, BDR will temporarily use another N - 1 slots/walsenders, so be careful to set the above parameters high enough to cater for this occasional peak demand.

2ndQPostgres Settings for BDR

The following Postgres settings need to be considered for commit at most once (CAMO), a feature that is only available for BDR in combination with 2ndQPostgres. Some of these are only available in 2ndQPostgres; others already exist in the community version, but only become relevant with BDR in combination with CAMO.

- `pg2q.enable_camo` - Used to enable and control the CAMO feature. Defaults to `off`. CAMO can be switched on per transaction by setting this to `remote_write`, `remote_commit_async`, or `remote_commit_flush`. For backwards-compatibility, the values `on`, `true`, and `1` set the safest `remote_commit_flush` mode. While `false` or `0` also disable CAMO.
- `report_transaction_id` - Deprecated, now automatically enabled together with `pg2q.enable_camo`. This config parameter can and should be removed from the global, session- and transaction-wide configuration.

- `synchronous_replication_availability` - Can optionally be `async` for increased availability by allowing an origin to continue and commit after its CAMO partner got disconnected. Under the default value of `wait`, the origin will wait indefinitely, and proceed to commit only after the CAMO partner reconnects and sends confirmation.
- `snapshot_timestamp` - Turns on the usage of timestamp-based snapshots and sets the timestamp to use.

pglogical Settings for BDR

BDR is also affected by some of the pglogical settings as it uses pglogical internally to implement the basic replication.

- `pglogical.track_subscription_apply` - Track apply statistics for each subscription.
- `pglogical.track_relation_apply` - Track apply statistics for each relation.
- `pglogical.track_apply_lock_timing` - Track lock timing when tracking statistics for relations.
- `pglogical.standby_slot_names` - When using physical Standby nodes intended for failover purposes, should be set to the replication slot(s) for each intended Standby.

BDR Specific Settings

There are also BDR specific configuration settings that can be set. Unless noted otherwise, values may be set by any user at any time.

Conflict Handling

- `bdr.default_conflict_detection` - Sets the default conflict detection method for newly created tables; accepts same values as bdr.alter_table_conflict_detection()

Global Sequence Parameters

- `bdr.default_sequence_kind` - Sets the default sequence kind.

DDL Handling

- `bdr.default_replica_identity` - Sets the default value for `REPLICA IDENTITY` on newly created tables. The `REPLICA IDENTITY` defines which information is written to the write-ahead log to identify rows which are updated or deleted.

  The accepted values are:

  - `DEFAULT` - records the old values of the columns of the primary key, if any (this is the default PostgreSQL behavior).
  - `FULL` - records the old values of all columns in the row.

  - `NOTHING` - records no information about the old row.

  See PostgreSQL documentation for more details.

  BDR can not replicate `UPDATE`s and `DELETE`s on tables without a `PRIMARY KEY` or `UNIQUE` constraint, unless the replica identity for the table is `FULL`, either by table-specific configuration or via `bdr.default_replica_identity`.

  If bdr.default_replica_identity is DEFAULT and there is a `UNIQUE` constraint on the table, it will not be automatically picked up as `REPLICA IDENTITY`. It needs to be set explicitly at the time of creating the table, or afterwards as described in the documentation above.

  Setting the replica identity of table(s) to `FULL` increases the volume of WAL written and the amount of data replicated on the wire for the table.

- `bdr.ddl_replication` - Automatically replicate DDL across nodes (default "on").

  This parameter can be only set by the bdr_superuser or superuser roles. However it is possible to change this setting using the `bdr.set_ddl_replication()` function, which can be `GRANT`ed to other users.

  Running DDL or calling BDR administration functions with `bdr.ddl_replication = off` can create situations where replication stops until an administrator can intervene. See the DDL replication chapter for details.

  A `LOG`-level log message is emitted to the PostgreSQL server logs whenever `bdr.ddl_replication` is set to `off`. Additionally, a `WARNING`-level message is written whenever replication of captured DDL commands or BDR replication functions is skipped due to this setting.

- `bdr.role_replication` - Automatically replicate ROLE commands across nodes (default "on"). This parameter is settable by a superuser only. This setting only works if `bdr.ddl_replication` is turned on as well.

  Turning this off without using external methods to ensure roles are in sync across all nodes may cause replicated DDL to interrupt replication until the administrator intervenes.

  See Role manipulation statements in the DDL replication chapter for details.

- `bdr.ddl_locking` - Configures the operation mode of global locking for DDL.

  This parameter can be only set by bdr_superuser or superuser roles. However it is possible to change this setting using the `bdr.set_ddl_locking()` function, which can be `GRANT`ed to other users.

  Possible options are:

  - off - do not use global locking for DDL operations
  - on - use global locking for all DDL operations
  - dml - only use global locking for DDL operations that need to prevent writes by taking the global DML lock for a relation

A LOG-level log message is emitted to the PostgreSQL server logs whenever `bdr.ddl_replication` is set to `off`. Additionally, a `WARNING` message is written whenever any global locking steps are skipped due to this setting. It is normal for some statements to result in two `WARNING`s, one for skipping the DML lock and one for skipping the DDL lock.

- `bdr.truncate_locking` - Configures the `TRUNCATE` locking behavior. If set to `true`, the `TRUNCATE` command will obey the `bdr.ddl_locking` setting - this is the behavior in BDR 3.7 and newer. When set to `false` (default) the `TRUNCATE` command will not lock - this is the behavior of BDR 3.6.10 and older.

Global Locking

- `bdr.ddl_locking` - Described above.
- `bdr.global_lock_max_locks` - Maximum number of global locks that can be held on a node (default 1000). May only be set at Postgres server start.
- `bdr.global_lock_timeout` - Sets the maximum allowed duration of any wait for a global lock (default 1 minute). A value of zero disables this timeout.
- `bdr.global_lock_statement_timeout` - Sets the maximum allowed duration of any statement holding a global lock (default 10 minutes). A value of zero disables this timeout.
- `bdr.global_lock_idle_timeout` - Sets the maximum allowed duration of idle time in transaction holding a global lock (default 10 minutes). A value of zero disables this timeout.

Node Management

- `bdr.replay_progress_frequency` - Interval for sending replication position info to the rest of the cluster (default 1 minute).

Generic Replication

- `bdr.xact_replication` - Replicate current transaction (default “on”).

  Turning this off will make the whole transaction local only.

  This parameter can be only set by the bdr_superuser or superuser roles.

  This parameter can only be set inside the current transaction using the `SET LOCAL` command; it cannot be set in the configuration file or the user default configuration.

> **Note**
> Even with transaction replication disabled, WAL will be generated but those changes will be filtered away on the origin.

> **Warning**
> Turning off `bdr.xact_replication` *will* lead to data inconsistency between nodes, and should only be used to recover from data divergence between nodes or in replication situations where changes on single nodes are required for replication to continue. Use at your own risk.

- `bdr.permit_unsafe_commands` - Option to override safety check on commands that are deemed unsafe for general use.

  Requires `bdr_superuser` or PostgreSQL superuser.

> **Warning**
> The commands that are normally not considered safe may either produce inconsistent results or break replication altogether. Use at your own risk.

- `bdr.maximum_clock_skew`

  This specifies what should be considered as the maximum difference between the incoming transaction commit timestamp and the current time on the subscriber before triggering `bdr.maximum_clock_skew_action`.

  This checks if the timestamp of the currently replayed transaction is in the future compared to the current time on the subscriber; and if it is, and the difference is larger than `bdr.maximum_clock_skew`, it will do the action specified by the `bdr.maximum_clock_skew_action` setting.

  The default is -1, which means: ignore clock skew (the check is turned off). It is valid to set 0 as when the clock on all servers are synchronized, the fact that we are replaying the transaction means it has been committed in the past.

- `bdr.maximum_clock_skew_action`

  This specifies the action to take if a clock skew higher than `bdr.maximum_clock_skew` is detected.

  There are two possible values for this option:

  - `WARN` - Log a warning about this fact. The warnings are logged once per minute (the default) at the maximum to prevent flooding the server log.
  - `WAIT` - Wait for as long as the current local timestamp is no longer older than remote commit timestamp minus the `bdr.maximum_clock_skew`.

CRDTs

- `bdr.crdt_raw_value` - Sets the output format of CRDT Data Types. The default output (when this setting is `off`) is to return only the current value of the base CRDT type (for example, a bigint for `crdt_pncounter`). When set to `on`, the returned value represents the full representation of the CRDT value, which can for example include the state from multiple nodes.

Max Prepared Transactions

- `max_prepared_transactions` - Needs to be set high enough to cope with the maximum number of concurrent prepared transactions across the cluster due to explicit two-phase commits, CAMO or Eager transactions. Exceeding the limit prevents a node from running a local two-phase commit or CAMO transaction, and will prevent all Eager transactions on the cluster. May only be set at Postgres server start.

Eager Replication

- `bdr.commit_scope` - Setting the commit scope to `global` enables eager all node replication (default `local`).
- `bdr.global_commit_timeout` - Timeout for both stages of a global two-phase commit (default 60s) as well as for CAMO-protected transactions in their commit phase, as a limit for how long to wait for the CAMO partner.

Commit at Most Once

- `bdr.camo_partner_of` - Allows specifying a CAMO partner per database. Expects pairs of database name and node name joined by a colon. Multiple pairs may be specified, but only the first occurrence per database is taken into account. For example: `'db1:node_4 test_db:test_node_3'`. May only be set at Postgres server start.
- `bdr.camo_origin_for` - Per-database node name of the origin of transactions in a CAMO pairing; for each database, this needs to match with the `bdr.camo_partner_of` setting on the corresponding origin node. May only be set at Postgres server start.
- `bdr.standby_dsn` - Allows manual override of the connection string (DSN) to reach the CAMO partner, in case it has changed since the crash of the local node. Should usually be unset. May only be set at Postgres server start.
- `bdr.camo_local_mode_delay` - The commit delay that applies in CAMO's Local mode to emulate the overhead that normally occurs with the CAMO partner having to confirm transactions. Defaults to 5 ms. Setting to 0 disables this feature.
- `bdr.camo_enable_client_warnings` - Emit warnings if an activity is carried out in the database for which CAMO properties cannot be guaranteed. This is enabled by default. Well-informed users can choose to disable this to reduce the amount of warnings going into their logs.

Timestamp-based Snapshots

- `bdr.timestamp_snapshot_keep` - For how long to keep valid snapshots for the timestamp-based snapshot usage (default 0, meaning do not keep past snapshots). Also see `snapshot_timestamp` above.

Monitoring and Logging

- `bdr.debug_level` - Defines the log level that BDR uses to write its debug messages. The default value is `debug2`. If you want to see detailed BDR debug output, set `bdr.debug_level = 'log'`.
- `bdr.trace_level` - Similar to the above, this defines the log level to use for BDR trace messages. Enabling tracing on all nodes of a BDR cluster may help 2ndQuadrant Support to diagnose issues. May only be set at Postgres server start.

> **Warning**
> Setting `bdr.debug_level` or `bdr.trace_level` to a value >= `log_min_messages` can produce a very large volume of log output, so it should not be enabled long term in production unless plans are in place for log filtering, archival and rotation to prevent disk space exhaustion.

Internals

- `bdr.raft_keep_min_entries` - The minimum number of entries to keep in the Raft log when doing log compaction (default 100). The value of 0 will disable log compaction. **WARNING: If log compaction is disabled, the log will grow in size forever.** May only be set at Postgres server start.
- `bdr.raft_response_timeout` - To account for network failures, the Raft consensus protocol implemented will time out requests after a certain amount of time. This timeout defaults to 30 seconds.
- `bdr.raft_log_min_apply_duration` - To move the state machine forward, Raft appends entries to its internal log. During normal operation, appending takes only a few milliseconds. This poses an upper threshold on the duration of that append action, above which an `INFO` message is logged. This may indicate an actual problem. Default value of this parameter is 3000 ms.
- `bdr.raft_log_min_message_duration` - When to log a consensus request. Measure round trip time of a bdr consensus request and log an `INFO` message if the time exceeds this parameter. Default value of this parameter is 5000 ms.
- `bdr.backwards_compatibility` - Specifies the version to be backwards-compatible to, in the same numerical format as used by `bdr.bdr_version_num`, e.g. `30618`. Enables exact behavior of a former BDR version, even if this has generally unwanted effects. Defaults to the current BDR version. Since this changes from release to release, we advise against explicit use within the configuration file unless the value is different to the current version.
- `bdr.track_replication_estimates` - Track replication estimates in terms of apply rates and catchup intervals for peer nodes. This information can be used by protocols like CAMO to estimate the readiness of a peer node. This parameter is enabled by default.
- `bdr.lag_tracker_apply_rate_weight` - We monitor how far behind peer nodes are in terms of applying WAL from the local node, and calculate a moving average of the apply rates for the lag tracking. This parameter specifies how much contribution newer calculated values have in this moving average calculation. Default value is 0.1.

Node Management

Each database that is member of a BDR group must be represented by its own node. A node is an unique identifier of such a database in the BDR group.

At present, each node can be a member of just one node group; this may be extended in later releases. Each node may subscribe to one or more Replication Sets to give fine-grained control over replication.

Creating and Joining a BDR Group

For BDR, every node has to have a connection to every other node. To make configuration easy, when a new node joins, it automatically configures all existing nodes to connect to it. For this reason, every node, including the first BDR node created, must know the PostgreSQL connection string (sometimes referred to as a DSN, for "data source name") that other nodes can use to connect to it. Both formats of connection string are supported. So you can use either key-value format like `host=myhost port=5432 dbname=mydb` or URI format `postgresql://myhost:5432/mydb`.

The SQL function `bdr.create_node_group()` is used to create the BDR group from the local node. Doing so activates BDR on that node and allows other nodes to join the BDR group (which consists only of one node at that point). You must specify the connection string that other nodes will use to connect to this node at the time of creation.

Once the node group is created, every further node can join the BDR group using the `bdr.join_node_group()` function.

Alternatively, the command line utility `bdr_init_physical` can be used to create a new node using `pg_basebackup` (or a physical standby) of an existing node. If using `pg_basebackup`, the `bdr_init_physical` utility can optionally specify the base backup of the target database only as opposed to the earlier behavior of backup of the entire database cluster. This should make this activity complete faster and also allow it to use less space due to the exclusion of unwanted databases. If only the target database is specified, then the excluded databases get cleaned up and removed on the new node.

The `bdr_init_physical` utility replaces the functionality of the `bdr_init_copy` utility from BDR1 and BDR2. It is the BDR3 equivalent of the pglogical `pglogical_create_subscriber` utility.

> **Warning**
> Only one node at the time should join the BDR node group, or be parted from it. If a new node is being joined while there is another join or part operation in progress, the new node will sometimes not have consistent data after the join has finished.

When a new BDR node is joined to an existing BDR group, or a node is subscribed to an upstream peer, the system must copy the existing data from the peer node(s) to the local node before replication can begin. This copy has to be carefully co-ordinated so that the local and remote data starts out *identical*; so

it's not sufficient to just use `pg_dump` yourself. The BDR extension provides built-in facilities for making this initial copy.

During the join process, the BDR extension will synchronize existing data using the provided source node as the basis and creates all metadata information needed for establishing itself in the mesh topology in the BDR group. If the connection between the source and the new node disconnects during this initial copy, the join process will need to be restarted from the beginning.

The node which is joining the cluster must not contain any schema or data which already exists on databases in the BDR group. It's recommended that the newly joining database is empty except for the BDR and pglogical extension. Ensure that all required database users and roles are created.

It's recommended to pick the source node which has the best connection (i.e. is closest) as the source node for joining, since that lowers the time needed for the join to finish.

The join procedure is coordinated using the Raft consensus algorithm, which requires the majority of existing nodes to be online and reachable.

The logical join procedure (which uses `bdr.join_node_group()` function) performs data sync doing `INSERT` operations individually within a single transaction, though this restriction is lifted in later releases. For performance reasons, it is recommended to use `bdr_init_physical` instead.

Note that the join process uses only one node as the source, so can be executed when nodes are down as long as a majority of nodes are available. This can cause a complexity when running logical join: During logical join, the commit timestamp of rows copied from the source node will be set to the latest commit timestamp on the source node. Committed changes on nodes that have a commit timestamp earlier than this because nodes are down or have significant lag could conflict with changes from other nodes; in this case, the newly joined node could be resolved differently to other nodes, causing a divergence. As a result, we recommend not to run a node join when significant replication lag exists between nodes, but if this is necessary then run LiveCompare on the newly joined node to correct any data divergence once all nodes are available and caught up.

Connection DSNs and SSL (TLS)

The DSN of a node is simple a `libpq` connection string, since nodes connect using `libpq`. As such, it can contain any permitted `libpq` connection parameter, including those for SSL. Note that the DSN must work as the connection string from the client connecting to the node in which it is specified. An example of such a set of parameters using a client certificate is:

```
sslmode=verify-full sslcert=bdr_client.crt sslkey=bdr_client.key
sslrootcert=root.crt
```

With this setup, the files `bdr_client.crt`, `bdr_client.key` and `root.crt` must be present in the data directory on each node, with the appropriate permissions. For `verify-full` mode the server's SSL certificate will be checked to ensure that it is directly or indirectly signed with the `root.crt` Certificate Authority, and that the host name or address used in the connection matches the contents of the certificate. In the case of a name, this can match a Subject Alternative Name or, if there are no such names in the certificate, the Subject's Common Name (CN) field. Postgres does not currently support Subject

Alternative Names for IP addresses, so if the connection is made by address rather than name, it must match the CN field.

The CN of the client certificate must be the name of the user making the BDR connection. This is usually the user `postgres`. Each node will require matching lines permitting the connection in the `pg_hba.conf` file, for example:

```
hostssl all         postgres 10.1.2.3/24 cert
hostssl replication postgres 10.1.2.3/24 cert
```

Another setup could be to use `SCRAM-SHA-256` passwords instead of client certificates, and not bother about verifying the server identity as long as the certificate is properly signed. here the DSN parameters might be just:

```
sslmode=verify-ca sslrootcert=root.crt
```

and the corresponding `pg_hba.conf` lines would be like this:

```
hostssl all         postgres 10.1.2.3/24 scram-sha-256
hostssl replication postgres 10.1.2.3/24 scram-sha-256
```

In such a scenario, the postgres user would need a `.pgpass` file containing the correct password.

Witness Nodes

If the cluster has an even number of nodes it may be beneficial to create an extra node to help break ties in the event of a network split (or network partition as it is sometimes called).

Rather than create an additional full-size node, you can create a micro node, sometimes called a Witness node. This is a normal BDR node, just that it is deliberately set up not to replicate any tables or data to it.

Logical Standby Nodes

BDR allows you to create a “logical standby node”, also known as an “offload node”, a “read-only node”, “receive-only node” or “logical read replicas”. A master node can have zero, one or more logical standby nodes.

With a physical standby node the node never comes up fully, forcing it to stay in continual recovery mode. BDR allows something similar. `bdr.join_node_group` has the pause_in_standby option to make the node stay in half-way-joined as a logical standby node. Logical standby nodes receive changes but do not send changes made locally to other nodes.

Later, if desired, use bdr.promote_node() to move the logical standby into a full, normal send/receive node.

A logical standby is sent data by one source node, defined by the DSN in `bdr.join_node_group`. Changes from all other nodes are received from this one source node, minimizing bandwidth between multiple sites.

There are multiple options for high availability:

- If the source node dies, one physical standby can be promoted into a master. In this case, the new master can continue to feed any/all logical standby nodes.
- If the source node dies, one logical standby can be promoted into a full node and replace the source in a failover operation similar to single master operation. Note that if there are multiple logical standby nodes, the other nodes cannot follow the new master, so the effectiveness of this technique is effectively limited to just one logical standby.

Note that in case a new standby is created of an existing BDR node, the necessary replication slots for operation are not synced to the new standby until at least 16MB of LSN has elapsed since the group slot was last advanced. In extreme cases, this may require a full 16MB before slots are synced/created on the streaming replica. In the event a failover or switchover occurs during this interval, the streaming standby cannot be promoted to replace its BDR node, as the group slot and other dependent slots do not exist yet. This is resolved automatically by BDR-EE, but not by BDR-SE.

The slot sync up process on the standby solves this by invoking a function on the upstream. This function moves the group slot in the entire BDR cluster by performing WAL switches and requesting all BDR peer nodes to replay their progress updates. The above causes the group slot to move ahead in a short timespan. This reduces the time required by the standby for the initial slots sync up allowing for faster failover to it, if required.

Logical standby nodes can themselves be protected using physical standby nodes, if desired, so Master->LogicalStandby->PhysicalStandby. Note that you cannot cascade from LogicalStandby to LogicalStandby.

Note that a logical standby does allow write transactions, so the restrictions of a physical standby do not apply. This can be used to great benefit, since it allows the logical standby to have additional indexes, longer retention periods for data, intermediate work tables, LISTEN/NOTIFY, temp tables, materialized views and other differences.

Any changes made locally to logical standbys that commit before the promotion will not be sent to other nodes. All transactions that commit after promotion will be sent onwards. If you perform writes to a logical standby, you are advised to take care to quiesce the database before promotion.

You may make DDL changes to logical standby nodes but they will not be replicated, nor will they attempt to take global DDL locks. BDR functions which act similarly to DDL will also not be replicated. See DDL Replication. If you have made incompatible DDL changes to a logical standby then the database is said to be a divergent node. Promotion of a divergent node will currently result in replication failing. As a result you should plan to either ensure that a logical standby node is kept free of divergent changes if you intend to use as a standby, or ensure that divergent nodes are never promoted.

Physical Standby Nodes

BDR also enables creation of traditional physical standby failover nodes as well. These are commonly intended to directly replace a BDR node within the cluster after a short promotion procedure. As with any standard Postgres cluster, a node may have any number of these physical replicas.

There are however, some minimal prerequisites for this to work properly due to use of replication slots, and other functional requirements in BDR:

- The connection between BDR Primary and Standby uses streaming replication through a physical replication slot.
- The Standby has:
  - `recovery.conf`:
    - `primary_conninfo` pointing to the Primary
    - `primary_slot_name` naming a physical replication slot on the Primary to be used only by this Standby
  - `postgresql.conf`:
    - `shared_preload_libraries = 'pglogical, bdr'` at minimum
    - `hot_standby = on`
    - `hot_standby_feedback = on`
- The Primary has:
  - `postgresql.conf`:
    - `pglogical.standby_slot_names` should specify the physical replication slot used for the Standby's `primary_slot_name`.

While this is enough to produce a working physical standby of a BDR node, there are some additional concerns that should be addressed.

Once established, the Standby requires sufficient time and WAL traffic to trigger an initial copy of the Primary's other BDR-related replication slots, including the BDR group slot. At minimum, slots on a Standby are only "live" and and will survive a failover if they report a non-zero `confirmed_flush_lsn` as reported by `pg_replication_slots`.

As a consequence, physical standby nodes in newly initialized BDR clusters with low amounts of write activity should be checked before assuming a failover will work normally. Failing to take this precaution can result in the Standby having an incomplete subset of required replication slots necessary to function as a BDR node, and thus an aborted failover.

BDR3 Enterprise installations can manually trigger creation of BDR-related replication slots on a physical standby using the following SQL syntax:

```
SELECT bdr.move_group_slot_all_nodes();
```

This will also advance the BDR group slot used to ensure all nodes have reached a minimum LSN across the cluster.

Upon failover, the Standby must perform one of two actions to replace the Primary:

1. Assume control of the same IP address or hostname as the Primary.

2. Inform the BDR cluster of the change in address by executing the bdr.alter_node_interface function on all other BDR nodes.

Once this is done, the other BDR nodes will re-establish communication with the newly promoted Standby -> Primary node. Since replication slots are only synchronized periodically, this new Primary may reflect a lower LSN than expected by the existing BDR nodes. If this is the case, BDR will fast-forward each lagging slot to the last location used by each BDR node.

Take special note of the `pglogical.standby_slot_names` parameter as well. While this is a pglogical configuration parameter, it is important to set in a BDR cluster where there is a Primary -> Physical Standby relationship. While pglogical uses this to ensure physical standby servers always receive WAL traffic before logical replicas, the BDR use case is much different.

BDR maintains a group slot that always reflects the state of the cluster node showing the most lag. With the addition of a physical replica, BDR must be informed that there is a non-participating node member that will, regardless, affect the state of the group slot.

Since the Standby does not directly communicate with the other BDR nodes, the `standby_slot_names` parameter informs BDR to consider named slots as necessary constraints on the group slot as well. When set, the group slot will be held if the Standby shows lag, even if the group slot would have normally been advanced.

As with any physical replica, this type of standby may also be configured as a synchronous replica. As a reminder, this requires:

- On the Standby:
  - Specifying a unique `application_name` in `primary_conninfo`
- On the Primary:
  - Enabling `synchronous_commit`
  - Including the Standby `application_name` in `synchronous_standby_names`

It is possible to mix physical Standby and other BDR nodes in `synchronous_standby_names`.

Node Restart and Down Node Recovery

BDR is designed to recover from node restart or node disconnection. The disconnected node will automatically rejoin the group by reconnecting to each peer node and then replicating any missing data from that node.

When a node starts up, each connection will begin showing `bdr.node_slots.state = catchup` and begin replicating missing data. Catching-up will continue for a period of time that depends upon the amount of missing data from each peer node, which will likely increase over time, depending upon the server workload.

If the amount of write activity on each node is not uniform then you may see that the catchup period from nodes with more data could take significantly longer than other nodes. Eventually, the slot state will change to `bdr.node_slots.state = streaming`.

Nodes that are offline for longer periods of time such as hours or days, can begin to cause resource issues for various reasons. Users should not plan on extended outages without understanding the following issues.

Each node retains change information (using one replication slot for each peer node) so it can later replay changes to a temporarily unreachable node. If a peer node remains offline indefinitely, this accumulated change information will eventually cause the node to run out of storage space for PostgreSQL transaction logs (*WAL* in `pg_wal`), and will likely cause the database server to shut down with an error like:

```
PANIC: could not write to file "pg_wal/xlogtemp.559": No space left on device
```

or report other out-of-disk related symptoms.

In addition, slots for offline nodes also hold back the catalog xmin, preventing vacuuming of catalog tables.

In BDR-EE, offline nodes also hold back freezing of data to prevent losing conflict resolution data. (see: Origin Conflict Detection). BDR-SE users may need to alter their configuration settings as specified.

Administrators should monitor for node outages (see: monitoring) and make sure nodes have sufficient free disk space. If the workload is predictable, it may be possible to calculate how much space is used over time, allowing a prediction of the maximum time a node can be down before critical issues arise.

Replication slots created by BDR must not be removed manually. Should that happen, the cluster is damaged and the node that was using the slot must be parted from the cluster, as described below.

Note that while a node is offline, the other nodes may not yet have received the same set of data from the offline node, so this may appear as a slight divergence across nodes. This imbalance across nodes is corrected automatically during the parting process. Later versions may do this at an earlier time.

Replication Slots created by BDR

On a BDR master node, the following replication slots are automatically created:

- One *group slot*, named `bdr_<database name>_<group name>`;
- N-1 *node slots*, named `bdr_<database name>_<group name>_<node name>`, where N is the total number of BDR nodes in the cluster, including logical standbys, if any.

The user **must not** drop those slots: they have been automatically created by BDR, and therefore must be dropped automatically by BDR itself.

On the other hand, replication slots required by software like Barman or pglogical can be created or dropped, using the appropriate commands for the software, without any effect on BDR.

For example, in a cluster composed by 3 nodes `alpha`, `beta` and `gamma` where BDR is used to replicate the `mydb` database, and the BDR group is called `mygroup`:

- Node `alpha` has three slots:
  - One group slot named `bdr_mydb_mygroup`
  - Two node slots named `bdr_mydb_mygroup_beta` and `bdr_mydb_mygroup_gamma`
- Node `beta` has three slots:
  - One group slot named `bdr_mydb_mygroup`
  - Two node slots named `bdr_mydb_mygroup_alpha` and `bdr_mydb_mygroup_gamma`
- Node `gamma` has three slots:
  - One group slot named `bdr_mydb_mygroup`
  - Two node slots named `bdr_mydb_mygroup_alpha` and `bdr_mydb_mygroup_beta`

Hashing Long Identifiers

Note that the name of a replication slot, like any other PostgreSQL identifier, cannot be longer than 63 bytes; BDR handles this by shortening the database name, the BDR group name and the name of the node, in case the resulting slot name is too long for that limit. The shortening of an identifier is carried out by replacing the final section of the string with a hash of the string itself.

As an example of this, consider a cluster that replicates a database named `db20xxxxxxxxxxxxxxx` (20 bytes long) using a BDR group named `group20xxxxxxxxxxxxx` (20 bytes long); the logical replication slot associated to node `a30xxxxxxxxxxxxxxxxxxxxxxxxxxx` (30 bytes long) will be called

`bdr_db20xxxx3597186_group20xbe9cbd0_a30xxxxxxxxxxxx7f304a2`

since `3597186`, `be9cbd0` and `7f304a2` are respectively the hashes of `db20xxxxxxxxxxxxxxx`, `group20xxxxxxxxxxxxx` and `a30xxxxxxxxxxxxxxxxxxxxxxxxxx`.

Removing a Node From a BDR Group

Since BDR is designed to recover from extended node outages it is necessary to explicitly tell the system if you are removing a node permanently. If you permanently shut down a node and don't tell the other nodes then performance will suffer and eventually the whole system will stop working.

Node removal, also called *parting*, is done using the `bdr.part_node()` function. You must specify the node name (as passed during node creation) to remove a node. The `bdr.part_node()` function can be called from any active node in the BDR group, including the node which is being removed.

Just like the join procedure, parting is done using Raft consensus and requires a majority of nodes to be online to work.

The parting process affects all nodes. The Raft leader will manage a vote between nodes to see which node has the most recent data from the parting node. Then all remaining nodes will make a secondary, temporary connection to the most-recent node allow them to catchup any missing data.

A parted node still is known to BDR, but won't consume resources. A node my well be re-added under the very same name as a parted node. In rare cases, it may be advisable to clear all metadata of a parted node with the function `bdr.drop_node()`.

Uninstalling BDR

Dropping the BDR extension will remove all the BDR objects in a node, including metadata tables. This can be done with the following command:

```
DROP EXTENSION bdr;
```

If the database depends on some BDR-specific objects, then the BDR extension cannot be dropped. Examples include:

- Tables using BDR-specific sequences such as timeshard or galloc
- Column using CRDT data types
- Views that depend on some BDR catalog tables

Those dependencies must be removed before dropping the BDR extension, for instance by dropping the dependent objects, altering the column type to a non-BDR equivalent, or changing the sequence type back to `local`.

> **Warning**
> The BDR extension **must only** be performed if the node has been successfully parted from its BDR node group, or if it is the last node in the group: dropping BDR and pglogical metadata will break replication to/from the other nodes.

> **Warning**
> When dropping a local BDR node, or the BDR extension in the local database, any preexisting session might still try to execute a BDR specific workflow, and therefore fail. The problem can be solved by disconnecting the session and then reconnecting the client, or by restarting the instance.
> Moreover, the "could not open relation with OID (...)" error could occur when (1) parting a node from a BDR cluster, then (2) dropping the BDR extension (3) recreating it, and finally (4) running `pglogical.replication_set_add_all_tables()`. Restarting the instance will solve the problem.

Similar considerations apply to the pglogical extension, which is required by BDR.

If pglogical is only used by BDR, then it is possible to drop both extensions with a single statement:

```
DROP EXTENSION pglogical, bdr;
```

Conversely, if the node is also using pglogical independently of BDR, e.g. for one-way replication of some tables to a remote database, then only the BDR extension should be dropped.

> **Warning**
> Dropping BDR from a database that independently uses pglogical can block an existing pglogical subscription from working further with the "BDR global lock manager not initialized yet" error. Restarting the instance will solve the problem.

Listing BDR Topology

Listing BDR Groups

The following (simple) query lists all the BDR node groups of which the current node is a member: (will currently return one row only)

```
SELECT node_group_name
FROM bdr.local_node_summary;
```

The configuration of each node group can be displayed using a more complex query:

```
SELECT g.node_group_name
, ns.default_repset_name
, node_group_insert_to_update         AS insert_to_update
, node_group_update_to_insert         AS update_to_insert
, node_group_ignore_redundant_updates AS ignore_redundant_updates
, node_group_check_full_tuple         AS check_full_tuple
, node_group_apply_delay              AS apply_delay
, node_group_check_constraints        AS check_constraints
FROM bdr.local_node_summary ns
JOIN bdr.node_group g USING (node_group_name)
```

Listing Nodes in a BDR Group

The list of all nodes in a given node group (e.g. `mygroup`) can be extracted from the `bdr.node_summary` view as in the following example:

```
SELECT node_name         AS name
, node_seq_id            AS ord
, peer_state_name        AS current_state
, peer_target_state_name AS target_state
, interface_connstr      AS dsn
FROM bdr.node_summary
WHERE node_group_name = 'mygroup';
```

Note that the read-only state of a node, as shown in the `current_state` or in the `target_state` query columns, is indicated as `STANDBY`.

List of Node States

- `NONE`: Node state is unset when the worker starts, expected to be set quickly to the current known state.
- `CREATED`: `bdr.create_node()` has been executed, but the node is not a member of any BDR cluster yet.
- `JOIN_START`: `bdr.join_node_group()` begins to join the local node to an existing BDR cluster.

- `JOINING`: The node join started and is currently at the initial sync phase, creating the schema and data on the node.
- `CATCHUP`: Initial sync phase is completed, now the join is at the last step of retrieving and applying transactions that were performed on the upstream peer node since the join started.
- `STANDBY`: Node join finished, but not yet started to broadcast changes. All joins spend some time in this state, but if defined as a Logical Standby the node will continue in this state.
- `PROMOTE`: Node was a logical standby and we just called bdr.promote_node to move the node state to `ACTIVE`. These two `PROMOTE`states have to be coherent to the fact, that only one node can be with a state higher than `STANDBY` but lower than `ACTIVE`.
- `PROMOTING`: Promotion from logical standby to full BDR node is in progress.
- `ACTIVE`: The node is a full BDR node and is currently `ACTIVE`. This is the most common node status.
- `PART_START`: Node was `ACTIVE` or `STANDBY` and we just called bdr.part_node to remove the node from the BDR cluster.
- `PARTING`: Node disconnects from other nodes and plays no further part in consensus or replication.
- `PART_CATCHUP`: Non-parting nodes synchronize any missing data from the recently parted node.
- `PARTED`: Node parting operation is now complete on all nodes.

Only one node at a time can be in either of the states PROMOTE or PROMOTING.

Node Management Interfaces

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

bdr.create_node

Creates a node

Synopsis

```
bdr.create_node(node_name text, local_dsn text)
```

Parameters

- `node_name` - name of the new node; only one node is allowed per database. Valid node names consist of lower case letters, numbers, hyphens and underscores.
- `local_dsn` - connection string to the node

Notes

This function just creates record for the local node with the associated public connection string. There can be only one local record so once it's it will error when trying to run again.

This function is a transactional function - it can be rolled back and the changes made by it are visible to current transaction.

The function will hold lock on the newly created bdr node until end of transaction.

bdr.drop_node

Drops a node. This function is *not intended for regular use* and shold only be executed under the instructions of 2ndQuadrant support.

This function removes the metadata for a given node from the local database. The node can be either:

- The **local** node, in which case all the node metadata is removed, including information about remote nodes;
- A **remote** node, in which case only metadata for that specific node is removed.

Synopsis

```
bdr.drop_node(node_name text, cascade boolean DEFAULT false, force boolean DEFAULT false)
```

Parameters

- `node_name` - Name of an existing node
- `cascade` - Whether to cascade to dependent objects, this will also delete the associated pglogical node. This option should be used with caution!
- `force` - Circumvents all sanity checks and forces the removal of all metadata for the given BDR node despite a possible danger of causing inconsistencies. A forced node drop is to be used by 2ndQuadrant support only in case of emergencies related to parting.

Notes

Before you run this you should already have parted the node using `bdr.part_node()`.

This function is only executed locally without any replication. The node being dropped is locked by this command for other commands that might want to modify it locally.

> **Note**
> BDR3 can have a maximum of 1024 node records (both ACTIVE and PARTED) at one time. This is because each node has a unique sequence number assigned to it, for use by timeshard sequences. PARTED nodes are not automatically cleaned up at the moment; should this become a problem, this function can be used to remove those records.

bdr.create_node_group

Creates a BDR group with the local node as the only member of the group.

Synopsis

```
bdr.create_node_group(node_group_name text,
                      insert_to_update boolean DEFAULT true,
                      update_to_insert boolean DEFAULT false,
                      ignore_redundant_updates boolean DEFAULT false,
                      check_full_tuple boolean DEFAULT false,
                      apply_delay interval DEFAULT INTERVAL '0',
                      check_constraints boolean DEFAULT true)
```

Parameters

- `node_group_name` - Name of the new BDR group; as with the node name, valid group names must consist of lower case letters, numbers and underscores, exclusively.
- `insert_to_update` - Whether an INSERT that conflicts with an existing tuple (due to an INSERT-INSERT conflict) should be converted to an UPDATE, default is yes.
  This option is deprecated and may be disabled or removed in future versions of BDR. Use `bdr.alter_node_set_conflict_resolver` instead.
- `update_to_insert` - Whether an UPDATE to a missing row (e.g. due to a concurrent DELETE) should be converted to an INSERT, default is no. This feature is only available for tables with `REPLICA IDENTITY` set to `FULL`. For other tables, this setting has no effect and the UPDATE to a missing row will be skipped.
  This option is deprecated and may be disabled or removed in future versions of BDR. Use `bdr.alter_node_set_conflict_resolver` instead.
- `ignore_redundant_updates` - Whether UPDATEs that don't actually change any attribute of a tuple can safely be ignored, defaults to no.
- `check_full_tuple` - Whether to use and compare all attributes of the existing tuple with the expected tuple to update, influences conflict detection, applies only to relations with Row Version Tracking enabled, defaults to off.
- `apply_delay` - An interval of time a subscriber waits before applying changes from a provider node. Defaults to 0.
- `check_constraints` - Should the apply process check constraints when writing replicated data. Defaults to true.

Note that all of these parameters, with the exception of the `node_group_name` are simply passed on to pglogical.

Notes

This function will pass request to local consensus worker that is running for the local node.

The function is not transactional. The creation of the group is a background process so once the function has finished the changes cannot be rolled back. Also the changes might not be immediately visible to the current transaction, the `bdr.wait_for_join_completion` can be called to wait until they are.

The group creation does not hold any locks.

bdr.alter_node_group_config

Changes configuration parameter(s) of an existing BDR group. Options with NULL value (default for all of them) will not be modified.

Synopsis

```
bdr.alter_node_group_config(node_group_name text,
                            insert_to_update boolean DEFAULT NULL,
                            update_to_insert boolean DEFAULT NULL,
                            ignore_redundant_updates boolean DEFAULT NULL,
                            check_full_tuple boolean DEFAULT NULL,
                            apply_delay interval DEFAULT NULL,
                            check_constraints boolean DEFAULT NULL)
```

Parameters

- `node_group_name` - Name of an existing BDR group; local node must be part of the group.
- `insert_to_update` - Whether an INSERT that conflicts with an existing tuple should be converted to an UPDATE.
  This option is deprecated and may be disabled or removed in future versions of BDR. Use `bdr.alter_node_set_conflict_resolver` instead.
- `update_to_insert` - Whether an UPDATE to a missing row (e.g. due to a concurrent DELETE) should be converted to an INSERT. See `bdr.create_node_group` for more details.
  This option is deprecated and may be disabled or removed in future versions of BDR. Use `bdr.alter_node_set_conflict_resolver` instead.
- `ignore_redundant_updates` - Whether UPDATEs that don't actually change any attribute of a tuple can safely be ignored.
- `check_full_tuple` - Whether to use and compare all attributes of the existing tuple with the expected tuple to update, influences conflict detection, applies only to relations with `REPLICA IDENTITY FULL`.
- `apply_delay` - An interval of time a subscriber waits before applying changes from a provider node.
- `check_constraints` - Whether the apply process will check the constraints when writing replicated data.

Note that all of the options parameters are simply used to control the pglogical writer.

Notes

This function will pass a request to the group consensus mechanism to change the defaults. The changes made are replicated globally via the consensus mechanism.

The function is not transactional. The request is processed in the background so the function call cannot be rolled back. Also the changes may not be immediately visible to the current transaction.

This function does not hold any locks.

> **Warning**
> When this function is used to change the `apply_delay` value, the change does not apply to nodes that are already members of the group.
> Note that this restriction has little consequence on production usage, because this value is normally not used outside of testing.

bdr.join_node_group

Joins the local node to an already existing BDR group.

Synopsis

```
bdr.join_node_group (
    join_target_dsn text,
    node_group_name text DEFAULT NULL,
    pause_in_standby boolean DEFAULT false,
    wait_for_completion boolean DEFAULT true,
    synchronize_structure text DEFAULT 'all'
)
```

Parameters

- `join_target_dsn` - Specifies the connection string to existing (source) node in the BDR group we wish to add local node to.
- `node_group_name` - Optional name of the BDR group, defaults to NULL which tries to autodetect the group name from information present on the source node.
- `pause_in_standby` - Optionally tells the join process to only join as a logical standby node which can be later promoted to a full member.
- `wait_for_completion` - Wait for the join process to complete before returning, defaults to true.

- `synchronize_structure` - Set what kind of structure (schema) synchronization should be done during the join. Valid options are 'all' which synchronizes, complete database structure and 'none' which will not synchronize any structure, however it will still synchronize data.

If `wait_for_completion` is specified as false, this is an asynchronous call which returns as soon as the joining procedure has started. Progress of the join can be seen in logs and the `bdr.state_journal_details` information view or by calling the `bdr.wait_for_join_completion()` function once `bdr.join_node_group()` returns.

Notes

This function will pass a request to the group consensus mechanism via the node to which the `join_target_dsn` connection string points to. The changes made are replicated globally via the consensus mechanism.

The function is not transactional. The joining process happens in the background and as such cannot be rolled back. The changes are only visible to the local transaction if `wait_for_completion` was set to `true` or by calling `bdr.wait_for_join_completion` later.

Node can only be part of a single group so this function can only be called once on each node.

Node join does not hold any locks in the BDR group.

bdr.promote_node

Promotes a local logical standby node to full member of BDR group.

Synopsis

```
bdr.promote_node(wait_for_completion boolean DEFAULT true)
```

Notes

This function will pass a request to the group consensus mechanism to change the defaults. The changes made are replicated globally via the consensus mechanism.

The function is not transactional. The promotion process happens in the background and as such cannot be rolled back. The changes are only visible to the local transaction if `wait_for_completion` was set to `true` or by calling `bdr.wait_for_join_completion` later.

The promotion process holds lock against other promotions. This lock will not block other `bdr.promote_node` calls, but will prevent the background process of promotion from moving forward on more than one node at a time.

bdr.wait_for_join_completion

Waits for the join procedure of a local node to finish.

Synopsis

```
bdr.wait_for_join_completion(verbose_progress boolean DEFAULT false)
```

Parameters

- `verbose_progress` - Optionally prints information about individual steps taken during the join procedure.

Notes

This function waits until the checks state of the local node reaches the target state which was set by `bdr.create_node_group`, `bdr.join_node_group` or `bdr.promote_node`.

bdr.part_node

Removes ("parts") the node from the BDR group, but does not remove data from the node.

The function can be called from any active node in the BDR group, including the node which is being removed. However, just to make it clear, once the node is PARTED it can not *part* other nodes in the cluster.

> **Note**
> If you are *parting* the local node you must set `wait_for_completion` to false, otherwise it will error.

> **Warning**
> This action is permanent. If you wish to temporarily halt replication to a node, see `bdr.alter_subscription_disable()`.

Synopsis

```
bdr.part_node (
    node_name text,
    wait_for_completion boolean DEFAULT true,
    force boolean DEFAULT false
)
```

Parameters

- `node_name` - Name of an existing node to part.
- `wait_for_completion` - If true, the function will not return until the node is fully parted from the cluster, otherwise the function will just start the parting procedure and returns immediately without waiting. Always set to false when executing on the local node, or when using force.
- `force` - Forces removal of the node on the local node. This will set the node state locally if consensus could not be reached or if the node parting process has stuck.

> **Warning**
> Using `force = true` may leave the BDR group in a inconsistent state and should be only used to recover from byzantine failures where it's impossible to remove the node any other way.**

Notes

This function will pass a request to the group consensus mechanism to part the given node. The changes made are replicated globally via the consensus mechanism. The parting process happens in the background and as such cannot be rolled back. The changes made by the parting process are only visible to the local transaction if `wait_for_completion` was set to `true`.

With `force` set to `true` this function will, on consensus failure, set the state of the given node only on the local node. In such case the function is transactional (because the function itself changes the node state) and can be rolled back. If the function is called on a node which is already in process of parting with `force` set to `true` it will also just mark the given node as parted locally and exit. This is only useful when the consensus cannot be reached on the cluster (majority of the nodes are down) or if the parting process gets stuck for whatever reason. But it's important to take into account that when parting node which was receiving writes, the parting process may take long time without being stuck as the other nodes need to resynchronize any missing data from the given node. The force parting completely skips this resynchronization and as such can leave the other nodes in inconsistent state.

The parting process does not hold any locks.

bdr.alter_node_interface

Changes the connection string (`DSN`) of a specified node.

Synopsis

```
bdr.alter_node_interface(node_name text, interface_dsn text)
```

Parameters

- `node_name` - name of an existing node to alter
- `interface_dsn` - new connection string for a node

Notes

This function is only run on the local node and the changes are only made on the local node. This means that it should normally be executed on every node in BDR group including the node which is being changed.

This function is transactional - it can be rolled back and the changes are visible to the current transaction.

The function holds lock on the local node.

bdr.alter_subscription_enable

Enables either the specified subscription or all the subscriptions of the local BDR node. Also known as resume subscription. No error is thrown if the subscription is already enabled.

Synopsis

```
bdr.alter_subscription_enable(
    subscription_name name DEFAULT NULL,
    immediate boolean DEFAULT false
)
```

Parameters

- `subscription_name` - Name of the subscription to enable; if NULL (the default), all subscriptions on local node will be enabled.
- `immediate` - This currently has no effect.

Notes

This function is not replicated and only affects local node subscriptions (either a specific node or all nodes).

This function is transactional - it can be rolled back and any catalog changes can be seen by the current transaction. The subscription workers will be started by a background process after the transaction has committed.

bdr.alter_subscription_disable

Disables either the specified subscription or all the subscriptions of the local BDR node. Optionally it can immediately stop all the workers associated with the disabled subscriptions as well. Also known as pause subscription. No error is thrown if the subscription is already disabled.

Synopsis

```
bdr.alter_subscription_disable(
    subscription_name name DEFAULT NULL,
    immediate boolean DEFAULT false
)
```

Parameters

- `subscription_name` - Name of the subscription to disable; if NULL (the default), all subscriptions on local node will be disabled.
- `immediate` - Immediate is used to force the action immediately, stopping all the workers associated with the disabled subscription. With this option true, this function cannot be run inside of transaction block.

Notes

This function is not replicated and only affects local node subscriptions (either a specific subscription or all subscriptions).

This function is transactional - it can be rolled back and any catalog changes can be seen by the current transaction. However, the timing of the subscription worker stopping depends on the value of `immediate`; if set to `true`, the workers will be stopped immediately; if set to `false`, they will be stopped at the `COMMIT` time.

> **Note**
> With the parameter `immediate` set to `true`, the stop will however wait for the workers to finish current work.

Node Management Commands

BDR also provides a command line utility for adding nodes to the BDR group via physical copy (`pg_basebackup`) of an existing node and for converting a physical standby of an existing node to a new node in the BDR group.

bdr_init_physical

This is a regular command which is added to the PostgreSQL's bin directory.

The user must specify a data directory. If this data directory is empty, the `pg_basebackup -X stream` command is used to fill the directory using a fast block-level copy operation.

When starting from an empty data directory, if the selective backup option is chosen then only that database will be copied from the source node. The excluded databases will be dropped and cleaned up on the new node.

If the specified data directory is non-empty, this will be used as the base for the new node. If the data directory is already active as a physical standby node, it is required to stop the standby before running `bdr_init_physical`, which will manage Postgres itself. Initially it will wait for catchup and then promote to a master node before joining the BDR group. Note that the `--standby` option, if used, will turn the existing physical standby into a logical standby node; it refers to the end state of the new BDR node, not the starting state of the specified data directory.

This command will drop all pglogical-only subscriptions and configuration from the database and will also drop all PostgreSQL native logical replication subscriptions from the database (or just disable them when the `-S` option is used) as well as any replication origins and slots.

It is the BDR3 version of the `pglogical_create_subscriber` utility.

Note that `bdr_init_physical` requires BDR versions to match between the original or source node and the new node to be initialized.

Synopsis

```
bdr_init_physical [OPTION] ...
```

Options

General Options

- `-D, --pgdata=DIRECTORY` - The data directory to be used for the new node; it can be either empty/non-existing directory, or a directory populated using the `pg_basebackup -X stream` command (required).
- `-l, --log-file=FILE` - Use FILE for logging; default is bdr_init_physical_postgres.log .
- `-n, --node-name=NAME` - The name of the newly created node (required).
- `--replication-sets=SETS` - The name of a comma-separated list of replication set names to use, all replication sets will be used if not specified.
- `--standby` - Create a logical standby (receive only node) rather than full send/receive node.
- `-s, --stop` - Stop the server once the initialization is done.
- `-v` - Increase logging verbosity.
- `-L` - Perform selective pg_basebackup when used in conjunction with an empty/non-existing data directory (-D option).
- `-S` - Instead of dropping logical replication subscriptions, just disable them.

Connection Options

- `-d, --remote-dsn=CONNSTR` - connection string for remote node (required)
- `--local-dsn=CONNSTR` - connection string for local node (required)

Configuration Files Override

- `--hba-conf -path` to the new pg_hba.conf
- `--postgresql-conf` - path to the new postgresql.conf
- `--postgresql-auto-conf` - path to the new postgresql.auto.conf
- `--recovery-conf` - path to the template recovery.conf

Notes

The replication set names specified in the command do not affect the data that exists in the data directory before the node joins the BDR group. This is true whether bdr_init_physical makes its own basebackup or an existing base backup is being promoted to a new BDR node. Thus the `--replication-sets` option only affects the data published and subscribed-to after the node joins the BDR node group. This behaviour is different from the way replication sets are used in a logical join i.e. when using `bdr.join_node_group()`.

Unwanted tables may be truncated by the operator after the join has completed. Refer to the `bdr.tables` catalog to determine replication set membership and identify tables that are not members of any subscribed-to replication set. It's strongly recommended that you truncate the tables rather than drop them, because:

1. DDL replication sets are not necessarily the same as row (DML) replication sets, so you could inadvertently drop the table on other nodes;
2. If you later want to add the table to a replication set and you have dropped it on some subset of nodes, you will need to take care to re-create it only on those nodes without creating DDL conflicts before you can add it to any replicaiton sets.

It's much simpler and safer to truncate your non-replicated tables, leaving them present but empty.

A future version of BDR may automatically omit or remove tables that are not part of the selected replication set(s) for a physical join, so your application should not rely on details of the behaviour documented here.

DDL Replication

DDL stands for “Data Definition Language”: the subset of the SQL language that creates, alters and drops database objects.

For operational convenience and correctness, BDR replicates most DDL actions, with these exceptions:

- Temporary or Unlogged relations
- Certain, mostly long-running DDL statements (see list below)
- Locking commands (LOCK)
- Table Maintenance commands (VACUUM, ANALYZE, CLUSTER)
- Actions of autovacuum
- Operational commands (CHECKPOINT, ALTER SYSTEM)
- Actions related to Databases or Tablespaces

Automatic DDL replication makes it easier to make certain DDL changes without having to manually distribute the DDL change to all nodes and ensure that they are consistent.

In the default replication set, DDL is replicated to all nodes by default. To replicate DDL, a DDL replication filter has to be added to the replication set. See DDL Replication Filtering.

BDR is significantly different to standalone PostgreSQL when it comes to DDL handling, and treating it as the same is the most common operational issue with BDR.

The main difference from table replication is that DDL replication does not replicate the result of the DDL, but the statement itself. This works very well in most cases, though introduces the requirement that the DDL must execute similarly on all nodes. A more subtle point is that the DDL must be immutable with respect to all datatype-specific parameter settings, including any datatypes introduced by extensions (i.e. not built-in). For example, the DDL statement must execute correctly in the default encoding used on each node.

DDL Replication Options

The `bdr.ddl_replication` parameter specifies replication behavior.

`bdr.ddl_replication = on` is the default and will replicate DDL to the default replication set, which by default means all nodes. Non-default replication sets do not replicate DDL, unless they have a DDL filter defined for them.

You can also replicate DDL to specific replication sets using the function `bdr.replicate_ddl_command()`. This can be helpful if you want to run DDL commands when a node is down, or if you want to have indexes or partitions that exist on a subset of nodes or rep sets, e.g. all nodes at site1.

```
SELECT bdr.replicate_ddl_command(
                'CREATE INDEX CONCURRENTLY ON foo (col7);',
                ARRAY['site1'],     -- the replication sets
                'on');              -- ddl_locking to apply
```

It is possible, but not recommended, to skip automatic DDL replication and execute it manually on each node using `bdr.ddl_replication` configuration parameters.

```
SET bdr.ddl_replication = off;
```

When set, it will make BDR skip both the global locking and the replication of executed DDL commands, so you must then run the DDL manually on all nodes.

> **Warning**
> Executing DDL manually on each node without global locking can cause the whole BDR group to stop replicating if conflicting DDL or DML is executed concurrently.

> **Note**
> Some commands like `ALTER TABLE ALTER COLUMN TYPE` use the DML locking to flush the replication queue, in order to ensure that there are no outstanding writes across the cluster. Such commands will do the DML locking for a short duration even with `bdr.global_locking` set to off. In order to disable DML locking for such statements the `bdr.ddl_locking` parameter must be explicitly set to off as well. Please refer to the ALTER TABLE section for more details related to the `ALTER COLUMN TYPE` sub command.

The `bdr.ddl_replication` parameter can only be set by the bdr_superuser or bdr_application roles, or in the config file. For convenience, a function is also provided to change the value within a session:

```
SELECT bdr.set_ddl_replication('off');
```

This allows it to be used sparingly and when essential to keep the cluster operational.

This function can be used to set the mode for the current transaction only, e.g.

```
BEGIN;
SELECT bdr.set_ddl_replication('off', true);
... other commands ...
COMMIT;
```

. . . though this cannot be used with VACUUM or CONCURRENTLY commands since they cannot be executed within a transaction block.

Executing DDL on BDR Systems

A BDR group is not the same as a standalone PostgreSQL server. It is based on asynchronous multi-master replication without central locking and without a transaction co-ordinator. This has important implications when executing DDL.

DDL that executes in parallel will continue to do so with BDR. DDL execution will respect the parameters that affect parallel operation on each node as it executes, so differences in the settings between nodes may be noticeable.

Execution of conflicting DDL needs to be prevented, otherwise DDL replication will end up causing errors and the replication will stop.

BDR offers 3 levels of protection against those problems:

`ddl_locking = 'dml'` is the best option for operations, usable when you execute DDL from only one node at a time. This is not the default, but it is recommended that you use this setting if you can control where DDL is executed from, to ensure that there are no inter-node conflicts. Intra-node conflicts are already handled by PostgreSQL.

`ddl_locking = on` is the strictest option, and is best when DDL might be executed from any node concurrently and you would like to ensure correctness.

`ddl_locking = off` is the least strict option, and is dangerous in general use. This option skips locks altogether and so avoids any performance overhead, making it a useful option when creating a new and empty database schema.

These options can only be set by the bdr_superuser or bdr_application roles, or in the config file. For convenience, a function is also provided to change the value within a session:

```
SELECT bdr.set_ddl_locking('off');
```

It also can be used to set the mode for the current transaction only, with the second parameter `true` (false by DEFAULT) e.g.:

```
BEGIN;
SELECT bdr.set_ddl_locking('off', true);
... other commands ...
COMMIT;
```

When using the `bdr.replicate_ddl_command`, it is possible to set this parameter directly via the third argument, using the specified `bdr.ddl_locking` setting only for the DDL commands passed to that function.

DDL Locking Details

There are two kinds of locks used to enforce correctness of DDL with BDR.

The first kind is known as a Global DDL Lock, and is only used when `ddl_locking = on`. A Global DDL Lock prevents any other DDL from executing on the cluster while each DDL statement runs. This ensures full correctness in the general case, but is clearly too strict for many simple cases. BDR acquires a global lock on DDL operations the first time in a transaction where schema changes are made. This effectively serializes the DDL-executing transactions in the cluster. In other words, while DDL is running, no other connection on any node can run another DDL command, **even if it affects different table(s)**.

To acquire a lock on DDL operations, the BDR node executing DDL contacts the other nodes in a BDR group and asks them to grant it the exclusive right to execute DDL. The lock request is sent via regular replication stream and the nodes respond via replication stream as well. So it's important that nodes (or at least a majority of the nodes) should be running without much replication delay. Otherwise it may take a very long time for the node to acquire the DDL lock. Once the majority of nodes agrees, the DDL execution is carried out.

The ordering of DDL locking is decided using the Raft protocol. DDL statements executed on one node will be executed in the same sequence on all other nodes.

In order to ensure that the node running a DDL has seen effects of all prior DDLs run in the cluster, it waits until it has caught up with the node that had run the previous DDL. If the node running the current DDL is lagging behind in replication with respect to the node that ran the previous DDL, then it make take very long to acquire the lock. Hence it's preferable to run DDLs from a single node or the nodes which have nearly caught up with replication changes originating at other nodes.

The second kind is known as a Relation DML Lock. This kind of lock is used when either `ddl_locking = on` or `ddl_locking = dml`, and the DDL statement might cause in-flight DML statements to fail, e.g. when you change a CHECK constraint on a column. Relation DML locks affect only one relation at a time. Relation DML locks ensure that no DDL executes while there are changes in the queue that might cause replication to halt with an error.

To acquire the global DML lock on a table, the BDR node executing the DDL contacts **all** other nodes in a BDR group, asking them to lock the table against writes, and we wait while all pending changes to that table are drained. Once all nodes are fully caught up, the originator of the DML lock is free to perform schema changes to the table and replicate them to the other nodes.

Waiting for pending DML operations to drain could take a long time, or longer if replication is currently lagging behind. This means that schema changes affecting row representation and constraints, unlike with data changes, can only be performed while all configured nodes are reachable and keeping up reasonably well with the current write rate. If such DDL commands absolutely must be performed while a node is down, the down node must first be removed from the configuration.

Locking behavior is specified by the `bdr.ddl_locking` parameter, as explained in Executing DDL on BDR systems:

- `ddl_locking = on` takes Global DDL Lock and, if needed, takes Relation DML Lock.
- `ddl_locking = dml` skips Global DDL Lock and, if needed, takes Relation DML Lock.
- `ddl_locking = off` skips both Global DDL Lock and Relation DML Lock.

Note also that some BDR functions make DDL changes, so for those functions, DDL locking behavior applies. This will be noted in the docs for each function.

Thus, `ddl_locking = dml` is safe only when we can guarantee that no conflicting DDL will be executed from other nodes, because with this setting, the statements which only require the Global DDL Lock will not use the global locking at all.

`ddl_locking = off` is safe only when the user can guarantee that there are no conflicting DDL and no conflicting DML operations on the database objects we execute DDL on. If you turn locking off and then

experience difficulties, you may lose in-flight changes to data; any issues caused will need to be resolved by the user application team.

In some cases, concurrently executing DDL cannot properly be serialized. Should these serialization failures occur, the DDL may be re-executed.

DDL replication is not active on Logical Standby nodes until they are promoted.

Note that some BDR management functions act like DDL, meaning that they will attempt to take global locks and their actions will be replicated, if DDL replication is active. The full list of replicated functions is listed in BDR Functions that behave like DDL.

Monitoring of global DDL locks and global DML locks is shown in the Monitoring chapter.

Minimizing the Impact of DDL

DDL replicated by BDR will be applied on other nodes. BDR3.6 does not support parallel apply, so while the DDL executes, no other changes can be applied.

To minimize the impact of DDL, transactions performing DDL should be short, should not be combined with lots of row changes, and should avoid long running foreign key or other constraint re-checks. This is good operational advice on any database, but is especially important on BDR. Please use ADD CONSTRAINT NOT VALID, followed by another transaction with VALIDATE CONSTRAINT.

An alternate way of executing long running DDL is to disable DDL replication and then to execute the DDL statement separately on each node.

REINDEX is replicated in versions up to BDR3.6, but not in BDR3.7 or later. Using REINDEX should be avoided because of the AccessExclusiveLocks it holds. Instead, REINDEX CONCURRENTLY should be used (or reindexdb –concurrently).

DDL replication can be disabled when using command line utilities like this:

```
$ export PGOPTIONS="-c bdr.ddl_locking=off -c bdr.ddl_replication=off"
$ reindexdb -S myschema --concurrently
```

Multiple DDL statements might benefit from bunching into a single transaction rather than fired as individual statements, so the DDL lock only has to be taken once. This may not be desirable if the table-level locks interfere with normal operations.

If DDL is holding the system up for too long, it is possible and safe to cancel the DDL on the originating node as you would cancel any other statement, e.g. with `Control-C` in `psql` or with `pg_cancel_backend()`. You cannot cancel a DDL lock from any other node.

It is possible to control how long the global lock will take with (optional) global locking timeout settings. The `bdr.global_lock_timeout` will limit how long the wait for acquiring the global lock can take before it is cancelled; `bdr.global_lock_statement_timeout` limits the runtime length of any statement in transaction that holds global locks, and `bdr.global_lock_idle_timeout` sets the maximum allowed idle time (time between statements) for a transaction holding any global locks. All of these timeouts can be disabled by setting their values to zero.

Once the DDL operation has committed on the originating node, it cannot be canceled or aborted. The BDR group must wait for it to apply successfully on other nodes that confirmed the global lock and for them to acknowledge replay. This is why it is important to keep DDL transactions short and fast.

Handling DDL With Down Nodes

If the node initiating the global DDL lock goes down after it has acquired the global lock (either DDL or DML), the lock stays active. The global locks will not time out, even if timeouts have been set. In case the node comes back up, it will automatically release all the global locks that it holds. If it stays down for a prolonged period time (or forever), remove the node from BDR group in order to release the global locks. This might be one reason for executing emergency DDL using the `bdr.set_ddl_locking` function.

If one of the other nodes goes down after it has confirmed the global lock, but before the command acquiring it has been executed, the execution of that command requesting the lock will continue as if the node was up.

As mentioned in the previous section, the global DDL lock only requires a majority of the nodes to respond, and so it will work if part of the cluster is down, as long as a majority is running and reachable, while the DML lock cannot be acquired unless the whole cluster is available.

If we have the global DDL or global DML lock and another node goes down, the command will continue normally and the lock will be released.

Statement Specific DDL Replication Concerns

Not all commands can be replicated automatically. Such commands are generally disallowed, unless DDL replication is turned off by turning `bdr.ddl_replication` off.

BDR prevents some DDL statements from running when it is active on a database. This protects the consistency of the system by disallowing statements that cannot be replicated correctly, or for which replication is not yet supported. Statements that are supported with some restrictions are covered in DDL Statements With Restrictions; while commands that are entirely disallowed in BDR are covered in prohibited DDL statements.

If a statement is not permitted under BDR, it is often possible to find another way to do the same thing. For example, you can't do an `ALTER TABLE` which adds column with a volatile default value, but it is generally possible to rephrase that as a series of independent `ALTER TABLE` and `UPDATE` statements that will work.

Generally unsupported statements are prevented from being executed, raising a `feature_not_supported` (SQLSTATE `0A000`) error.

DDL Statements Requiring a DML Lock

For non-temporary, non-unlogged tables, these DDL command types require a global DML lock on that table only:

- some variants of `ALTER TABLE` (see ALTER TABLE Locking below for details)
- `DROP TABLE`
- `CREATE UNIQUE INDEX` (non-unique index creations do NOT need a DML lock)
- `DROP INDEX` (both unique and non-unique indexes require a DML lock)
- `CREATE POLICY`
- `CREATE RULE`
- `ALTER SEQUENCE`
- `DROP SEQUENCE`

Actions on indexes, views, materialized views and foreign tables never require a global DML lock.

Actions on objects created in the current transaction never require a global DML lock.

ALTER TABLE Locking

The following variants of `ALTER TABLE` will only take DDL lock and **not** a DML lock. This applies to the `ALTER MATERIALIZED VIEW` variant, too:

- `ALTER TABLE ... ADD COLUMN ... (immutable) DEFAULT`
- `ALTER TABLE ... ALTER COLUMN ... SET DEFAULT expression`
- `ALTER TABLE ... ALTER COLUMN ... DROP DEFAULT`
- `ALTER TABLE ... ALTER COLUMN ... TYPE` if it does not require rewrite
- `ALTER TABLE ... ALTER COLUMN ... SET STATISTICS`
- `ALTER TABLE ... VALIDATE CONSTRAINT`
- `ALTER TABLE ... ATTACH PARTITION`
- `ALTER TABLE ... DETACH PARTITION`
- `ALTER TABLE ... ENABLE TRIGGER` (`ENABLE REPLICA TRIGGER` will still take a DML lock)
- `ALTER TABLE ... CLUSTER ON`
- `ALTER TABLE ... SET WITHOUT CLUSTER`
- `ALTER TABLE ... SET ( storage_parameter = value [, ... ] )`
- `ALTER TABLE ... RESET ( storage_parameter = value [, ... ] )`
- `ALTER TABLE ... OWNER TO`

All other variants of `ALTER TABLE` take a DML lock on the table being modified. Some variants of `ALTER TABLE` have restrictions, noted below.

Non Replicated DDL Statements

REFRESH MATERIALIZED VIEW

`REFRESH MATERIALIZED VIEW` will only refresh the view on the node where it was executed. As a result, the contents of Materialized Views will likely differ between nodes and should never be assumed to be the same.

LOCK TABLE

`LOCK TABLE` is only executed locally and is not replicated. Normal replication happens after transaction commit, so `LOCK TABLE` would not have any effect on other nodes.

For globally locking table, users can request a global DML lock explicitly by calling `bdr.global_lock_table()`.

DDL on Databases and Tablespaces

DDL commands on Databases and Tablespaces can be executed while BDR is active on a database. The action will execute normally on the local node, but the DDL is not replicated to other nodes. This is because they are global objects and not limited to a single database.

The following commands are restricted in this way:

- CREATE DATABASE
- ALTER DATABASE
- DROP DATABASE
- COMMENT ON DATABASE
- SECURITY LABEL ON DATABASE
- CREATE TABLESPACE
- ALTER TABLESPACE
- DROP TABLESPACE
- COMMENT ON TABLESPACE
- SECURITY LABEL ON TABLESPACE

DDL Statements With Restrictions

The following statements or statement options are not currently permitted when BDR is active on a database. If used, they will fail with an ERROR; workarounds are listed in the next section.

CREATE TABLE

Generally, `CREATE TABLE` is allowed. There are a few options/subcommands that are not supported.

The unsupported commands are:

- `CONSTRAINT ... EXCLUDE` - Not supported due to required internode locking.
- `AS SELECT` - Not supported unless creating a `TEMPORARY` table. This restriction may be lifted in later versions. (Also applies to SELECT INTO.)
- `WITH OIDS` - This is an outdated option not recommended by PostgreSQL itself; in BDR the Oids of rows would be different on different nodes, making it even less useful than normal.
- `CONSTRAINT ... EXCLUDE` - Not supported due to required internode locking.

CREATE MATERIALIZED VIEW

Not supported. This restriction may be lifted in later versions.

ALTER TABLE

Generally, `ALTER TABLE` commands are allowed. There are, however, several sub-commands that are not supported:

- `ADD COLUMN ... DEFAULT (non-immutable expression)` - This is not allowed because it would currently result in different data on different nodes. See Adding a Column for a suggested workaround.
- `ADD CONSTRAINT ... EXCLUDE` - Exclusion constraints are not supported for now. Exclusion constraints do not make much sense in an asynchronous system and lead to changes that cannot be replayed.
- `ALTER TABLE ... SET WITH[OUT] OIDS` - Is not supported for the same reasons as in `CREATE TABLE`.
- `ALTER COLUMN ... SET STORAGE external` - Will be rejected if the column is one of the columns of the replica identity for the table.
- `ALTER COLUMN ... TYPE` - Changing a column's type is not supported if the command causes the whole table to be rewritten, which occurs when the change is not binary coercible. Note that binary coercible changes may only be allowed one way. For example, the change from VARCHAR(128) to VARCHAR(256) is binary coercible and therefore allowed, whereas the change VARCHAR(256) to VARCHAR(128) is not binary coercible and therefore normally disallowed. For non-replicated `ALTER COLUMN ... TYPE` it can be allowed if the column is automatically castable to the new type (it does not contain the `USING` clause). See below for an example. Table rewrites would hold an AccessExclusiveLock for extended periods on larger tables, so such commands are likely to be infeasible on highly available databases in any case. See Changing a Column's Type for a suggested workarounds.

The following example fails because it tries to add a constant value of type `timestamp` onto a column of type `timestamptz`. The cast between `timestamp` and `timestamptz` relies upon the time zone of the session and so is not immutable.

```
ALTER TABLE foo
  ADD expiry_date timestamptz DEFAULT timestamp '2100-01-01 00:00:00' NOT NULL;
```

This next example works because the type change is binary coercible and so does not cause a table rewrite, so it will execute as a catalog-only change.

```
CREATE TABLE foo (id BIGINT PRIMARY KEY, description VARCHAR(20));
ALTER TABLE foo ALTER COLUMN description TYPE VARCHAR(128);
```

However, making this change to reverse the above command is not possible because the change from VARCHAR(128) to VARCHAR(20) is not binary coercible.

```
ALTER TABLE foo ALTER COLUMN description TYPE VARCHAR(20);
```

See later for suggested workarounds.

It is useful to provide context for different types of ALTER TABLE ... ALTER COLUMN TYPE (ATCT) operations that are possible in general and in non-replicated environments.

Some ATCT operations only update the metadata of the underlying column type and do not require a rewrite of the underlying table data. This is typically the case when the existing column type and the target type are binary coercible. For example:

```
CREATE TABLE sample (col1 BIGINT PRIMARY KEY, col2 VARCHAR(128), col3 INT);
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(256);
```

It will also be OK to change the column type to `VARCHAR` or `TEXT` datatypes because of binary coercibility. Again, this is just a metadata update of the underlying column type.

```
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR;
ALTER TABLE sample ALTER COLUMN col2 TYPE TEXT;
```

However, if you want to reduce the size of col2, then that will lead to a rewrite of the underlying table data. Rewrite of a table is normally restricted.

```
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(64);
ERROR:  ALTER TABLE ... ALTER COLUMN TYPE that rewrites table data may not affect replicated
```

To give an example with non-text types, consider col3 above with type INTEGER. An ATCT operation which tries to convert to SMALLINT or BIGINT will fail in a similar manner as above.

```
ALTER TABLE sample ALTER COLUMN col3 TYPE bigint;
ERROR:  ALTER TABLE ... ALTER COLUMN TYPE that rewrites table data may not affect replicated
```

In both the above failing cases, there exists an automatic assignment cast from the current types to the target types. However there is no binary coercibility, which ends up causing a rewrite of the underlying table data.

In such cases, in controlled DBA environments, it is possible to change the type of a column to an automatically castable one, by adopting a rolling upgrade for the type of this column in a non-replicated environment on all the nodes, one by one. If the DDL is not replicated and the change of the column type is to an automatically castable one as above, then it is possible to allow the rewrite locally on the node performing the alter, along with concurrent activity on other nodes on this same table. This non-replicated ATCT operation can then be repeated on all the nodes one by one to bring about the desired change of the column type across the entire BDR cluster. Note that because this involves a rewrite, the activity will still take the DML lock for a brief period, and thus requires that the whole cluster is available. With the above specifics in place, the rolling upgrade of the non-replicated alter activity can be carried out as below:

```
-- foreach node in BDR cluster do:
SET bdr.ddl_replication TO FALSE;
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(64);
ALTER TABLE sample ALTER COLUMN col3 TYPE BIGINT;
RESET bdr.ddl_replication;
-- done
```

Due to automatic assignment casts being available for many data types, this local non-replicated ATCT operation supports a wide variety of conversions. Also note that ATCT operations that use a `USING` clause are likely to fail because of the lack of automatic assignment casts. A few common conversions with automatic assignment casts are mentioned below.

```
-- foreach node in BDR cluster do:
SET bdr.ddl_replication TO FALSE;
ATCT operations to-from {INTEGER, SMALLINT, BIGINT}
ATCT operations to-from {CHAR(n), VARCHAR(n), VARCHAR, TEXT}
ATCT operations from numeric types to text types
RESET bdr.ddl_replication;
-- done
```

The above is not an exhaustive list of possibly allowable ATCT operations in a non-replicated environment. Obviously, not all ATCT operations will work. The cases where no automatic assignment is possible will fail even if we disable DDL replication. So, while conversion from numeric types to text types works in non-replicated environment, conversion back from text type to numeric types will fail.

```
SET bdr.ddl_replication TO FALSE;
-- conversion from BIGINT to TEXT works
ALTER TABLE sample ALTER COLUMN col3 TYPE TEXT;
-- conversion from TEXT back to BIGINT fails
ALTER TABLE sample ALTER COLUMN col3 TYPE BIGINT;
ERROR:  ALTER TABLE ... ALTER COLUMN TYPE which cannot be automatically cast to new type may r
RESET bdr.ddl_replication;
```

While the ATCT operations in non-replicated environments support a variety of type conversions, it is important to note that the rewrite can still fail if the underlying table data contains values that cannot be assigned to the new data type. For example, the current type for a column might be `VARCHAR(256)` and we tried a non-replicated ATCT operation to convert it into `VARCHAR(128)`. If there is any existing data in the table which is wider than 128 bytes, then the rewrite operation will fail locally.

```
INSERT INTO sample VALUES (1, repeat('a', 200), 10);
SET bdr.ddl_replication TO FALSE;
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(128);
INFO:  in rewrite
ERROR:  value too long for type character varying(128)
```

If underlying table data meets the characteristics of the new type, then the rewrite will succeed. However, there is a possibility that replication will fail if other nodes (which have not yet performed the non-replicated rolling data type upgrade) introduce new data that is wider than 128 bytes concurrently to this local ATCT operation. This will bring replication to a halt in the cluster. So it is important to be aware of the data type restrictions and characteristics at the database and application levels while performing these non-replicated rolling data type upgrade operations. It is **strongly** recommended and advisable to perform and test such ATCT operations in controlled and fully-aware DBA environments. We need to be aware that these ATCT operations are asymmetric, and backing out certain changes that fail could lead to table rewrites lasting long durations.

Also note that the above implicit castable ALTER activity cannot be performed in transaction blocks.

CREATE SEQUENCE

Generally `CREATE SEQUENCE` is supported, but when using distributed sequences, some options have no effect.

ALTER SEQUENCE

Generally `ALTER SEQUENCE` is supported, but when using distributed sequences, some options have no effect.

Role manipulation statements

Users are global objects in a PostgreSQL instance, which means they span multiple databases while BDR operates on an individual database level. This means that role manipulation statement handling needs extra thought.

BDR requires that any roles that are referenced by any replicated DDL must exist on all nodes. The roles are not required to have the same grants, password, etc., but they must exist.

BDR will replicate role manipulation statements if `bdr.role_replication` is enabled (default) *and role manipulation statements are run in a BDR-enabled database*.

The role manipulation statements include the following statements:

- CREATE ROLE
- ALTER ROLE
- DROP ROLE
- GRANT ROLE
- CREATE USER
- ALTER USER
- DROP USER
- CREATE GROUP
- ALTER GROUP
- DROP GROUP

In general, either:

- The system should be configured with `bdr.role_replication = off` and all role (user and group) changes should be deployed by external orchestration tools like Ansible, Puppet, Chef, etc., or explicitly replicated via `bdr.replicate_ddl_command(...)`; or
- The system should be configured so that exactly one BDR-enabled database on the PostgreSQL instance has `bdr.role_replication = on` and all role management DDL should be run on that database.

It is strongly recommended that you run all role management commands within one database.

If role replication is turned off, then the administrator must ensure that any roles used by DDL on one node also exist on the other nodes, or BDR apply will stall with an `ERROR` until the role is created on the other node(s).

Note: BDR will *not* capture and replicate role management statements when they are run on a non-BDR-enabled database within a BDR-enabled PostgreSQL instance. For example if you have DBs 'bdrdb' (bdr group member) and 'postgres' (bare db), and `bdr.role_replication = on`, then a `CREATE USER` run in `bdrdb` will be replicated, but a `CREATE USER` run in `postgres` will not.

ALTER TYPE

Users should note that `ALTER TYPE` is replicated but a Global DML lock is *not* applied to all tables that use that data type, since PostgreSQL does not record those dependencies. See workarounds, below.

Restricted DDL Workarounds

As noted in DDL Statements With Restrictions, BDR limits some kinds of DDL operations, in particular some variants of the `ALTER TABLE` command that manipulate the row representation in the table.

It is often possible to split up this operation into smaller changes.

Adding a Column

To add a column with a volatile default, run these commands in separate transactions:

```
ALTER TABLE mytable ADD COLUMN newcolumn coltype; -- Note the lack of DEFAULT or NOT NULL

ALTER TABLE mytable ALTER COLUMN newcolumn DEFAULT volatile-expression;

BEGIN;
SELECT bdr.global_lock_table('mytable');
UPDATE mytable SET newcolumn = default-expression;
COMMIT;
```

This splits schema changes and row changes into separate transactions that can be executed by BDR and result in consistent data across all nodes in a BDR group.

For best results, batch the update into chunks so that you do not update more than a few tens or hundreds of thousands of rows at once. This can be done using a `PROCEDURE` with embedded transactions.

It is important that the last batch of changes runs in a transaction that takes a global DML lock on the table, otherwise it is possible to miss rows that are inserted concurrently into the table on other nodes.

If required, `ALTER TABLE mytable ALTER COLUMN newcolumn NOT NULL;` can be run after the `UPDATE` has finished.

Changing a Column's Type

PostgreSQL causes a table rewrite in some cases where it could be avoided, for example:

```
CREATE TABLE foo (id BIGINT PRIMARY KEY, description VARCHAR(128));
ALTER TABLE foo ALTER COLUMN description TYPE VARCHAR(20);
```

This statement can be rewritten to avoid a table rewrite by making the restriction a table constraint rather than a datatype change, which can then be validated in a subsequent command to avoid long locks, if desired.

```
CREATE TABLE foo (id BIGINT PRIMARY KEY, description VARCHAR(128));
ALTER TABLE foo
  ALTER COLUMN description TYPE varchar,
  ADD CONSTRAINT description_length_limit CHECK (length(description) <= 20) NOT VALID;
ALTER TABLE foo VALIDATE CONSTRAINT description_length_limit;
```

Should the validation fail, then it is possible to UPDATE just the failing rows. This technique can be used for TEXT and VARCHAR using `length()`, or with NUMERIC datatype using `scale()`.

In case a table rewrite cannot be avoided, the following approach serves as a general solution:

- extend the table with a new column of the required type and with a temporary name
- ensure newly inserted or updated rows get the new column populated by using either a default (if possible) or a trigger to copy over data from the old column
- rewrite all pre-existing rows to copy over data from the old column, possibly in multiple small transactions
- wait for all nodes to have applied the table rewrite transaction(s)
- re-add required indices and constraints, as required
- in a single transaction, drop the old column and rename the new one in its place

More details and a specific example of how this approach works is given in Appendix E.

CREATE TABLE AS SELECT

Instead of `CREATE TABLE AS SELECT`, you can achieve the same effect using:

```
CREATE TABLE mytable;
INSERT INTO mytable SELECT ... ;
```

CREATE MATERIALIZED VIEW

Instead of `CREATE MATERIALIZED VIEW`, you can achieve the same effect using:

```
CREATE TABLE mytable;
INSERT INTO mytable SELECT ... ;
```

Changing Other Types

The `ALTER TYPE` statement is replicated, but affected tables are not locked.

When this DDL is used, the user should ensure that the statement has successfully executed on all nodes before using the new type. This can be achieved using the `bdr.wait_slot_confirm_lsn()` function.

For example,

```
ALTER TYPE contact_method ADD VALUE 'email';
SELECT bdr.wait_slot_confirm_lsn(NULL, NULL);
```

will ensure that the DDL has been written to all nodes before using the new value in DML statements.

BDR Functions that behave like DDL

The following BDR management functions act like DDL. This means that they will attempt to take global locks and their actions will be replicated, if DDL replication is active and DDL filter settings allow that. For detailed information, see the documentation for the individual functions.

Replication Set Management

- bdr.create_replication_set
- bdr.alter_replication_set
- bdr.drop_replication_set
- bdr.replication_set_add_table
- bdr.replication_set_remove_table
- bdr.replication_set_add_ddl_filter
- bdr.replication_set_remove_ddl_filter

Conflict Management

- bdr.alter_table_conflict_detection

Sequence Management

- bdr.alter_sequence_set_kind

Stream Triggers

- bdr.create_conflict_trigger
- bdr.create_transform_trigger
- bdr.drop_trigger

Security and Roles

The BDR3 extension can be created only by superusers, although if desired, it is possible to set up the `pgextwlist` extension and configure it to allow BDR3 to be created by a non-superuser.

Configuring and managing BDR3 does not require superuser access, nor is that recommended. The privileges required by BDR3 are split across the following default roles, named similarly to the PostgreSQL default roles:

- *bdr_superuser* - the highest-privileged role, having access to all BDR tables and functions.
- *bdr_read_all_stats* - the role having read-only access to the tables, views and functions, sufficient to understand the state of BDR.
- *bdr_monitor* - at the moment the same as `bdr_read_all_stats`, to be extended later.
- *bdr_application* - the minimal privileges by applications running BDR.

These BDR default roles are created when the BDR3 extension is installed. See BDR Default Roles below for more details.

Granting privileges on catalog objects

Administrators should not grant explicit privileges on catalog objects such as tables, views and functions; manage access to those objects by granting one of the roles documented in BDR Default Roles.

This requirement is a consequence of the flexibility that allows joining a node group even if the nodes on either side of the join do not have the exact same version of BDR (and therefore of the BDR catalog).

More precisely, if privileges on individual catalog objects have been explicitly granted, then the `bdr.join_node_group()` procedure could fail because the corresponding GRANT statements extracted from the node being joined might not apply to the node that is joining.

Role Management

Users are global objects in a PostgreSQL instance. `CREATE USER` and `CREATE ROLE` commands are replicated automatically if they are executed in the database where BDR is running and the `bdr.role_replication` is turned on. However, if these commands are executed in other databases in the same PostgreSQL instance then they will not be replicated, even if those users have rights on the BDR database.

When a new BDR node joins the BDR group, existing users are not automatically copied unless the node is added using `bdr_init_physical`. This is intentional and is an important security feature. PostgreSQL allows users to access multiple databases, with the default being to access any database. BDR does not know which users access which database and so cannot safely decide which users to copy across to the new node.

PostgreSQL allows you to dump all users with the command:

```
pg_dumpall --roles-only > roles.sql
```

The file `roles.sql` can then be edited to remove unwanted users before re-executing that on the newly created node. Other mechanisms are possible, depending on your identity and access management solution (IAM), but are not automated at this time.

Roles and Replication

DDL changes executed by a user are applied as that same user on each node.

DML changes to tables are replicated as the table-owning user on the target node. It is recommended - but not enforced - that a table is owned by the same user on each node.

If table A is owned by user X on node1 and owned by user Y on node2, then if user Y has higher privileges than user X, this could be viewed as a privilege escalation. Since some nodes have different use cases, we allow this but warn against it to allow the security administrator to plan and audit this situation.

On tables with row level security policies enabled, changes will be replicated without re-enforcing policies on apply. This is equivalent to the changes being applied as `NO FORCE ROW LEVEL SECURITY`, even if `FORCE ROW LEVEL SECURITY` is specified. If this is not desirable, specify a row_filter that avoids replicating all rows. It is recommended - but not enforced - that the row security policies on all nodes be identical or at least compatible.

Note that bdr_superuser controls replication for BDR and may add/remove any table from any replication set. bdr_superuser does not need, nor is it recommended to have, any privileges over individual tables. If the need exists to restrict access to replication set functions, restricted versions of these functions can be implemented as `SECURITY DEFINER` functions and `GRANT`ed to the appropriate users.

Connection Role

When allocating a new BDR node, the user supplied in the DSN for the `local_dsn` argument of `bdr.create_node` and the `join_target_dsn` of `bdr.join_node_group` are used frequently to refer to, create, and manage database objects. This is especially relevant during the initial bootstrapping process, where the specified accounts may invoke operations on database objects directly or through the pglogical module rather than BDR.

BDR is carefully written to prevent privilege escalation attacks even when using a role with `SUPERUSER` rights in these DSNs.

To further reduce the attack surface, a more restricted user may be specified in the above DSNs. At a minimum, such a user must be granted permissions on all nodes, such that following stipulations are satisfied:

- the user has the `REPLICATION` attribute
- it is granted the `CREATE` permission on the database
- it inherits the `pglogical_superuser` and `bdr_superuser` roles

- it owns all database objects to replicate, either directly or via permissions from the owner role(s).

(Only in combination with Postgres 10, an additional explicit `USAGE` permission on schema `pglogical` is required on the joining node.)

Once all nodes are joined, the permissions may be further reduced to just the following to still allow DML and DDL replication:

- The user has the `REPLICATION` attribute.
- It inherits the `pglogical_superuser` and `bdr_superuser` roles.

Triggers

In PostgreSQL, triggers may be created by both the owner of a table and anyone who has been granted the TRIGGER privilege. Triggers granted by the non-table owner would execute as the table owner in BDR, which could cause a security issue.

BDR mitigates this problem by using stricter rules on who can create a trigger on a table:

- superuser
- bdr_superuser
- Owner of the table can create triggers according to same rules as in PostgreSQL (must have EXECUTE privilege on function used by the trigger).
- Users who have TRIGGER privilege on the table can only create a trigger if they create the trigger using a function that is owned by the same owner as the table and they satisfy standard PostgreSQL rules (again must have EXECUTE privilege on the function). So if both table and function have same owner and the owner decided to give a user both TRIGGER privilege on the table and EXECUTE privilege on the function, it is assumed that it is okay for that user to create a trigger on that table using this function.
- Users who have TRIGGER privilege on the table can create triggers using functions that are defined with the SECURITY DEFINER clause if they have EXECUTE privilege on them. This clause makes the function always execute in the context of the owner of the function itself both in standard PostgreSQL and BDR.

The above logic is built on the fact that in PostgreSQL, the owner of the trigger is not the user who created it but the owner of the function used by that trigger.

The same rules apply to existing tables, and if the existing table has triggers which are not owned by the owner of the table and do not use SECURITY DEFINER functions, it will not be possible to add it to a replication set.

These checks were added with BDR 3.6.19. An application that relies on the behavior of previous versions can set `bdr.backwards_compatibility` to 30618 (or lower) to behave like earlier versions.

BDR replication apply uses the system-level default search_path only. Replica triggers, stream triggers and index expression functions may assume other search_path settings which will then fail when they

execute on apply. To ensure this does not occur, resolve object references clearly using either the default search_path only (always use fully qualified references to objects, e.g. schema.objectname), or set the search path for a function using ALTER FUNCTION ... SET search_path = ... for the functions affected.

Catalog Tables

System catalog and Information Schema tables are always excluded from replication by BDR.

In addition, tables owned by extensions are excluded from replication.

BDR Functions & Operators

All BDR functions are exposed in the `bdr` schema. Any calls to these functions should be schema qualified, rather than putting `bdr` in the `search_path`.

All BDR operators are available via `pg_catalog` schema to allow users to exclude the `public` schema from the search_path without problems.

BDR Default Roles

BDR default roles are created when the BDR3 extension is installed. Note that after BDR3 extension is dropped from a database, the roles continue to exist and need to be dropped manually if required. This allows BDR to be used in multiple databases on the same PostgreSQL instance without problem.

Remember that the `GRANT ROLE` DDL statement does not participate in BDR replication, thus you should execute this on each node of a cluster.

bdr_superuser

- ALL PRIVILEGES ON ALL TABLES IN SCHEMA BDR
- ALL PRIVILEGES ON ALL ROUTINES IN SCHEMA BDR

bdr_read_all_stats

- SELECT ON
- `bdr.apply_log_summary`
- `bdr.ddl_epoch`
- `bdr.ddl_replication`
- `bdr.global_consensus_journal_details`
- `bdr.global_lock`

- bdr.global_locks
- bdr.local_consensus_state
- bdr.local_node_summary
- bdr.node
- bdr.node_catchup_info
- bdr.node_conflict_resolvers
- bdr.node_group
- bdr.node_local_info
- bdr.node_peer_progress
- bdr.node_slots
- bdr.node_summary
- bdr.replication_sets
- bdr.sequences
- bdr.state_journal_details
- bdr.stat_relation
- bdr.stat_subscription
- bdr.subscription
- bdr.subscription_summary
- bdr.tables
- bdr.worker_errors
- EXECUTE ON
- bdr.bdr_edition
- bdr.bdr_version
- bdr.bdr_version_num
- bdr.conflict_resolution_to_string
- bdr.conflict_type_to_string
- bdr.decode_message_payload

- bdr.get_global_locks
- bdr.get_raft_status
- bdr.get_relation_stats
- bdr.get_slot_flush_timestamp
- bdr.get_sub_progress_timestamp
- bdr.get_subscription_stats
- bdr.peer_state_name
- bdr.show_subscription_status

bdr_monitor

Same as bdr_read_all_stats

bdr_application

- EXECUTE ON
- All functions for column_timestamps datatypes
- All functions for CRDT datatypes
- bdr.alter_sequence_set_kind
- bdr.create_conflict_trigger
- bdr.create_transform_trigger
- bdr.drop_trigger
- bdr.global_lock_table
- bdr.is_camo_partner_connected
- bdr.logical_transaction_status
- bdr.ri_fkey_trigger
- bdr.seq_nextval
- bdr.set_ddl_locking
- bdr.set_ddl_replication
- bdr.trigger_get_committs
- bdr.trigger_get_conflict_type
- bdr.trigger_get_row
- bdr.trigger_get_type
- bdr.trigger_get_xid
- bdr.wait_slot_confirm_lsn

Verification

BDR has been verified using the following tools and approaches.

Coverity

Coverity Scan has been used to verify the BDR stack providing coverage against vulnerabilities using the following rules and coding standards:

- MISRA C
- ISO 26262
- ISO/IEC TS 17961
- OWASP Top 10
- CERT C
- CWE Top 25
- AUTOSAR

CIS Benchmark

CIS PostgreSQL Benchmark v1, 19 Dec 2019 has been used to verify the BDR stack. Using the `cis_policy.yml` configuration available as an option with TPAexec gives the following results for the Scored tests:

	Result	Description
1.4	PASS	Ensure systemd Service Files Are Enabled
1.5	PASS	Ensure Data Cluster Initialized Successfully
2.1	PASS	Ensure the file permissions mask is correct
2.2	PASS	Ensure the PostgreSQL pg_wheel group membership is correct
3.1.2	PASS	Ensure the log destinations are set correctly
3.1.3	PASS	Ensure the logging collector is enabled
3.1.4	PASS	Ensure the log file destination directory is set correctly
3.1.5	PASS	Ensure the filename pattern for log files is set correctly

	Result	Description
3.1.6	PASS	Ensure the log file permissions are set correctly
3.1.7	PASS	Ensure 'log_truncate_on_rotation' is enabled
3.1.8	PASS	Ensure the maximum log file lifetime is set correctly
3.1.9	PASS	Ensure the maximum log file size is set correctly
3.1.10	PASS	Ensure the correct syslog facility is selected
3.1.11	PASS	Ensure the program name for PostgreSQL syslog messages is correct
3.1.14	PASS	Ensure 'debug_print_parse' is disabled
3.1.15	PASS	Ensure 'debug_print_rewritten' is disabled
3.1.16	PASS	Ensure 'debug_print_plan' is disabled
3.1.17	PASS	Ensure 'debug_pretty_print' is enabled
3.1.18	PASS	Ensure 'log_connections' is enabled
3.1.19	PASS	Ensure 'log_disconnections' is enabled
3.1.21	PASS	Ensure 'log_hostname' is set correctly
3.1.23	PASS	Ensure 'log_statement' is set correctly
3.1.24	PASS	Ensure 'log_timezone' is set correctly

	Result	Description
3.2	PASS	Ensure the PostgreSQL Audit Extension (pgAudit) is enabled
4.1	PASS	Ensure sudo is configured correctly
4.2	PASS	Ensure excessive administrative privileges are revoked
4.3	PASS	Ensure excessive function privileges are revoked
4.4	PASS	Tested Ensure excessive DML privileges are revoked
5.2	Not Tested	Ensure login via 'host' TCP/IP Socket is configured correctly
6.2	PASS	Ensure 'backend' runtime parameters are configured correctly
6.7	Not Tested	Ensure FIPS 140-2 OpenSSL Cryptography Is Used
6.8	PASS	Ensure SSL is enabled and configured correctly
7.3	PASS	Ensure WAL archiving is configured and functional

Note that test 5.2 can PASS if audited manually, but does not have an automatable test.

Test 6.7 succeeds on default deployments using CentOS, but it requires extra packages on Debian variants.

Conflicts

BDR is an active/active or multi-master DBMS. If used asynchronously, writes to the same or related row(s) from multiple different nodes can result in data conflicts when using standard data types.

Conflicts aren't ERRORs, they are events that can be detected and resolved automatically as they occur by BDR, in most cases. Resolution depends upon the nature of the application and the meaning of the data, so it is important that BDR provides the application a range of choices as to how to resolve conflicts.

By default conflicts are resolved at row level. That is, when changes from two nodes conflict, we pick either the local or remote tuple and discard the other one. For example we may compare commit timestamps for the two conflicting changes and keep the newer one. This ensures all nodes converge to the same result, and establishes commit-order-like semantics on the whole cluster.

Conflict handling is fully configurable, as described later in this chapter. Conflicts can be detected and handled differently for each table using conflict triggers, available with BDR-EE , described in the Stream Triggers chapter .

Column-level conflict detection and resolution is available with BDR-EE , described in the CLCD chapter .

Conflict-free data types (CRDTs) are also available with BDR-EE , described in the CRDT chapter . However it's still important to understand information in this chapter even when using CRDTs exclusively.

If you wish to avoid conflicts, you can use Eager replication or in certain workloads CAMO, both additional features in BDR-EE.

This chapter covers conflicts with standard data types in detail.

Some clustering systems use distributed lock mechanisms to prevent concurrent access to data. These can perform reasonably when servers are very close but cannot support geographically distributed applications where very low latency is critical for acceptable performance.

Distributed locking is essentially a pessimistic approach, whereas BDR advocates an optimistic approach: avoid conflicts where possible but allow some types of conflict to occur and resolve them when they arise.

How conflicts happen

Inter-node conflicts arise as a result of sequences of events that could not happen if all the involved transactions happened concurrently on the same node. Because the nodes only exchange changes after transactions commit, each transaction is individually valid on the node it committed on but would not be valid if applied on another node that has done other conflicting work at the same time. Since BDR replication essentially replays the transaction on the other nodes, the replay operation can fail if there is a conflict between a transaction being applied and a transaction that was committed on the receiving node.

The reason most conflicts can't happen when all transactions run on a single node is that PostgreSQL has inter-transaction communication mechanisms to prevent it - `UNIQUE` indexes, `SEQUENCE`s, row and relation locking, `SERIALIZABLE` dependency tracking, etc. All of these mechanisms are ways to communicate between ongoing transactions to prevent undesirable concurrency issues.

BDR does not have a distributed transaction manager or lock manager. That's part of why it performs well with latency and network partitions. As a result, *transactions on different nodes execute entirely independently from each other*, when using the default, lazy replication. Less independence between nodes can avoid conflicts altogether, which is why BDR also offers eager replication for when this is important.

Types of conflict

PRIMARY KEY or UNIQUE Conflicts

The most common conflicts are row conflicts where two operations affect a row with the same key in ways they could not do on a single node. BDR can detect most of those and will apply the update_if_newer conflict resolver.

Row conflicts include:

- `INSERT` vs `INSERT`
- `UPDATE` vs `UPDATE`
- `UPDATE` vs `DELETE`
- `INSERT` vs `UPDATE`
- `INSERT` vs `DELETE`
- `DELETE` vs `DELETE`

The view `bdr.node_conflict_resolvers` provides information on how conflict resolution is currently configured for all known conflict types.

INSERT/INSERT Conflicts

The most common conflict, `INSERT`/`INSERT`, arises where `INSERT`s on two different nodes create a tuple with the same `PRIMARY KEY` values (or the same values for a single `UNIQUE` constraint if no `PRIMARY KEY` exists). BDR handles this by retaining the most recently inserted tuple of the two, according to the originating host's timestamps, unless a user-defined conflict handler overrides this.

This conflict will generate the `insert_exists` conflict type, which is by default resolved by choosing the newer (based on commit time) row and keeping only that one (`update_if_newer` resolver). Other resolvers can be configured - see Conflict Resolution for details.

To resolve this conflict type in the Enterprise Edition, you can also use column-level conflict resolution and user-defined conflict triggers.

This type of conflict can be effectively eliminated by use of Global Sequences.

INSERTs that Violate Multiple UNIQUE Constraints

An INSERT/INSERT conflict can violate more than one UNIQUE constraint (of which one might be the PRIMARY KEY). If a new row violates more than one UNIQUE constraint and that results in a conflict against more than one other row then the apply of the replication change will result in ERROR.

BDR can only handle an INSERT/INSERT conflict on multiple constraints as long as the violations are all produced by the same other row.

In case of such a conflict, you must manually remove the conflicting tuple(s) from the local side by DELETEing it or by UPDATEing it so that it no longer conflicts with the new remote tuple. There may be more than one conflicting tuple. There is not currently any built-in facility to ignore, discard or merge tuples that conflict with more than one local row.

UPDATE/UPDATE Conflicts

Where two concurrent UPDATEs on different nodes change the same tuple (but not its PRIMARY KEY), an UPDATE/UPDATE conflict can occur on replay.

These can generate different conflict kinds based on configuration and situation. If the table is configured with Row Version Conflict Detection then the original (key) row is compared with the local row; if they are different, the update_differing conflict is generated. When using Origin Conflict Detection, the origin of the row is checked (origin is node which the current local row came from), if that has changed, the update_origin_change conflict is generated. In all other cases, the UPDATE is normally applied without conflict being generated.

Both of these conflicts are resolved same way as insert_exists as described above.

UPDATE Conflicts on the PRIMARY KEY

BDR cannot currently perform conflict resolution where the PRIMARY KEY is changed by an UPDATE operation. It is permissible to update the primary key, but you must ensure that no conflict with existing values is possible.

Conflicts on update of the primary key are Divergent Conflicts and require manual operator intervention.

Updating a PK is possible in PostgreSQL, but there are issues in both PostgreSQL and BDR.

Let's create a very simple example schema to explain:

```
CREATE TABLE pktest (pk integer primary key, val integer);
INSERT INTO pktest VALUES (1,1);
```

Updating the Primary Key column is possible, so this SQL succeeds:

```
UPDATE pktest SET pk=2 WHERE pk=1;
```

. . . but if we have multiple rows in the table, e.g.:

```
INSERT INTO pktest VALUES (3,3);
```

. . . then some UPDATEs would succeed:

```
UPDATE pktest SET pk=4 WHERE pk=3;

SELECT * FROM pktest;
 pk | val
----+-----
  2 |   1
  4 |   3
(2 rows)
```

. . . but other UPDATEs fail with constraint errors:

```
UPDATE pktest SET pk=4 WHERE pk=2;
ERROR:  duplicate key value violates unique constraint "pktest_pkey"
DETAIL:  Key (pk)=(4) already exists
```

So PostgreSQL applications that UPDATE PKs need to be very careful to avoid runtime errors, even without BDR.

With BDR, the situation becomes more complex if UPDATEs are allowed from multiple locations at same time.

Executing these two changes concurrently works:

```
node1: UPDATE pktest SET pk=pk+1 WHERE pk = 2;
node2: UPDATE pktest SET pk=pk+1 WHERE pk = 4;

SELECT * FROM pktest;
 pk | val
----+-----
  3 |   1
  5 |   3
(2 rows)
```

. . . but executing these next two changes concurrently will cause a divergent error, since both changes are accepted. But when the changes are applied on the other node it will result in update_missing conflicts.

```
node1: UPDATE pktest SET pk=1 WHERE pk = 3;
node2: UPDATE pktest SET pk=2 WHERE pk = 3;
```

. . . leaving the data different on each node:

```
node1:
SELECT * FROM pktest;
 pk | val
----+-----
  1 |   1
  5 |   3
(2 rows)
```

```
node2:
SELECT * FROM pktest;
 pk | val
----+-----
  2 |   1
  5 |   3
(2 rows)
```

This situation can be identified and resolved using LiveCompare.

Concurrent conflicts give problems. Executing these two changes concurrently is not easily resolvable:

```
node1: UPDATE pktest SET pk=6, val=8 WHERE pk = 5;
node2: UPDATE pktest SET pk=6, val=9 WHERE pk = 5;
```

Both changes are applied locally, causing a divergence between the nodes. But then apply on the target fails on both nodes with a duplicate key value violation ERROR, which causes replication to halt and currently requires manual resolution.

This duplicate key violation error can now be avoided and replication will not break, if you set the conflict_type `update_pkey_exists` to `skip`, `update` or `update_if_newer`. This may still lead to divergence depending on the nature of the update.

You can avoid divergence in cases like the one described above where the same old key is being updated by the same new key concurrently by setting `update_pkey_exists` to `update_if_newer`.

As a result, we recommend strongly against allowing PK UPDATEs in your applications, especially with BDR. If there are parts of your application that change Primary Keys then those changes should be made using Eager replication to avoid concurrent changes.

UPDATEs that Violate Multiple UNIQUE Constraints

Like INSERTs that Violate Multiple UNIQUE Constraints, where an incoming `UPDATE` violates more than one `UNIQUE` index (and/or the `PRIMARY KEY`), BDR cannot apply the standard conflict resolution if more than one conflicting row is identified.

These are Divergent Conflicts and will therefore require manual operator intervention.

BDR supports deferred unique constraints. If a transaction can commit on the source then it will apply cleanly on target, unless it sees conflicts. However, a deferred Primary Key cannot be used as a REPLICA IDENTITY, so the use cases are already limited by that and the warning about using multiple unique constraints, above.

UPDATE/DELETE Conflicts

It is possible for one node to `UPDATE` a row that another node simultaneously `DELETE`s. In this case an `UPDATE`/`DELETE` conflict can occur on replay.

If the DELETEd row is still detectable (the deleted row wasn't removed by VACUUM), the update_recently_deleted conflict will be generated. By default the UPDATE will just be skipped, but the resolution for this can be configured; see Conflict Resolution for details.

The deleted row can be cleaned up from the database by the time the UPDATE is receiver in case the local node is lagging behind in replication. In this case BDR cannot differentiate between UPDATE/DELETE conflicts and INSERT/UPDATE Conflicts and will simply generate the update_missing conflict. This conflict is by default resolved by skipping the UPDATE.

Another type of conflicting DELETE and UPDATE is a DELETE operation that comes after the row was UPDATEd locally. In this situation, the outcome depends upon the type of conflict detection used. When using the default, Origin Conflict Detection, no conflict is detected at all, leading to the DELETE being applied and the row removed. If you enable Row Version Conflict Detection, a delete_recently_updated conflict is generated. The default resolution for this conflict type is to to apply the DELETE and remove the row, but this can be configured or handled via a conflict trigger.

INSERT/UPDATE Conflicts

When using the default asynchronous mode of operation, a node may receive an UPDATE of a row before the original INSERT was received. This can only happen with 3 or more nodes being active (see Conflicts with 3 or more nodes below).

When this happens the update_missing conflict is generated. This conflict is by default resolved by skipping the UPDATE. Other resolvers like insert_or_skip or insert_or_error try to INSERT new row based on data from UPDATE when possible (when the whole row was received) For the reconstruction of the row to be possible, the table either needs to have REPLICA IDENTITY FULL or the row must not contain any TOASTed data.

See TOAST Support Details for more info about TOASTed data.

INSERT/DELETE Conflicts

Similarly to the INSERT/UPDATE conflict, the node may also receive a DELETE operation on a row for which it didn't receive an INSERT yet. This is again only possible with 3 or more nodes set up (see Conflicts with 3 or more nodes below).

BDR cannot currently detect this conflict type: the INSERT operation will not generate any conflict type and the INSERT will be applied.

The DELETE operation will always generate a delete_missing conflict which is by default resolved by skipping the operation.

DELETE/DELETE Conflicts

A DELETE/DELETE conflict arises where two different nodes concurrently delete the same tuple.

This will always generate a delete_missing conflict which is by default resolved by skipping the operation.

This conflict is harmless since both DELETEs have the same effect, so one of them can be safely ignored.

Conflicts with 3 or more nodes

If one node `INSERT`s a row which is then replayed to a 2nd node and `UPDATE`d there, a 3rd node can receive the `UPDATE` from the 2nd node before it receives the `INSERT` from the 1st node. This is an `INSERT`/`UPDATE` conflict.

These conflicts are handled by discarding the `UPDATE`. This can lead to *different data on different nodes*, i.e. these are Divergent Conflicts.

Note that this conflict type can only happen with 3 or more masters, of which at least 2 must be actively writing.

Also, the replication lag from node 1 to node 3 must be high enough to allow the following sequence of actions:

1. node 2 receives INSERT from node 1
2. node 2 performs UPDATE
3. node 3 receives UPDATE from node 2
4. node 3 receives INSERT from node 1

Using `insert_or_error` or in some cases the `insert_or_skip` conflict resolver for the `update_missing` conflict type is a viable mitigation strategy for these conflicts. Note however that enabling this option opens the door for `INSERT`/`DELETE` conflicts; see below.

1. node 1 performs UPDATE
2. node 2 performs DELETE
3. node 3 receives DELETE from node 2
4. node 3 receives UPDATE from node 1, turning it into an INSERT

If these are problems it's recommended to tune freezing settings for a table or database so that they are correctly detected as `update_recently_deleted`. This is done automatically in BDR Enterprise Edition.

Another alternative is to use Eager Replication to prevent these conflicts.

INSERT/DELETE conflicts can also occur with 3 or more nodes. Such a conflict is identical to `INSERT`/`UPDATE`, except with the `UPDATE` replaced by a `DELETE`. This can result in a delete_missing conflict.

BDR could choose to make each INSERT into a check-for-recently deleted, as occurs with an update_missing conflict. However, the cost of doing this penalizes majority of users, so at this time we simply log delete_missing.

Later releases will automatically resolve INSERT/DELETE anomalies via re-checks using LiveCompare when delete_missing conflicts occur. These can be performed manually by applications by checking conflict logs or conflict log tables; see later.

These conflicts can occur in two main problem use cases:

- INSERT, followed rapidly by a DELETE - as can be used in queuing applications
- Any case where the PK identifier of a table is re-used

Neither of these cases is common and we recommend not replicating the affected tables if these problem use cases occur.

BDR has problems with the latter case because BDR relies upon the uniqueness of identifiers to make replication work correctly.

Applications that insert, delete and then later re-use the same unique identifiers can cause difficulties. This is known as the ABA Problem. BDR has no way of knowing whether the rows are the current row, the last row or much older rows. https://en.wikipedia.org/wiki/ABA_problem

Unique identifier reuse is also a business problem, since it is prevents unique identification over time, which prevents auditing, traceability and sensible data quality. Applications should not need to reuse unique identifiers.

Any identifier reuse that occurs within the time interval it takes for changes to pass across the system will cause difficulty. Although that time may be short in normal operation, down nodes may extend that interval to hours or days.

We recommend that applications do not reuse unique identifiers; but if they do, take steps to avoid reuse within a period of less than a year.

Any application that uses Sequences or UUIDs will not suffer from this problem.

Foreign Key Constraint Conflicts

Conflicts between a remote transaction being applied and existing local data can also occur for `FOREIGN KEY` constraints (FKs).

BDR applies changes with `session_replication_role = 'replica'` so foreign keys are **not** re-checked when applying changes. In an active/active environment this can result in FK violations if deletes occur to the referenced table at the same time as inserts into the referencing table. This is similar to an INSERT/DELETE conflict.

First, we will explain the problem and then provide solutions.

In single-master PostgreSQL, any INSERT/UPDATE that refers to a value in the referenced table will have to wait for DELETEs to finish before they can gain a row-level lock. If a DELETE removes a referenced value then the INSERT/UPDATE will fail the FK check.

In multi-master BDR there are no inter-node row-level locks. So an INSERT on the referencing table does not wait behind a DELETE on the referenced table, so both actions can occur concurrently. Thus an INSERT/UPDATE on one node on the referencing table can utilize a value at the same time that as a DELETE on the referenced table on another node. This then results in a value in the referencing table that is no longer present in the referenced table.

In practice, this only occurs if DELETEs occur on referenced tables in separate transactions from DELETEs on referencing tables. This is not a common operation.

In a parent-child relationship, e.g. Orders -> OrderItems it isn't typical to do this, more likely to mark an OrderItem as cancelled than to remove it completely. For reference/lookup data it would be strange to completely remove entries at the same time as using those same values for new fact data.

While there is a possibility of dangling FKs, the risk of this in general is very low and so BDR does not impose a generic solution to cover this case. Once users understand the situation in which this occurs, two solutions are possible:

The first solution is to restrict the use of FKs to closely related entities that are generally modified from only one node at a time, are infrequently modified, or where the modification's concurrency is application-mediated. This simply avoids any FK violations at the application level.

The second solution is to add triggers to protect against this case using the BDR-provided functions `bdr.ri_fkey_trigger()` and `bdr.ri_fkey_on_del_trigger()`. When called as `BEFORE` triggers, these functions will use `FOREIGN KEY` information to avoid FK anomalies by setting referencing columns to NULL, much as if we had a SET NULL constraint. Note that this re-checks ALL FKs in one trigger, so you only need to add one trigger per table to prevent FK violation.

As an example, we have two tables: Fact and RefData. Fact has an FK that references RefData. Fact is the referencing table and RefData is the referenced table. One trigger needs to be added to each table.

Add a trigger that will set columns to NULL in Fact if the referenced row in RefData has already been deleted.

```
CREATE TRIGGER bdr_replica_fk_iu_trg
    BEFORE INSERT OR UPDATE ON fact
    FOR EACH ROW
    EXECUTE PROCEDURE bdr.ri_fkey_trigger();

ALTER TABLE fact
    ENABLE REPLICA TRIGGER bdr_replica_fk_iu_trg;
```

Add a trigger that will set columns to NULL in Fact at the time a DELETE occurs on the RefData table.

```
CREATE TRIGGER bdr_replica_fk_d_trg
    BEFORE DELETE ON refdata
    FOR EACH ROW
    EXECUTE PROCEDURE bdr.ri_fkey_on_del_trigger();

ALTER TABLE refdata
    ENABLE REPLICA TRIGGER bdr_replica_fk_d_trg;
```

Adding both triggers will avoid dangling foreign keys.

TRUNCATE Conflicts

TRUNCATE behaves similarly to a DELETE of all rows, but performs this action by physical removal of the table data, rather than row-by-row deletion. As a result, row-level conflict handling is not available, so

TRUNCATE commands do not generate conflicts with other DML actions, even when there is a clear conflict.

As a result the ordering of replay could cause divergent changes if another DML is executed close in time to TRUNCATE statements.

Users may wish to take one of the following options:

- Ensure TRUNCATE is not executed alongside other concurrent DML and rely on LiveCompare to highlight any such inconsistency.
- Replace TRUNCATE with a DELETE statement with no WHERE clause, noting that this is likely to have very poor performance on larger tables.
- Set the `bdr.truncate_locking` to `true` which changes TRUNCATE behavior to be controlled by `bdr.ddl_locking` like other statements which can conflict with DML operations. This will be the default behavior on BDR 3.7 and newer. Note that this requires all nodes to be up and running in order to execute the TRUNCATE command.
- Explicitly request a global DML lock using the function bdr.global_lock_table() prior to executing the TRUNCATE command to prevent any race conditions that could result in divergent changes, as shown below. Note that this requires all nodes to be up and running in order to execute the TRUNCATE command.

```
BEGIN;
SELECT bdr.global_lock_table('foo');
TRUNCATE foo;
COMMIT;
```

Exclusion Constraint Conflicts

BDR doesn't support exclusion constraints, and prevents their creation.

If an existing stand-alone database is converted to a BDR database then all exclusion constraints should be manually dropped.

In a distributed asynchronous system it is not possible to ensure that no set of rows that violate the constraint exists, because all transactions on different nodes are fully isolated. Exclusion constraints would lead to replay deadlocks where replay could not progress from any node to any other node because of exclusion constraint violations.

If you force BDR to create an exclusion constraint, or you don't drop existing ones when converting a standalone database to BDR, you should expect replication to break. You can get it to progress again by removing or altering the local tuple(s) that an incoming remote tuple conflicts with so that the remote transaction can be applied.

Data Conflicts for Roles and Tablespace differences

Conflicts can also arise where nodes have global (PostgreSQL-system-wide) data, like roles, that differs. This can result in operations - mainly `DDL` - that can be run successfully and committed on one node, but then fail to apply to other nodes.

For example, `node1` might have a user named `fred`, but that user was not created on `node2`. If `fred` on `node1` creates a table, it will be replicated with its owner set to `fred`. When the DDL command is applied to `node2`, the DDL will fail because there is no user named `fred`. This failure will emit an `ERROR` in the PostgreSQL logs.

Administrator intervention is required to resolve this conflict by creating the user `fred` in the database where BDR is running. You may wish to set bdr.role_replication = on to resolve this in future.

Lock Conflicts and Deadlock Aborts

Because BDR writer processes operate much like normal user sessions, they are subject to the usual rules around row and table locking. This can sometimes lead to BDR writer processes waiting on locks held by user transactions, or even by each other.

Relevant locking includes:

- explicit table-level locking (`LOCK TABLE ...`) by user sessions
- explicit row-level locking (`SELECT ... FOR UPDATE/FOR SHARE`) by user sessions
- implicit locking because of row `UPDATE`s, `INSERT`s or `DELETE`s, either from local activity or from replication from other nodes

It is even possible for a BDR writer process to deadlock with a user transaction, where the user transaction is waiting on a lock held by the writer process and vice versa. Two writer processes may also deadlock with each other. PostgreSQL's deadlock detector will step in and terminate one of the problem transactions. If the BDR writer process is terminated it will simply retry and generally succeed.

All these issues are transient and generally require no administrator action. If a writer process is stuck for a long time behind a lock on an idle user session, the administrator may choose to terminate the user session to get replication flowing again, but this is no different to a user holding a long lock that impacts another user session.

Use of the log_lock_waits facility in PostgreSQL can help identify locking related replay stalls.

Divergent Conflicts

Divergent conflicts arise when data that should be the same on different nodes differs unexpectedly. Divergent conflicts should not occur, but not all such conflicts can be reliably prevented at the time of writing.

Changing the `PRIMARY KEY` of a row can lead to a divergent conflict if another node changes the key of the same row before all nodes have replayed the change. Avoid changing primary keys, or change them only on one designated node.

Divergent conflicts involving row data generally require administrator action to manually adjust the data on one of the nodes to be consistent with the other one. Such conflicts should not arise so long as BDR is used as documented and settings or functions marked as unsafe are avoided.

The administrator must manually resolve such conflicts. Use of the advanced options such as `bdr.ddl_replication` and `bdr.ddl_locking` may be required depending on the nature of the conflict. However, careless use of these options can make things much worse and it isn't possible to give general instructions for resolving all possible kinds of conflict.

TOAST Support Details

PostgreSQL uses out of line storage for larger columns called TOAST.

The TOAST values handling in logical decoding (which BDR is built on top of) and logical replication is different from in-line data stored as part of the main row in the table.

TOAST value will be logged into the transaction log (WAL) only if the value has changed. This can cause problems especially when handling UPDATE conflicts because the UPDATE statement which didn't change a value of a toasted column will produce a row without that column. As mentioned in the INSERT/UPDATE Conflicts BDR will produce an error if `update_missing` conflict is resolved using `insert_or_error` and there are missing TOAST columns.

However there are more subtle issues than the above one in case of concurrent workloads with asynchronous replication (eager transactions are not affected). Imagine for example following workload on a BDR cluster with 3 nodes called A,B and C:

1. on node A: txn A1 does an UPDATE SET col1 = 'toast data...' and commits first
2. on node B: txn B1 does UPDATE SET other_column = 'anything else'; and commits after A1
3. on node C: the connection to node A lags behind
4. on node C: txn B1 is applied first, it misses the TOASTed column in col1, but gets applied without conflict
5. on node C: txn A1 will conflict (on update_origin_change) and get skipped
6. node C will miss the toasted data from A1 forever

The above is not usually a problem when using BDR (it would be when using either built-in logical replication or plain pglogical for multi-master) because BDR adds its own logging of TOAST columns when it detects a local UPDATE to a row which recently replicated a TOAST column modification and the local UPDATE is not modifying the TOAST. Thus BDR will prevent any inconsistency for TOASTed data across different nodes, at the price of increased WAL logging when updates occur on multiple nodes (i.e. when origin changes for a tuple). Additional WAL overhead will be zero if all updates made from a single node, as is normally the case with BDR AlwaysOn architecture.

> **Note**
> Running `VACUUM FULL` or `CLUSTER` on just the TOAST table without also doing same on the main table will remove metadata needed for the extra logging to work, which means that for a short period of time after such statement the protection against these concurrency issues will not be present.

For the `insert_or_error` conflict resolution, the use of `REPLICA IDENTITY FULL` is however still required.

None of these problems associated with TOASTed columns affect tables with `REPLICA IDENTITY FULL` as this setting will always log a TOASTed values as part of the key since the whole row is considered to be part of the key. Both BDR and pglogical are smart enough to reconstruct the new row, filling the missing data from the key row. Be aware that as a result, the use of `REPLICA IDENTITY FULL` can increase WAL size significantly.

Avoiding or Tolerating Conflicts

In most cases the application can be designed to avoid conflicts, or to tolerate them.

Conflicts can only happen if there are things happening at the same time on multiple nodes, so the simplest way to avoid conflicts is to only ever write to one node, or to only ever write to a specific row in a specific way from one specific node at a time. This happens naturally in many applications, for example, many consumer applications only allow data to be changed by the owning user, e.g. changing the default billing address on your account, so data changes seldom experience update conflicts.

It might happen that you make a change just before a node goes down, so the change appears to have been lost. You might then make the same change again, leading to two updates via different nodes. When the down node comes back up it will try to send the older change to other nodes, but it will be rejected because the last update of the data is kept.

For `INSERT`/`INSERT` conflicts, use of Global Sequences can completely prevent this type of conflict.

For applications that assign relationships between objects, e.g. a room booking application, applying update_if_newer may not give an acceptable business outcome, i.e. it isn't useful to confirm to two people separately that they have booked the same room. The simplest resolution is to use Eager replication to ensure that only one booking succeeds. More complex ways might be possible, depending upon the application, e.g. assign 100 seats to each node and allow those to be booked by a writer on that node, but if none are available locally, use a distributed locking scheme or Eager replication once most seats have been reserved.

Another technique for ensuring certain types of update only occur from one specific node would be to route different types of transaction through different nodes:

- e.g. receiving parcels on one node, but delivering parcels via another node.
- e.g. a service application where orders are input on one node, work is prepared on a second node and then served back to customers on another.

The best course of action is frequently to allow conflicts to occur and design the application to work with BDR's conflict resolution mechanisms to cope with the conflict.

Conflict Detection

BDR provides three mechanisms for conflict detection:

- Origin Conflict Detection (default)
- Row Version Conflict Detection
- Column-Level Conflict Detection

Origin Conflict Detection

(Previously known as Timestamp Conflict Detection, but this was confusing).

Origin conflict detection uses and relies on commit timestamps as recorded on the host where the transaction originates from. This requires clocks to be in sync to work correctly, or to be within a tolerance of the fastest message between two nodes. If this is not the case, conflict resolution will tend to favour the node that is further ahead. Clock skew between nodes can be managed using the parameters `bdr.maximum_clock_skew` and `bdr.maximum_clock_skew_action`.

Row origins are only available if track_commit_timestamps = on.

Conflicts are initially detected based upon whether the replication origin has changed or not, so conflict triggers will be called in situations that may turn out not to be actual conflicts. Hence, this mechanism is not precise since it can generate false positive conflicts.

Origin info is available only up to the point where a row is frozen. Updates arriving for a row after it has been frozen will not raise a conflict, so will be applied in all cases. This is the normal case when we add a new node by bdr_init_physical, so raising conflicts would cause many false positive cases in that case.

When a node has been offline for some time reconnects and begins sending data changes this could potentially cause divergent errors if the newly arrived updates are actually older than the frozen rows that they update. Inserts and Deletes are not affected by this situation.

Users are advised to not leave down nodes for extended outages, as discussed in Node Restart and Down Node Recovery.

To handle this situation gracefully, BDR-EE will automatically hold back the freezing of rows while a node is down. No changes to parameter settings are required. .

Row Version Conflict Detection

Alternatively, BDR provides the option to use row versioning and make conflict detection independent of the nodes' system clock.

Row version conflict detection requires 3 things to be enabled. If any of these steps are not performed correctly then Origin Conflict Detection will be used.

1. `check_full_tuple` must be enabled for the BDR node group
2. `REPLICA IDENTITY FULL` must be enabled on all tables that intended to use row version conflict detection.
3. Row Version Tracking must be enabled on the table by using `bdr.alter_table_conflict_detection`. This function will add a new column (with user defined name) and an `UPDATE` trigger which manages the new column value. The column will be created as `INTEGER` type.

Although the counter is incremented only on UPDATE, detection of conflicts for both UPDATE and DELETE is possible using this technique.

This approach resembles Lamport timestamps and fully prevents the ABA problem for conflict detection.

bdr.alter_table_conflict_detection

This function changes how the conflict detection works for a given table.

Synopsis

```
bdr.alter_table_conflict_detection(relation regclass,
                                   method text,
                                   column_name name DEFAULT NULL)
```

Parameters

- `relation` - name of the relation for which to set new conflict detection method
- `method` - which conflict detection method to use
- `column_name` - which column to use for storing of the column detection data, can be skipped, in which case column name will be automatically chosen based on the conflict detection method, the `row_origin` method does not require extra column for metadata storage

The recognized methods for conflict detection are:

- `row_origin` - origin of the previous change made on the tuple (see Origin Conflict Detection above), this is the only method supported which does not require extra column in the table
- `row_version` - row version column (see Row Version Conflict Detection above)
- `column_commit_timestamp` - per-column commit timestamps (described in the CLCD chapter)
- `column_modify_timestamp` - per-column modification timestamp (described in the CLCD chapter)

Notes

For more information about the difference between `column_commit_timestamp` and `column_modify_timestamp` conflict detection methods, see Current vs Commit Timestamp section in the CLCD chapter.

This function uses the same replication mechanism as `DDL` statements. This means the replication is affected by the ddl filters configuration.

The function will take a `DML` global lock on the relation for which column-level conflict resolution is being enabled.

This function is transactional - the effects can be rolled back with the `ROLLBACK` of the transaction and the changes are visible to the current transaction.

The `bdr.alter_table_conflict_detection` function can be only executed by the owner of the `relation`, unless `bdr.backwards_compatibility` is set is set to 30618 or below.

> **Warning**
> Please note that when changing the conflict detection method from one that uses extra column for storage of metadata, that column will be dropped.

> **Warning**
> This function automatically disables CAMO (together with a warning, as long as these are not disabled with `bdr.camo_enable_client_warnings`).

List of Conflict Types

BDR recognizes the following conflict types, which can be used as the `conflict_type` parameter:

- `insert_exists` - an incoming insert conflicts with an existing row via a primary key or an unique key/index (formerly called `insert_insert`)
- `update_differing` - an incoming update's key row is differing from a local row, this can only happen when using Row Version Conflict Detection. (formerly called `update_update`)
- `update_origin_change` - an incoming update is modifying a row which was last changed by a different node (also formerly called `update_update`)
- `update_missing` - an incoming update is trying to modify a row which does not exist (formerly called `update_delete`)
- `update_recently_deleted` - an incoming update is trying to modify a row which was recently deleted
- `update_pkey_exists` - an incoming update has modified the `PRIMARY KEY` to a value which already exists on the node which is applying the change
- `delete_recently_updated` - an incoming delete with an older commit timestamp than the most recent update of the row on the current node, or when using Row Version Conflict Detection. (formerly called `update_update`)

- `delete_missing` - an incoming delete is trying to remove a row which does not exist (formerly called `delete_delete`)
- `target_column_missing` - the target table is missing one or more columns present in the incoming row
- `source_column_missing` - the incoming row is missing one or more columns that are present in the target table
- `target_table_missing` - target table is missing

Conflict Resolution

Most conflicts can be resolved automatically. BDR defaults to a last-update-wins mechanism, or more accurately the update_if_newer conflict resolver. This mechanism will retain the most recently inserted or changed row of the two conflicting ones based on the same commit timestamps used for conflict detection. The behavior in certain corner case scenarios depends on the settings used for bdr.create_node_group and alternatively for bdr.alter_node_group_config.

BDR lets the user override the default behavior of conflict resolution via the following function:

bdr.alter_node_set_conflict_resolver

This function sets the behavior of conflict resolution on a given node.

Synopsis

```
bdr.alter_node_set_conflict_resolver(node_name text,
                                     conflict_type text,
                                     conflict_resolver text)
```

Parameters

- `node_name` - name of the node that is being changed
- `conflict_type` - conflict type for which the setting should be applied (see List of Conflict Types)
- `conflict_resolver` - which resolver to use for the given conflict type (see List of Conflict Resolvers)

Notes

Currently only the local node can be changed. The function call is not replicated. If you want to change settings on multiple nodes, the function must be run on each of them.

Note that the configuration change made by this function will override any default behavior of conflict resolutions specified via bdr.create_node_group or bdr.alter_node_group_config.

This function is transactional - the changes made can be rolled back and are visible to the current transaction.

List of Conflict Resolvers

There are several conflict resolvers available in BDR, with differing coverage of the conflict types they can handle:

- `error` - throws error and stops replication. Can be used for any conflict type.
- `skip` - skips processing of the remote change and continues replication with the next change. Can be used for `insert_exists`, `update_differing`, `update_origin_change`, `update_missing`, `update_recently_deleted`, `update_pkey_exists`, `delete_recently_updated`, `delete_missing`, `target_table_missing`, `target_column_missing` and `source_column_missing` conflict types.
- `skip_if_recently_dropped` - skip the remote change if it's for a table that does not exist on downstream because it has been recently (currently within 1 day) dropped on the downstream; throw an error otherwise. Can be used for the `target_table_missing` conflict type. `skip_if_recently_dropped` conflict resolver may pose challenges if a table with the same name is recreated shortly after it's dropped. In that case, one of the nodes may see the DMLs on the recreated table before it sees the DDL to recreate the table. It will then incorrectly skip the remote data, assuming that the table is recently dropped and cause data loss. It is hence recommended to not reuse the object names immediately after they are dropped along with this conflict resolver.
- `update_if_newer` - update if the remote row was committed later (as determined by the wall clock of the originating server) than the conflicting local row. If the timestamps are same, the node id is used as a tie-breaker to ensure that same row is picked on all nodes (higher nodeid wins). Can be used for `insert_exists`, `update_differing`, `update_origin_change` and `update_pkey_exists` conflict types.
- `update` - always perform the replicated action. Can be used for `insert_exists` (will turn the INSERT into UPDATE), `update_differing`, `update_origin_change`, `update_pkey_exists`, and `delete_recently_updated` (performs the delete).
- `insert_or_skip` - try to build a new row from available information sent by the origin and INSERT it; if there is not enough information available to build a full row, skip the change. Can be used for `update_missing` and `update_recently_deleted` conflict types.
- `insert_or_error` - try to build new row from available information sent by origin and INSERT it, if there is not enough information available to build full row, throw error and stop the replication. Can be used for `update_missing` and `update_recently_deleted` conflict types.
- `ignore` - ignore any missing target column and continue processing. Can be used for the `target_column_missing` conflict type.
- `ignore_if_null` - ignore a missing target column if the extra column in the remote row contains a NULL value, otherwise throw error and stop replication. Can be used for the `target_column_missing` conflict type.
- `use_default_value` - fill the missing column value with the default (including NULL if that's the column default) and continue processing. Any error while processing the default or violation of constraints (i.e. NULL default on NOT NULL column) will stop replication. Can be used for the `source_column_missing` conflict type.

The `insert_exists`, `update_differing`, `update_origin_change`, `update_missing` and

`delete_missing` conflict types can also be resolved by user-defined logic using Conflict Triggers.

List of Conflict Resolutions

The conflict resolution represents the kind of resolution chosen by the conflict resolver, and corresponds to the specific action which was taken to resolve the conflict.

The following conflict resolutions are currently supported for the `conflict_resolution` parameter:

- `apply_remote` - the remote (incoming) row has been applied
- `skip` - the processing of the row was skipped (no change has been made locally)
- `merge` - a new row was created, merging information from remote and local row
- `user` - user code (a conflict trigger) has produced the row that was written to the target table

Conflict Logging

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

bdr.alter_node_add_log_config

Add a named conflict logging configuration for a node.

Synopsis

```
bdr.alter_node_add_log_config(node_name text,
                              log_config_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

- `node_name` - name of the node that is being changed
- `log_config_name` - name of the logging configuration
- `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

Notes

Currently only the local node can be changed. The function call is not replicated. If you want to change settings on multiple nodes, the function must be run on each of them.

This function is transactional - the changes can be rolled back and are visible to the current transaction.

Listing Conflict Logging Configurations

The view `bdr.node_log_config` shows all the logging configurations. It lists the name of the logging configuration, where it logs and which conflicts type and resolution it logs.

Logging to a Table

Conflicts will be logged to a table if `log_to_table` is set non-NULL value. The target table can be any user table which contains any recognized columns. There is a preexisting table with all the recognized columns called `bdr.apply_log`, so this table can be used as the 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 `bdr.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 List of Conflict Types)
- `conflict_resolution` of type `integer` - conflict resolution chosen (see List of Conflict Resolutions)
- `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 it won't be saved.

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

bdr.alter_node_remove_log_config

Remove an existing conflict logging configuration from a node.

Synopsis

```
bdr.alter_node_remove_log_config(node_name text,
                                 log_config_name text)
```

Parameters

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

Notes

Currently only the local node can be changed. The function call is not replicated. If you want to change settings on multiple nodes, the function must be run on each of them.

This function is transactional - the changes can be rolled back and are visible to the current transaction.

Conflict Reporting

Conflicts logged to tables can be summarized in reports. This allows application owners to identify, understand and resolve conflicts and/or introduce application changes to prevent them.

```
SELECT nspname, relname
, date_trunc('day', local_time) :: date AS date
, count(*)
FROM bdr.apply_log
GROUP BY 1,2,3
ORDER BY 1,2;

 nspname | relname |    date    | count
```

```
---------+---------+------------+-------
 my_app  | test    | 2019-04-05 |     1
(1 row)
```

Sequences

Many applications require that unique surrogate ids be assigned to database entries. Often the database `SEQUENCE` object is used to produce these. In PostgreSQL these can be either a manually created sequence using the `CREATE SEQUENCE` command and retrieved by calling `nextval()` function, or `serial` and `bigserial` columns or alternatively `GENERATED BY DEFAULT AS IDENTITY` columns.

However, standard sequences in PostgreSQL are not multi-node aware, and only produce values that are unique on the local node. This is important because unique ids generated by such sequences will cause conflict and data loss (by means of discarded `INSERT`s) in multi-master replication.

BDR Global Sequences

For this reason BDR provides an application-transparent way to generate unique ids using sequences on bigint or bigserial datatypes across the whole BDR group, called **global sequences**.

BDR global sequences provide an easy way for applications to use the database to generate unique synthetic keys in an asynchronous distributed system that works for most cases, but not necessarily all.

Using global sequences allows you to avoid the problems with insert conflicts. If you define a `PRIMARY KEY` or `UNIQUE` constraint on a column which is using a global sequence, it is not possible for any node to ever get the same value as any other node. When BDR synchronizes inserts between the nodes, they can never conflict.

BDR global sequences extend PostgreSQL sequences, so are crash-safe. To use them you must have been granted the `bdr_application` role.

There are various possible algorithms for global sequences:

- Timeshard sequences
- Globally-allocated range sequences

Timeshard sequences generate values using an algorithm that does not require inter-node communication at any point, so is faster and more robust, as well as having the useful property of recording the timestamp at which they were created. Timeshard sequences have the restriction that they work only for 64-bit BIGINT datatypes and produce values 19 digits long, which may be too long for use in some host language datatypes such as Javascript Integer types. Globally-allocated sequences allocate a local range of values which can be replenished as-needed by inter-node consensus, making them suitable for either BIGINT or INTEGER sequences.

A global sequence can be created using the `bdr.alter_sequence_set_kind()` function. This function takes a standard PostgreSQL sequence and marks it as a BDR global sequence. It can also convert the sequence back to the standard PostgreSQL one (see below).

BDR also provides the configuration variable `bdr.default_sequence_kind`, which determines what kind of sequence will be created when the `CREATE SEQUENCE` command is executed or when a `serial`, `bigserial` or `GENERATED BY DEFAULT AS IDENTITY` column is created. Valid settings are

- `local` (the default) meaning that newly created sequences are the standard PostgreSQL (local) sequences.
- `galloc` which always creates globally-allocated range sequences.
- `timeshard` which creates time-sharded global sequences for BIGINT sequences, but will throw ERRORs when used with INTEGER sequences.

The `bdr.sequences` view shows information about individual sequence kinds.

`currval()` and `lastval()` work correctly for all types of global sequence.

Timeshard Sequences

The ids generated by timeshard sequences are loosely time-ordered so they can be used to get the approximate order of data insertion, like standard PostgreSQL sequences. Values generated within the same millisecond might be out of order, even on one node. The property of loose time-ordering means they are suitable for use as range partition keys.

Timeshard sequences work on one or more nodes and do not require any inter-node communication after the node join process completes. So they may continue to be used even if there's the risk of extended network partitions and are not affected by replication lag or inter-node latency.

Timeshard sequences generate unique ids in a different way to standard sequences. The algorithm uses 3 components for a sequence number. The first component of the sequence is a timestamp at the time of sequence number generation. The second component of the sequence number is the unique id assigned to each BDR node, which ensures that the ids from different nodes will always be different. Finally, the third component is the number generated by the local sequence itself.

While adding a unique node id to the sequence number would be enough to ensure there are no conflicts, we also want to keep another useful property of sequences, which is the fact that the ordering of the sequence numbers roughly corresponds to the order in which data was inserted into the table. Putting the timestamp first ensures this.

A few limitations and caveats apply to timeshard sequences.

Timeshard sequences are 64-bits wide and need a `bigint` or `bigserial`. Values generated will be at least 19 digits long. There is no practical 32-bit `integer` version, so cannot be used with `serial` sequences - use globally-allocated range sequences instead.

There is a limit of 8192 sequence values generated per millisecond on any given node for any given sequence. If more than 8192 sequences per millisecond are generated from one sequence on one node, the generated values will wrap around and could collide. There is no check on that for performance reasons; the value is not reset to 0 at the start of each ms. Collision will usually result in a `UNIQUE` constraint violation on `INSERT` or `UPDATE`. It cannot cause a replication conflict because sequence values generated on different nodes cannot *ever* collide since they contain the nodeid.

In practice this is harmless since values are not generated fast enough to trigger this limitation as there will be other work being done, rows inserted, indexes updated, etc. Despite that, applications should have a `UNIQUE` constraint in place where they absolutely rely on a lack of collisions.

Perhaps more importantly, the timestamp component will run out of values in the year 2050 and if used in combination with bigint, the values will wrap to negative numbers in the year 2033. This means that sequences generated after 2033 will have negative values. If you plan to deploy your application beyond this date, try one of UUIDs, KSUUIDs and Other Approaches mentioned below, or use globally-allocated range sequences instead.

The `INCREMENT` option on a sequence used as input for timeshard sequences is effectively ignored. This could be relevant for applications that do sequence ID caching, like many object-relational mapper (ORM) tools, notably Hibernate. Because the sequence is time-based this has little practical effect since the sequence will have advanced to a new non-colliding value by the time the application can do anything with the cached values.

Similarly, the `START`, `MINVALUE`, `MAXVALUE` and `CACHE` settings may be changed on the underlying sequence, but there is no benefit to doing so. The sequence's low 14 bits are used and the rest is discarded, so the value range limits do not affect the function's result. For the same reason, `setval()` is not useful for timeshard sequences.

Globally-allocated range Sequences

The globally-allocated range (or `galloc`) sequences allocate ranges (chunks) of values to each node. When the local range is used up, a new range is allocated globally by consensus amongst the other nodes. This uses the key space efficiently but requires that the local node be connected to a majority of the nodes in the cluster for the sequence generator to progress when the currently assigned local range has been used up.

Unlike timeshard sequences, galloc sequences support all sequence data types provided by PostgreSQL - smallint, integer and bigint. This means that galloc sequences can be used in environments where 64-bit sequences are problematic, such as using integers in javascript since that supports only 53-bit values, or when the sequence is displayed on output with limited space.

The range assigned by each voting is currently predetermined based on the datatype the sequence is using:

- smallint - 1 000 numbers
- integer - 1 000 000 numbers
- bigint - 1 000 000 000 numbers

Each node will allocate 2 chunks of seq_chunk_size, one for the current use plus a reserved chunk for future usage, so the values generated from any one node will increase monotonically. However, viewed globally, the values generated will not be ordered at all. This could cause a loss of performance due to the effects on b-tree indexes and will typically mean that generated values will not be useful as range partition keys.

The main downside of the galloc sequences is that once the assigned range is used up, the sequence generator has to ask for consensus about the next range for the local node which requires inter-node communication, which could lead to delays or operational issues if the majority of the BDR group is not accessible. This may be avoided in later releases.

The `CACHE`, `START`, `MINVALUE` and `MAXVALUE` options work correctly with galloc sequences, however you need to set them before transforming the sequence to galloc kind. The `INCREMENT BY` option also works correctly, however, you cannot assign an increment value which is equal to or more than the above ranges assigned for each sequence datatype. `setval()` does not reset the global state for galloc sequences and should not be used.

A few limitations apply to galloc sequences. BDR tracks galloc sequences in a special BDR catalog `bdr.sequence_alloc`. This catalog is required to track the currently allocated chunks for the galloc sequences. The sequence name and namespace is stored in this catalog. Since the sequence chunk allocation is managed via RAFT whereas any changes to the sequence name/namespace is managed via replication stream, BDR currently does not support renaming galloc sequences, or moving them to another namespace or renaming the namespace that contains a galloc sequence. The user should be mindful of this limitation while designing application schema.

Usage

Before transforming a local sequence to galloc, you need to take care of these prerequisites:

When sequence kind is altered to galloc, it will be rewritten and restart from the defined start value of the local sequence. If this happens on an existing sequence in a production database you will need to query the current value then set the start value appropriately. To assist with this use case, BDR allows users to pass a starting value with the function `bdr.alter_sequence_set_kind()`. If you are already using offset and you have writes from multiple nodes, you need to check what is the greatest used value and restart the sequence at least to the next value.

```
-- determine highest sequence value across all nodes
SELECT max((x->'response'->0->>'nextval')::bigint)
    FROM json_array_elements(
        bdr.run_on_all_nodes(
            E'SELECT nextval(\'public.sequence\');'
            )::jsonb AS x;

-- turn into a galloc sequence
SELECT bdr.alter_sequence_set_kind('public.sequence'::regclass, 'galloc', $MAX+MARGIN);
```

Since users cannot lock a sequence, you must leave a $MARGIN value to allow operations to continue while the max() value is queried.

The `bdr.sequence_alloc` table will give information on the chunk size and what ranges are allocated around the whole cluster. In this example we started our sequence from `333`, and we have two nodes in the cluster, we can see that we have a number of allocation 4, that is 2 per node and the chunk size is 1000000 that is related to an integer sequence.

```
SELECT * FROM bdr.sequence_alloc
    WHERE seqid = 'public.categories_category_seq'::regclass;
          seqid          | seq_chunk_size | seq_allocated_up_to | seq_nallocs |
seq_last_alloc
-------------------------+----------------+---------------------+-------------+--------------
```

```
 categories_category_seq |        1000000 |             4000333 |
4 | 2020-05-21 20:02:15.957835+00
(1 row)
```

To see the ranges currently assigned to a given sequence on each node, use these queries:

- Node `Node1` is using range from `333` to `2000333`.

```
SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = '(0,2)'; -- first range
 range_start | range_end
-------------+-----------
         334 |   1000333
(1 row)

SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = '(0,3)'; -- second range
 range_start | range_end
-------------+-----------
     1000334 |   2000333
(1 row)
```

- Node `Node2` is using range from `2000004` to `4000003`.

```
SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = '(0,2)'; -- first range
 range_start | range_end
-------------+-----------
     2000334 |   3000333
(1 row)

SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = '(0,3)'; -- second range
 range_start | range_end
-------------+-----------
     3000334 |   4000333
```

NOTE You can't combine it to single query (like WHERE ctid IN ('(0,2)', '(0,3)')) as that will still only show the first range.

When a node finishes a chunk, it will ask a consensus for a new one and get the first available; in our case, it will be from 4000334 to 5000333. This will be the new reserved chunk, and it will start to consume the old reserved chunk.

UUIDs, KSUUIDs and Other Approaches

There are other ways to generate globally unique ids without using the global sequences which can be used with BDR. For example:

- UUIDs, and their BDR variant, KSUUIDs
- Local sequences with a different offset per node (i.e. manual)
- An externally co-ordinated natural key

Please note that BDR applications **cannot** use other methods safely: counter-table based approaches relying on `SELECT ... FOR UPDATE`, `UPDATE ... RETURNING ...` or similar for sequence generation will not work correctly in BDR, because BDR doesn't take row locks between nodes. The same values will be generated on more than one node. For the same reason the usual strategies for "gapless" sequence generation do not work with BDR. In most cases the application should coordinate generation of sequences that must be gapless from some external source using two-phase commit, or it should only generate them on one node in the BDR group.

UUIDs and KSUUIDs

`UUID` keys instead avoid sequences entirely and use 128-bit universal unique identifiers. These are random or pseudorandom values that are large enough that it's nearly impossible for the same value to be generated twice. There is no need for nodes to have continuous communication when using `UUID` keys.

In the incredibly unlikely event of a collision, conflict detection will choose the newer of the two inserted records to retain. Conflict logging, if enabled, will record such an event, but it is *exceptionally* unlikely to ever occur, since collisions only become practically likely after about `2^64` keys have been generated.

The main downside of `UUID` keys is that they're somewhat space- and network inefficient, consuming more space not only as a primary key, but also where referenced in foreign keys and when transmitted on the wire. Additionally, not all applications cope well with `UUID` keys.

BDR provides functions for working with a K-Sortable variant of `UUID` data, known as KSUUID, which generates values that can be stored using PostgreSQL's standard `UUID` data type. A `KSUUID` value is similar to `UUIDv1` in that it stores both timestamp and random data, following the `UUID` standard. The difference is that `KSUUID` is K-Sortable, meaning that it's weakly sortable by timestamp. This makes it more useful as a database key, improving the effectiveness of search, allows natural time-sorting of result data and because it produces more compact `btree` indexes. Unlike `UUIDv1`, `KSUUID` values do not include the MAC of the computer on which they were generated, so there should be no security concerns from using `KSUUID`s.

`KSUUID` v2 is now recommended in all cases. Values generated are directly sortable with regular comparison operators.

There are two versions of `KSUUID` in BDR. The legacy `KSUUID` v1 is now deprecated but kept in order to support existing installations and should not be used for new installations. The internal contents of the v1

and v2 are not compatible and as such the functions to manipulate them are also not compatible. The v2 of `KSUUID` also no longer stores the `UUID` version number.

Step & Offset Sequences

In offset-step sequences, a normal PostgreSQL sequence is used on each node. Each sequence increments by the same amount and starts at differing offsets. For example with step 1000, node1's sequence generates 1001, 2001, 3001, and so on, node2's generates 1002, 2002, 3002, etc. This scheme works well even if the nodes cannot communicate for extended periods, but requires that the designer specify a maximum number of nodes when establishing the schema and requires per-node configuration. However, mistakes can easily lead to overlapping sequences.

It is relatively simple to configure this approach with BDR by creating the desired sequence on one node, like this:

```
CREATE TABLE some_table (
    generated_value bigint primary key
);

CREATE SEQUENCE some_seq INCREMENT 1000 OWNED BY some_table.generated_value;

ALTER TABLE some_table ALTER COLUMN generated_value SET DEFAULT nextval('some_seq');
```

... then on each node calling `setval()` to give each node a different offset starting value, e.g.:

```
-- On node 1
SELECT setval('some_seq', 1);

-- On node 2
SELECT setval('some_seq', 2);

 -- ... etc
```

You should be sure to allow a large enough `INCREMENT` to leave room for all the nodes you may ever want to add, since changing it in future is difficult and disruptive.

If you use `bigint` values there is no practical concern about key exhaustion even if you use offsets of 10000 or more. You'll need hundreds of years with hundreds of machines doing millions of inserts per second to have any chance of approaching exhaustion.

BDR does not currently offer any automation for configuration of the per-node offsets on such step/offset sequences.

Composite Keys

A variant on step/offset sequences is to use a composite key composed of `PRIMARY KEY (node_number, generated_v` where the node number is usually obtained from a function that returns a different number on each node.

Such a function may be created by temporarily disabling DDL replication and creating a constant SQL function, or by using a one-row table that isn't part of a replication set to store a different value in each node.

Global Sequence Management Interfaces

BDR provides an interface for converting between a standard PostgreSQL sequence and the BDR global sequence.

Note that the following functions are considered to be `DDL` so DDL replication and global locking applies to them.

bdr.alter_sequence_set_kind

Sets the kind of a sequence. Once set, `seqkind` is only visible via the `bdr.sequences` view; in all other ways the sequence will appear as a normal sequence.

BDR treats this function as `DDL` so DDL replication and global locking applies, if that is currently active. See DDL Replication.

Cannot be used on a sequence created for a `SERIAL` datatype.

Synopsis

```
bdr.alter_sequence_set_kind(seqoid regclass, seqkind text, int64 start DEFAULT NULL)
```

Parameters

- `seqoid` - name or Oid of the sequence to be altered
- `seqkind` - `local` for a standard PostgreSQL sequence, `timeshard` for BDR global sequence which uses the "time and sharding" based algorithm described in the BDR Global Sequences section, or `galloc` for globally-allocated range sequences which use consensus between nodes to assign unique ranges of sequence numbers to each node
- `start` - start value for local and galloc sequence - passing any NOT NULL value is same as calling `ALTER SEQUENCE ... START WITH ... RESTART`.

Notes

When changing the sequence kind to `galloc`, the first allocated range for that sequence will use the sequence start value as starting point. When there are already existing values used by the sequence before it was changed to `galloc`, it is recommended to move the starting point so that the newly generated values will not conflict the existing ones using the following command:

```
ALTER SEQUENCE seq_name START starting_value RESTART
```

This function uses the same replication mechanism as `DDL` statements. This means the replication is affected by the ddl filters configuration.

The function will take a global `DDL` lock. It will also lock the sequence locally.

This function is transactional - the effects can be rolled back with the `ROLLBACK` of the transaction and the changes are visible to the current transaction.

The `bdr.alter_sequence_set_kind` function can be only executed by the owner of the sequence, unless `bdr.backwards_compatibility` is set is set to 30618 or below.

bdr.extract_timestamp_from_timeshard

Extract the timestamp component of the `timeshard` sequence. The return value is of type "timestamptz".

Synopsis

```
bdr.extract_timestamp_from_timeshard(timeshard_seq bigint)
```

Parameters

- `timeshard_seq` - value of a timeshard sequence

Notes

This function is only executed on the local node.

bdr.extract_nodeid_from_timeshard

Extract the nodeid component of the `timeshard` sequence.

Synopsis

```
bdr.extract_nodeid_from_timeshard(timeshard_seq bigint)
```

Parameters

- `timeshard_seq` - value of a timeshard sequence

Notes

This function is only executed on the local node.

bdr.extract_localseqid_from_timeshard

Extract the local sequence value component of the `timeshard` sequence.

Synopsis

```
bdr.extract_localseqid_from_timeshard(timeshard_seq bigint)
```

Parameters

- `timeshard_seq` - value of a timeshard sequence

Notes

This function is only executed on the local node.

bdr.timestamp_to_timeshard

Convert a timestamp value to a dummy timeshard sequence value.

This is useful for doing indexed searches or comparisons of values in the timeshard column and for a specific timestamp.

For example, given a table `foo` with a column `id` which is using a `timeshard` sequence, we can get the number of changes since yesterday midnight like this:

```
SELECT count(1) FROM foo WHERE id > bdr.timestamp_to_timeshard('yesterday')
```

A query formulated this way will use an index scan on the column `id`.

Synopsis

```
bdr.timestamp_to_timeshard(ts timestamptz)
```

Parameters

- `ts` - timestamp to be used for the timeshard sequence generation

Notes

This function is only executed on local node.

KSUUID v2 Functions

Functions for working with `KSUUID` v2 data, K-Sortable UUID data.

bdr.gen_ksuuid_v2

Generates a new `KSUUID` v2 value, using the value of timestamp passed as an argument or current system time if NULL is passed. If you want to generate KSUUID automatically using system time pass NULL argument.

The return value is of type “UUID”.

Synopsis

```
bdr.gen_ksuuid_v2(timestamptz)
```

Notes

This function is only executed on the local node.

bdr.ksuuid_v2_cmp

Compare the `KSUUID` v2 values.

Returns 1 if first value is newer, -1 if second value is lower, or zero if they are equal.

Synopsis

```
bdr.ksuuid_v2_cmp(uuid, uuid)
```

Parameters

- `UUID` - `KSUUID` v2 to compare

Notes

This function is only executed on local node.

bdr.extract_timestamp_from_ksuuid_v2

Extract the timestamp component of `KSUUID` v2. The return value is of type “timestamptz”.

Synopsis

```
bdr.extract_timestamp_from_ksuuid_v2(uuid)
```

Parameters

- `UUID` - `KSUUID` v2 value to extract timestamp from

Notes

This function is only executed on the local node.

KSUUID v1 Functions

Functions for working with `KSUUID` v1 data, K-Sortable UUID data(v1).

bdr.gen_ksuuid

Generates a new `KSUUID` v1 value, using the current system time. The return value is of type "UUID".

Synopsis

```
bdr.gen_ksuuid()
```

Notes

This function is only executed on the local node.

bdr.uuid_v1_cmp

Compare the `KSUUID` v1 values.

Returns 1 if first value is newer, -1 if second value is lower, or zero if they are equal.

Synopsis

```
bdr.uuid_v1_cmp(uuid, uuid)
```

Notes

This function is only executed on the local node.

Parameters

- `UUID` - `KSUUID` v1 to compare

bdr.extract_timestamp_from_ksuuid

Extract the timestamp component of `KSUUID` v1 or `UUIDv1` values. The return value is of type "timestamptz".

Synopsis

```
bdr.extract_timestamp_from_ksuuid(uuid)
```

Parameters

- `UUID` - `KSUUID` v1 value to extract timestamp from

Notes

This function is only executed on the local node.

Column-Level Conflict Detection

By default, conflicts are resolved at row level. That is, when changes from two nodes conflict, we pick either the local or remote tuple and discard the other one. For example, we may compare commit timestamps for the two conflicting changes and keep the newer one. This ensures that all nodes converge to the same result, and establishes commit-order-like semantics on the whole cluster.

However, in some cases it may be appropriate to resolve conflicts at the column-level rather than the row-level.

Consider a simple example, where we have a table "t" with two integer columns "a" and "b", and a single row `(1,1)`. Assume that on one node we execute:

```
UPDATE t SET a = 100
```

... while on another node we concurrently (before receiving the preceding `UPDATE`) execute:

```
UPDATE t SET b = 100
```

This results in an `UPDATE-UPDATE` conflict. With the `update_if_newer` conflict resolution, we compare the commit timestamps and keep the new row version. Assuming the second node committed last, we end up with `(1,100)`, effectively discarding the change to column "a".

For many use cases this is the desired and expected behaviour, but for some this may be an issue - consider for example a multi-node cluster where each part of the application is connected to a different node, updating a dedicated subset of columns in a shared table. In that case, the different components may step on each other's toes, overwriting their changes.

For such use cases, it may be more appropriate to resolve conflicts on a given table at the column-level. To achieve that, BDR will track the timestamp of the last change for each column separately, and use that to pick the most recent value (essentially `update_if_newer`).

Applied to the previous example, we'll end up with `(100,100)` on both nodes, despite neither of the nodes ever seeing such a row.

When thinking about column-level conflict resolution, it may be useful to see tables as vertically partitioned, so that each update affects data in only one slice. This eliminates conflicts between changes to different subsets of columns. In fact, vertical partitioning may even be a practical alternative to column-level conflict resolution.

Column-level conflict resolution requires the table to have `REPLICA IDENTITY FULL`. The `bdr.alter_table_conflict_detection` function does check that, and will fail with an error otherwise.

Enabling and Disabling Column-Level Conflict Resolution

The Column-Level Conflict Resolution is managed by the bdr.alter_table_conflict_detection() function.

Example

To illustrate how the `bdr.alter_table_conflict_detection()` is used, consider this example that creates a trivial table `test_table` and then enable column-level conflict resolution on it:

```
db=# CREATE TABLE my_app.test_table (id SERIAL PRIMARY KEY, val INT);
CREATE TABLE

db=# ALTER TABLE my_app.test_table REPLICA IDENTITY FULL;
ALTER TABLE

db=# SELECT bdr.alter_table_conflict_detection('my_app.test_table'::regclass, 'column_modify_
 column_timestamps_enable
--------------------------

(1 row)

db=# \d test_table
                                                                    Table "my_app.test_tal
 Column |         Type          | Collation | Nullable |
Default
--------+-----------------------+-----------+----------+-----------------------------------
 id     | integer               |           | not null | nextval('test_table_id_seq'::regclas
 val    | integer               |           |          |
 cts    | bdr.column_timestamps |           | not null | bdr.column_timestamps_create('curren
Indexes:
    "test_table_pkey" PRIMARY KEY, btree (id)
Triggers:
    bdr_clcd_before_insert BEFORE INSERT ON test_table FOR EACH ROW EXECUTE PROCEDURE bdr.colu
    bdr_clcd_before_update BEFORE UPDATE ON test_table FOR EACH ROW EXECUTE PROCEDURE bdr.colu
```

You can see that the function added a new `cts` column (as specified in the function call), but it also created two triggers (`BEFORE INSERT` and `BEFORE UPDATE`) that are responsible for maintaining timestamps in the new column before each change.

Also worth mentioning is that the new column specifies `NOT NULL` with a default value, which can however leverage the fast default mode implemented in PostgreSQL 11. That means the `ALTER TABLE ... ADD COLUMN` does not perform a table rewrite.

Note: We discourage using columns with the `bdr.column_timestamps` data type for other purposes as it may have various negative effects (it switches the table to column-level conflict resolution, which will not work correctly without the triggers etc.).

Listing Table with Column-Level Conflict Resolution

Tables having column-level conflict resolution enabled can be listed with the following query, which detects the presence of a column of type `bdr.column_timestamp`:

```
SELECT nc.nspname, c.relname
FROM pg_attribute a
JOIN (pg_class c JOIN pg_namespace nc ON c.relnamespace = nc.oid)
  ON a.attrelid = c.oid
JOIN (pg_type t  JOIN pg_namespace nt ON t.typnamespace = nt.oid)
  ON a.atttypid = t.oid
WHERE NOT pg_is_other_temp_schema(nc.oid)
  AND nt.nspname = 'bdr'
  AND t.typname = 'column_timestamps'
  AND NOT a.attisdropped
  AND c.relkind IN ('r', 'v', 'f', 'p');
```

bdr.column_timestamps_create

This function creates column-level conflict resolution. This is called within `column_timestamp_enable`.

Synopsis

```
bdr.column_timestamps_create(p_source cstring, p_timestamp timestampstz)
```

Parameters

- `p_source` - The two options are 'current' or 'commit'.
- `p_timestamp` - Timestamp is dependent on the source chosen: if 'commit', then TIMESTAMP_SOURCE_COMMIT; if 'current', then TIMESTAMP_SOURCE_CURRENT.

DDL Locking

When enabling or disabling column timestamps on a table, the code uses DDL locking to ensure that there are no pending changes from before the switch, to ensure we only see conflicts with either timestamps in both tuples or neither of them. Otherwise, the code might unexpectedly see timestamps in the local tuple and NULL in the remote one. It also ensures that the changes are resolved the same way (column-level or row-level) on all nodes.

Current vs Commit Timestamp

An important question is what timestamp to assign to modified columns.

By default, the timestamp assigned to modified columns is the current timestamp, as if obtained from `clock_timestamp`. This is simple, and for many cases it is perfectly correct (e.g. when the conflicting rows modify non-overlapping subsets of columns).

It may however have various unexpected effects:

- The timestamp changes during statement execution, so if an `UPDATE` affects multiple rows, each will get a slightly different timestamp. This means that the effects of concurrent changes may get "mixed" in various ways (depending on how exactly the changes performed on different nodes interleave).
- The timestamp is unrelated to the commit timestamp, and using it to resolve conflicts means that the result is not equivalent to the commit order, which means it likely is not serializable.

Note: We may add statement and transaction timestamps in the future, which would address issues with mixing effects of concurrent statements or transactions. Still, neither of these options can ever produce results equivalent to commit order.

It is possible to also use the actual commit timestamp, although this feature is currently considered experimental. To use the commit timestamp, set the last parameter to `true` when enabling column-level conflict resolution:

```
SELECT bdr.column_timestamps_enable('test_table'::regclass, 'cts', true);
```

Commit timestamps currently have a couple of restrictions that are explained in the "Limitations" section.

Inspecting Column Timestamps

The column storing timestamps for modified columns is maintained automatically by triggers, and must not be modified directly. It may be useful to inspect the current timestamps value, for example while investigating how a particular conflict was resolved.

There are three functions for this purpose:

- `bdr.column_timestamps_to_text(bdr.column_timestamps)`

This function returns a human-readable representation of the timestamp mapping, and is used when casting the value to `text`:

```
db=# select cts::text from test_table;
                                                  cts
-------------------------------------------------------------------------------------------
 {source: current, default: 2018-09-23 19:24:52.118583+02, map: [2 : 2018-09-23 19:25:02.5906
(1 row)
```

- `bdr.column_timestamps_to_jsonb(bdr.column_timestamps)`

This function turns a JSONB representation of the timestamps mapping, and is used when casting the value to `jsonb`:

```
db=# select jsonb_pretty(cts::jsonb) from test_table;
                    jsonb_pretty
-------------------------------------------------
 {                                               +
     "map": {                                    +
         "2": "2018-09-23T19:24:52.118583+02:00"  +
     },                                          +
     "source": "current",                        +
     "default": "2018-09-23T19:24:52.118583+02:00"+
 }
(1 row)
```

- `bdr.column_timestamps_resolve(bdr.column_timestamps, xid)`

This function updates the mapping with the commit timestamp for the attributes modified by the most recent transaction (if it already committed). This only matters when using the commit timestamp. For example in this case, the last transaction updated the second attribute (with `attnum = 2`):

```
test=# select cts::jsonb from test_table;
                                                                                                  cts
-------------------------------------------------------------------------------------------------------------------------------
 {"map": {"2": "2018-09-23T19:29:55.581823+02:00"}, "source": "commit", "default": "2018-09-2
(1 row)

db=# select bdr.column_timestamps_resolve(cts, xmin)::jsonb from test_table;
                                                 column_timestamps_resolve
-------------------------------------------------------------------------------------------------------------------------------
 {"map": {"2": "2018-09-23T19:29:55.581823+02:00"}, "source": "commit", "default": "2018-09-2
(1 row)
```

Handling column conflicts using CRDT Data Types

By default, column-level conflict resolution simply picks the value with a higher timestamp and discards the other one. It is however possible to reconcile the conflict in different (more elaborate) ways, for example using CRDT types that allow “merging” the conflicting values without discarding any information.

While pglogical does not include any such data types, it allows adding them separately and registering them in a catalog `crdt_handlers`. Aside from the usual data type functions (input/output, ...) each CRDT type has to implement a merge function, which takes exactly three arguments (local value, old remote value, new remote value) and produces a value merging information from those three values.

Limitations

- The attributes modified by an `UPDATE` are determined by comparing the old and new row in a trigger. This means that if the attribute does not change a value, it will not be detected as modified

even if it is explicitly set. For example, `UPDATE t SET a = a` will not mark `a` as modified for any row. Similarly, `UPDATE t SET a = 1` will not mark `a` as modified for rows that are already set to 1.

- For `INSERT` statements, we do not have any old row to compare the new one to, so we consider all attributes to be modified and assign them a new timestamp. This applies even for columns that were not included in the `INSERT` statement and received default values. We could detect which attributes have a default value, but it is not possible to decide if it was included automatically or specified explicitly by the user.

This effectively means column-level conflict resolution does not work for `INSERT-INSERT` conflicts (even if the `INSERT` statements specify different subsets of columns, because the newer row will have all timestamps newer than the older one).

- By treating the columns independently, it is easy to violate constraints in a way that would not be possible when all changes happen on the same node. Consider for example a table like this:

```
CREATE TABLE t (id INT PRIMARY KEY, a INT, b INT, CHECK (a > b));
INSERT INTO t VALUES (1, 1000, 1);
```

. . . and assume one node does:

```
UPDATE t SET a = 100;
```

. . . while another node does concurrently:

```
UPDATE t SET b = 500;
```

Each of those updates is valid when executed on the initial row, and so will pass on each node. But when replicating to the other node, the resulting row violates the `CHECK (A > b)` constraint, and the replication will stop until the issue is resolved manually.

- The column storing timestamp mapping is managed automatically. Do not specify or override the value in your queries, as it may result in unpredictable effects (we do ignore the value where possible anyway).
- The timestamp mapping is maintained by triggers, but the order in which triggers execute does matter. So if you have custom triggers that modify tuples and are executed after the `pgl_clcd_` triggers, the modified columns will not be detected correctly.
- When using regular timestamps to order changes/commits, it is possible that the conflicting changes have exactly the same timestamp (because two or more nodes happened to generate the same timestamp). This risk is not unique to column-level conflict resolution, as it may happen even for regular row-level conflict resolution, and we use node id as a tie-breaker in this situation (the higher node id wins), which ensures that same changes are applied on all nodes.

- It is possible that there is a clock skew between different nodes. While it may induce somewhat unexpected behavior (discarding seemingly newer changes because the timestamps are inverted), clock skew between nodes can be managed using the parameters `bdr.maximum_clock_skew` and `bdr.maximum_clock_skew_action`.
- The underlying pglogical subscription must not discard any changes, which could easily cause divergent errors (particularly for CRDT types). The subscriptions must have `ignore_redundant_updates` set to false (which is the default).

Existing groups created with non-default value for `ignore_redundant_updates` can be altered like this:

```
SELECT bdr.alter_node_group_config('group', ignore_redundant_updates := false);
```

Conflict-free Replicated Data Types

Conflict-free replicated data types (CRDT) support merging values from concurrently modified rows, instead of discarding one of the rows (which is what traditional conflict resolution does).

Each CRDT type is implemented as a separate PostgreSQL data type, with an extra callback added to the `bdr.crdt_handlers` catalog. The merge process happens within pglogical on the apply side; no additional user action is required.

CRDTs require the table to have column-level conflict resolution enabled as documented in the CLCD chapter.

The only action taken by the user is the use of a particular data type in CREATE/ALTER TABLE, rather than standard built-in data types such as integer; e.g. consider the following table with one regular integer counter and a single row:

```
CREATE TABLE non_crdt_example (
    id      integer             PRIMARY KEY,
    counter integer             NOT NULL DEFAULT 0
);

INSERT INTO non_crdt_example (id) VALUES (1);
```

If we issue the following SQL on two nodes at same time:

```
UPDATE non_crdt_example
   SET counter = counter + 1    -- "reflexive" update
 WHERE id = 1;
```

... the resulting values can be seen using this query, after both updates are applied:

```
SELECT * FROM non_crdt_example WHERE id = 1;
   id |   counter
 -----+-----------
    1 |         1
(1 row)
```

... showing that we've lost one of the increments, due to the update_if_newer conflict resolver. If you use the CRDT counter data type instead, you should observe something like this:

```
CREATE TABLE crdt_example (
    id      integer             PRIMARY KEY,
    counter bdr.crdt_gcounter   NOT NULL DEFAULT 0
);

ALTER TABLE crdt_example REPLICA IDENTITY FULL;

SELECT bdr.alter_table_conflict_detection('crdt_example', 'column_modify_timestamp', 'cts');
```

```
INSERT INTO crdt_example (id) VALUES (1);
```

Again we issue the following SQL on two nodes at same time, then wait for the changes to be applied:

```
UPDATE crdt_example
   SET counter = counter + 1    -- "reflexive" update
 WHERE id = 1;

SELECT id, counter FROM crdt_example WHERE id = 1;
   id |   counter
 -----+-----------
    1 |         2
(1 row)
```

This shows that CRDTs correctly allow accumulator columns to work, even in the face of asynchronous concurrent updates that otherwise conflict.

The `crdt_gcounter` type is an example of state-based CRDT types, that work only with reflexive UPDATE SQL, such as `x = x + 1`, as shown above.

The `bdr.crdt_raw_value` configuration option determines whether queries return the current value or the full internal state of the CRDT type. By default only the current numeric value is returned. When set to `true`, queries return representation of the full state - the special hash operator (#) may be used to request only the current numeric value without using the special operator (this is the default behavior).

Note: The `bdr.crdt_raw_value` applies only formatting of data returned to clients; i.e. simple column references in the select list. Any column references in other parts of the query (e.g. `WHERE` clause or even expressions in the select list) may still require use of the # operator.

Another class of CRDT data types exists, which we refer to as “delta CRDT” types (and are a special subclass of operation-based CRDTs, as explained later).

With delta CRDTs, any update to a value is automatically compared to the previous value on the same node and then a change is applied as a delta on all other nodes.

```
CREATE TABLE crdt_delta_example (
    id       integer            PRIMARY KEY,
    counter  bdr.crdt_delta_counter NOT NULL DEFAULT 0
);

ALTER TABLE crdt_delta_example REPLICA IDENTITY FULL;

SELECT bdr.alter_table_conflict_detection('crdt_delta_example', 'column_modify_timestamp', 'c

INSERT INTO crdt_delta_example (id) VALUES (1);
```

If we issue the following SQL on two nodes at same time:

```
UPDATE crdt_delta_example
   SET counter = 2          -- notice NOT counter = counter + 2
 WHERE id = 1;
```

The resulting values can be seen using this query, after both updates are applied:

```
SELECT id, counter FROM crdt_delta_example WHERE id = 1;
   id | counter
 -----+---------
    1 |       4
(1 row)
```

With a regular `integer` column the result would be `2`, of course. But when we UPDATE the row with a delta CRDT counter, we start with the OLD row version, make a NEW row version and send both to the remote node, where we compare them with the version we find there (let's call that the LOCAL version). Standard CRDTs merge the NEW and the LOCAL version, while delta CRDTs compare the OLD and NEW versions and apply the delta to the LOCAL version.

The CRDT types are installed as part of `bdr` into the `bdr` schema. For convenience, the basic operators (`+`, `#` and `!`) and a number of common aggregate functions (`min`, `max`, `sum` and `avg`) are created in `pg_catalog`, to make them available without having to tweak `search_path`.

An important question is how query planning and optimization works with these new data types. CRDT types are handled transparently - both `ANALYZE` and the optimizer work, so estimation and query planning works fine, without having to do anything else.

State-based and operation-based CRDTs

Following the notation from [1], we do implement both operation-based and state-based CRDTs.

Operation-based CRDT Types (CmCRDT)

The implementation of operation-based types is quite trivial, because the operation is not transferred explicitly but computed from the old and new row received from the remote node.

Currently, we implement these operation-based CRDTs:

- `crdt_delta_counter` - `bigint` counter (increments/decrements)
- `crdt_delta_sum` - `numeric` sum (increments/decrements)

These types leverage existing data types (e.g. `crdt_delta_counter` is a domain on a `bigint`), with a little bit of code to compute the delta.

This approach is possible only for types where we know how to compute the delta, but the result is very simple and cheap (both in terms of space and CPU), and has a couple of additional benefits (e.g. we can leverage operators / syntax for the under-lying data type).

The main disadvantage is that it's not possible to reset this value reliably in an asynchronous and concurrent environment.

Note: We could also implement more complicated operation-based types by creating custom data types, storing the state and the last operation (we decode and transfer every individual change, so we don't need multiple operations). But at that point we lose the main benefits (simplicity, reuse of existing data types) without gaining any advantage compared to state-based types (still no capability to reset, . . .), except for the space requirements (we don't need a per-node state).

State-based CRDT Types (CvCRDT)

State-based types require a more complex internal state, and so can't use the regular data types directly the way operation-based types do.

Currently, we implement four state-based CRDTs:

- `crdt_gcounter` - `bigint` counter (increment-only)
- `crdt_gsum` - `numeric` sum/counter (increment-only)
- `crdt_pncounter` - `bigint` counter (increments/decrements)
- `crdt_pnsum` - `numeric` sum/counter (increments/decrements)

The internal state typically includes per-node information, increasing the on-disk size but allowing additional benefits. The need to implement custom data types implies more code (in/out functions and operators).

The advantage is the ability to reliably reset the values, a somewhat self-healing nature in the presence of lost changes (which should not happen in properly-operated cluster), and the ability to receive changes from other than source nodes.

Consider for example that a value is modified on node A, and the change gets replicated to B, but not C (due to network issue between A and C). If B modifies the value, and this change gets replicated to C, it will include even the original change from A. With operation-based CRDTs the node C would not receive the change until the A-C network connection starts working again.

The main disadvantages of CvCRDTs are higher costs, both in terms of disk space - we need a bit of information for each node, including nodes that have been already removed from the cluster). The complex nature of the state (serialized into varlena types) means increased CPU usage.

Disk-Space Requirements

An important consideration is the overhead associated with CRDT types, particularly the on-disk size.

For operation-based types this is rather trivial, because the types are merely domains on top of other types, and so have the same disk space requirements (no matter how many nodes are there).

- `crdt_delta_counter` - same as `bigint` (8 bytes)

- `crdt_delta_sum` - same as `numeric` (variable, depending on precision and scale)

There is no dependency on the number of nodes, because operation-based CRDT types do not store any per-node information.

For state-based types the situation is more complicated. All the types are variable-length (stored essentially as a `bytea` column), and consist of a header and a certain amount of per-node information for each node that *modified* the value.

For the `bigint` variants, formulas computing approximate size are (`N` denotes the number of nodes that modified this value):

- `crdt_gcounter` - `32B (header) + N * 12B (per-node)`
- `crdt_pncounter` - `48B (header) + N * 20B (per-node)`

For the `numeric` variants there is no exact formula, because both the header and per-node parts include `numeric` variable-length values. To give you an idea of how many such values we need to keep:

- `crdt_gsum`
- fixed: `20B (header) + N * 4B (per-node)`
- variable: `(2 + N)` `numeric` values
- `crdt_pnsum`
- fixed: `20B (header) + N * 4B (per-node)`
- variable: `(4 + 2 * N)` `numeric` values

Note: It does not matter how many nodes are in the cluster, if the values are never updated on multiple nodes. It also does not matter if the updates were concurrent (causing a conflict) or not.

Note: It also does not matter how many of those nodes were already removed from the cluster. There is no way to compact the state yet.

CRDT Types vs Conflicts Handling

As tables may contain both CRDT and non-CRDT columns (in fact, most columns are expected to be non-CRDT), we need to do both the regular conflict resolution and CRDT merge.

The conflict resolution happens first, and is responsible for deciding which tuple to keep (applytuple) and which one to discard. The merge phase happens next, merging data for CRDT columns from the discarded tuple into the applytuple.

Note: This makes CRDT types somewhat more expensive compared to plain conflict resolution, because the merge needs to happen every time, even when the conflict resolution can use one of the fast-paths (modified in the current transaction, etc.).

Note: CRDT types require properly working conflict detection, which is only supported by HeapWriter. That means other writers (e.g. SPI) do not support CRDTs (i.e. the merging will not work).

CRDT Types vs. Conflict Reporting

By default, detected conflicts are written into the server log. Without CRDT types this makes perfect sense, because the conflict resolution essentially throws away one half of the available information (local or remote row, depending on configuration). This presents a data loss.

CRDT types allow both parts of the information to be combined without throwing anything away, eliminating the data loss issue. This makes the conflict reporting unnecessary.

For this reason, we skip the conflict reporting when the conflict can be fully-resolved by CRDT merge, that is if each column meets at least one of these two conditions:

1) the values in local and remote tuple are the same (NULL or equal)
2) it uses a CRDT data type (and so can be merged)

Note: This means we also skip the conflict reporting when there are no CRDT columns, but all values in local/remote tuples are equal.

Resetting CRDT Values

Resetting CRDT values is possible but requires special handling. The asynchronous nature of the cluster means that different nodes may see the reset operation (no matter how it's implemented) at different places in the change stream. Different nodes may also initiate a reset concurrently; i.e. before observing the reset from the other node.

In other words, to make the reset operation behave correctly, it needs to be commutative with respect to the regular operations. Many naive ways to reset a value (which may work perfectly well on a single-node) fail for exactly this reason.

For example, the simplest approach to resetting a value might be:

```
UPDATE crdt_table SET cnt = 0 WHERE id = 1;
```

With state-based CRDTs this does not work - it throws away the state for the other nodes, but only locally. It will be added back by merge functions on remote nodes, causing diverging values, and eventually receiving it back due to changes on the other nodes.

With operation-based CRDTs, this may seem to work because the update is interpreted as a subtraction of `-cnt`. But it only works in the absence of concurrent resets. Once two nodes attempt to do a reset at the same time, we'll end up applying the delta twice, getting a negative value (which is not what we expected from a reset).

It might also seem that `DELETE + INSERT` can be used as a reset, but this has a couple of weaknesses too. If the row is reinserted with the same key, it's not guaranteed that all nodes will see it at the same position in the stream of operations (with respect to changes from other nodes). BDR specifically discourages re-using the same Primary Key value since it can lead to data anomalies in concurrent cases.

State-based CRDT types can reliably handle resets, using a special `!` operator like this:

```
UPDATE tab SET counter = !counter WHERE ...;
```

By “reliably” we mean the values do not have the two issues illustrated above - multiple concurrent resets and divergence.

Operation-based CRDT types can only be reset reliably using Eager Replication, since this avoids multiple concurrent resets. Eager Replication can also be used to set either kind of CRDT to a specific value.

Implemented CRDT data types

Currently there are six CRDT data types implemented - grow-only counter and sum, positive-negative counter and sum, and delta counter and sum. The counters and sums behave mostly the same, except that the “counter” types are integer-based (`bigint`), while the “sum” types are decimal-based (`numeric`).

Additional CRDT types, described at [1], may be implemented later.

The currently implemented CRDT data types can be listed with the following query:

```
SELECT n.nspname, t.typname
FROM bdr.crdt_handlers c
JOIN (pg_type t JOIN pg_namespace n ON t.typnamespace = n.oid)
  ON t.oid = c.crdt_type_id;
```

grow-only counter (`crdt_gcounter`)

- supports only increments with non-negative values (`value + int` and `counter + bigint` operators)
- current value of the counter can be obtained either using # operator or by casting it to `bigint`
- is not compatible with simple assignments like `counter = value` (which is common pattern when the new value is computed somewhere in the application)
- allows simple reset of the counter, using the `!` operator (`counter = !counter`)
- internal state can be inspected using `crdt_gcounter_to_text`

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    cnt      bdr.crdt_gcounter NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);      -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129824); -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4531);  -- error: negative value
```

```
-- enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection('crdt_test', 'column_modify_timestamp', 'cts');

-- increment counters
UPDATE crdt_test SET cnt = cnt + 1 WHERE id = 1;
UPDATE crdt_test SET cnt = cnt + 120 WHERE id = 2;

-- error: minus operator not defined
UPDATE crdt_test SET cnt = cnt - 1 WHERE id = 1;

-- error: increment has to be non-negative
UPDATE crdt_test SET cnt = cnt + (-1) WHERE id = 1;

-- reset counter
UPDATE crdt_test SET cnt = !cnt WHERE id = 1;

-- get current counter value
SELECT id, cnt::bigint, cnt FROM crdt_test;

-- show internal structure of counters
SELECT id, bdr.crdt_gcounter_to_text(cnt) FROM crdt_test;
```

grow-only sum (`crdt_gsum`)

- supports only increments with non-negative values (`sum + numeric`)
- current value of the sum can be obtained either by using # operator or by casting it to `numeric`
- is not compatible with simple assignments like `sum = value` (which is the common pattern when the new value is computed somewhere in the application)
- allows simple reset of the sum, using the `!` operator (`sum = !sum`)
- internal state can be inspected using `crdt_gsum_to_text`

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    gsum     bdr.crdt_gsum NOT NULL DEFAULT 0.0
);

INSERT INTO crdt_test VALUES (1, 0.0);      -- initialized to 0
INSERT INTO crdt_test VALUES (2, 1298.24); -- initialized to 1298.24
INSERT INTO crdt_test VALUES (3, -45.31);  -- error: negative value
```

```
-- enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection('crdt_test', 'column_modify_timestamp', 'cts');

-- increment sum
UPDATE crdt_test SET gsum = gsum + 11.5 WHERE id = 1;
UPDATE crdt_test SET gsum = gsum + 120.33 WHERE id = 2;

-- error: minus operator not defined
UPDATE crdt_test SET gsum = gsum - 15.2 WHERE id = 1;

-- error: increment has to be non-negative
UPDATE crdt_test SET gsum = gsum + (-1.56) WHERE id = 1;

-- reset sum
UPDATE crdt_test SET gsum = !gsum WHERE id = 1;

-- get current sum value
SELECT id, gsum::numeric, gsum FROM crdt_test;

-- show internal structure of sums
SELECT id, bdr.crdt_gsum_to_text(gsum) FROM crdt_test;
```

positive-negative counter (`crdt_pncounter`)

- supports increments with both positive and negative values (through `counter + int` and `counter + bigint` operators)
- current value of the counter can be obtained either by using # operator or by casting to `bigint`
- is not compatible with simple assignments like `counter = value` (which is the common pattern when the new value is computed somewhere in the application)
- allows simple reset of the counter, using the `!` operator (`counter = !counter`)
- internal state can be inspected using `crdt_pncounter_to_text`

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    cnt      bdr.crdt_pncounter NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);      -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129824); -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4531);  -- initialized to -4531
```

```
-- enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection('crdt_test', 'column_modify_timestamp', 'cts');

-- increment counters
UPDATE crdt_test SET cnt = cnt + 1      WHERE id = 1;
UPDATE crdt_test SET cnt = cnt + 120    WHERE id = 2;
UPDATE crdt_test SET cnt = cnt + (-244) WHERE id = 3;

-- decrement counters
UPDATE crdt_test SET cnt = cnt - 73    WHERE id = 1;
UPDATE crdt_test SET cnt = cnt - 19283 WHERE id = 2;
UPDATE crdt_test SET cnt = cnt - (-12) WHERE id = 3;

-- get current counter value
SELECT id, cnt::bigint, cnt FROM crdt_test;

-- show internal structure of counters
SELECT id, bdr.crdt_pncounter_to_text(cnt) FROM crdt_test;

-- reset counter
UPDATE crdt_test SET cnt = !cnt WHERE id = 1;

-- get current counter value after the reset
SELECT id, cnt::bigint, cnt FROM crdt_test;
```

positive-negative sum (`crdt_pnsum`)

- supports increments with both positive and negative values (through `sum + numeric`)
- current value of the sum can be obtained either by using # operator or by casting to `numeric`
- is not compatible with simple assignments like `sum = value` (which is the common pattern when the new value is computed somewhere in the application)
- allows simple reset of the sum, using the `!` operator (`sum = !sum`)
- internal state can be inspected using `crdt_pnsum_to_text`

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    pnsum    bdr.crdt_pnsum NOT NULL DEFAULT 0
);
```

```
INSERT INTO crdt_test VALUES (1, 0);       -- initialized to 0
INSERT INTO crdt_test VALUES (2, 1298.24); -- initialized to 1298.24
INSERT INTO crdt_test VALUES (3, -45.31);  -- initialized to -45.31

-- enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection('crdt_test', 'column_modify_timestamp', 'cts');

-- increment sums
UPDATE crdt_test SET pnsum = pnsum + 1.44     WHERE id = 1;
UPDATE crdt_test SET pnsum = pnsum + 12.20    WHERE id = 2;
UPDATE crdt_test SET pnsum = pnsum + (-24.34) WHERE id = 3;

-- decrement sums
UPDATE crdt_test SET pnsum = pnsum - 7.3      WHERE id = 1;
UPDATE crdt_test SET pnsum = pnsum - 192.83   WHERE id = 2;
UPDATE crdt_test SET pnsum = pnsum - (-12.22) WHERE id = 3;

-- get current sum value
SELECT id, pnsum::numeric, pnsum FROM crdt_test;

-- show internal structure of sum
SELECT id, bdr.crdt_pnsum_to_text(pnsum) FROM crdt_test;

-- reset sum
UPDATE crdt_test SET pnsum = !pnsum WHERE id = 1;

-- get current sum value after the reset
SELECT id, pnsum::numeric, pnsum FROM crdt_test;
```

delta counter (`crdt_delta_counter`)

- is defined a `bigint` domain, so works exactly like a `bigint` column
- supports increments with both positive and negative values
- is compatible with simple assignments like `counter = value` (common when the new value is computed somewhere in the application)
- no simple way to reset the value (reliably)

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    cnt      bdr.crdt_delta_counter NOT NULL DEFAULT 0
);
```

```
INSERT INTO crdt_test VALUES (1, 0);      -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129824); -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4531);  -- initialized to -4531

-- enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection('crdt_test', 'column_modify_timestamp', 'cts');

-- increment counters
UPDATE crdt_test SET cnt = cnt + 1      WHERE id = 1;
UPDATE crdt_test SET cnt = cnt + 120    WHERE id = 2;
UPDATE crdt_test SET cnt = cnt + (-244) WHERE id = 3;

-- decrement counters
UPDATE crdt_test SET cnt = cnt - 73    WHERE id = 1;
UPDATE crdt_test SET cnt = cnt - 19283 WHERE id = 2;
UPDATE crdt_test SET cnt = cnt - (-12) WHERE id = 3;

-- get current counter value
SELECT id, cnt FROM crdt_test;
```

delta sum (`crdt_delta_sum`)

- is defined as a `numeric` domain, so works exactly like a `numeric` column
- supports increments with both positive and negative values
- is compatible with simple assignments like `sum = value` (common when the new value is computed somewhere in the application)
- no simple way to reset the value (reliably)

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    dsum     bdr.crdt_delta_sum NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);       -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129.824); -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4.531);  -- initialized to -4531

-- enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection('crdt_test', 'column_modify_timestamp', 'cts');
```

```
-- increment counters
UPDATE crdt_test SET dsum = dsum + 1.32   WHERE id = 1;
UPDATE crdt_test SET dsum = dsum + 12.01  WHERE id = 2;
UPDATE crdt_test SET dsum = dsum + (-2.4) WHERE id = 3;

-- decrement counters
UPDATE crdt_test SET dsum = dsum - 7.33   WHERE id = 1;
UPDATE crdt_test SET dsum = dsum - 19.83  WHERE id = 2;
UPDATE crdt_test SET dsum = dsum - (-1.2) WHERE id = 3;

-- get current counter value
SELECT id, cnt FROM crdt_test;
```

[1] https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

Durability & Performance Options

Overview

Synchronous or *Eager Replication* synchronizes between at least two nodes of the cluster before committing a transaction. This provides three properties of interest to applications, which are related, but can all be implemented individually:

- *Durability*: writing to multiple nodes increases crash resilience and allows the data to be recovered after a crash and restart.
- *Visibility*: with the commit confirmation to the client, the database guarantees immediate visibility of the committed transaction on some sets of nodes.
- *No Conflicts After Commit*: the client can rely on the transaction to eventually be applied on all nodes without further conflicts, or get an abort directly informing the client of an error.

PGLogical (PGL) integrates with the `synchronous_commit` option of Postgres itself, providing a variant of synchronous replication, which can be used between BDR nodes. In addition, BDR offers Eager All-Node Replication and Commit At Most Once in the Enterprise Edition.

Postgres itself provides Physical Streaming Replication (PSR), which is uni-directional, but offers a synchronous variant that can used in combination with BDR.

This chapter covers the various forms of synchronous or eager replication and its timing aspects.

Comparison

Most options for synchronous replication available to BDR allow for different levels of synchronization, offering different trade-offs between performance and protection against node or network outages.

The following table summarizes what a client can expect from a peer node replicated to after having received a COMMIT confirmation from the origin node the transaction was issued to.

Variant	Mode	Received	Visible	Durable
PGL/BDR	off (default)	no	no	no
PGL/BDR	remote_write (2)	yes	no	no
PGL/BDR	on (2)	yes	yes	yes
PGL/BDR	remote_apply (2)	yes	yes	yes
PSR	remote_write (2)	yes	no	no (1)
PSR	on (2)	yes	no	yes
PSR	remote_apply (2)	yes	yes	yes
CAMO	remote_write (2)	yes	no	no
CAMO	remote_commit_async (2)	yes	yes	no

Variant	Mode	Received	Visible	Durable
CAMO	remote_commit_flush (2)	yes	yes	yes
Eager	n/a	yes	yes	yes

(1) written to the OS, durable if the OS remains running and only Postgres crashes.

(2) unless switched to Local mode (if allowed) by setting `synchronous_replication_availability` to `async`, otherwise the values for the asynchronous BDR default apply.

Reception ensures the peer will be able to eventually apply all changes of the transaction without requiring any further communication, i.e. even in the face of a full or partial network outage. All modes considered synchronous provide this protection.

Visibility implies the transaction was applied remotely, and any possible conflicts with concurrent transactions have been resolved. Without durability, i.e. prior to persisting the transaction, a crash of the peer node may revert this state (and require re-transmission and re-application of the changes).

Durability relates to the peer node's storage and provides protection against loss of data after a crash and recovery of the peer node. If the transaction has already been visible before the crash, it will be recovered to be visible, again. Otherwise, the transaction's payload is persisted and the peer node will be able to apply the transaction eventually (without requiring any re-transmission of data).

Internal Timing of Operations

For a better understanding of how the different modes work, it is helpful to realize PSR and PGLogical apply transactions rather differently.

With physical streaming replication, the order of operations is:

- origin flushes a commit record to WAL, making the transaction visible locally
- peer node receives changes and issues a write
- peer flushes the received changes to disk
- peer applies changes, making the transaction visible locally

With PGLogical, the order of operations is different:

- origin flushes a commit record to WAL, making the transaction visible locally
- peer node receives changes into its apply queue in memory
- peer applies changes, making the transaction visible locally
- peer persists the transaction by flushing to disk

For CAMO and Eager All Node Replication, note that the origin node waits for a confirmation prior to making the transaction visible locally. The order of operations is:

- origin flushes a prepare or pre-commit record to WAL

- peer node receives changes into its apply queue in memory
- peer applies changes, making the transaction visible locally
- peer persists the transaction by flushing to disk
- origin commits and makes the transaction visible locally

The following table summarizes the differences.

Variant	Order of apply vs persist on peer nodes	Replication before or after origin WAL commit record write
PSR	persist first	after
PGL	apply first	after
CAMO	apply first	before (triggered by pre-commit)
Eager	apply first	before (triggered by prepare)

Configuration

The following table provides an overview of which configuration settings are required to be set to a non-default value (req) or optional (opt), but affecting a specific variant.

setting (GUC)	PSR	PGL	CAMO	Eager
synchronous_standby_names	req	req	n/a	n/a
synchronous_commit	opt	opt	n/a	n/a
synchronous_replication_availability	opt	opt	opt	n/a
pg2q.enable_camo	n/a	n/a	req	n/a
bdr.camo_origin_for	n/a	n/a	req	n/a
bdr.camo_partner_of (on partner node)	n/a	n/a	req	n/a
bdr.commit_scope	n/a	n/a	n/a	req
bdr.global_commit_timeout	n/a	n/a	opt	opt

Planned Shutdown and Restarts

When using PGL or CAMO in combination with `remote_write`, care must be taken with planned shutdown or restart. By default, the apply queue is consumed prior to shutting down. However, in the `immediate` shutdown mode, the queue is discarded at shutdown, leading to the stopped node "forgetting" transactions in the queue. A concurrent failure of another node could lead to loss of data, as if both nodes failed.

To ensure the apply queue gets flushed to disk, please use either `smart` or `fast` shutdown for maintenance tasks. This maintains the required synchronization level and prevents loss of data.

Synchronous Replication using PGLogical

Usage

To enable synchronous replication using PGLogical, the application name of the relevant BDR peer nodes need to be added to `synchronous_standby_names`. The use of `FIRST x` or `ANY x` offers a lot of flexibility, if this does not conflict with the requirements of non-BDR standby nodes.

Once added, the level of synchronization can be configured per transaction via `synchronous_commit`, which defaults to `on` - meaning that adding to `synchronous_standby_names` already enables synchronous replication. Setting `synchronous_commit` to `local` or `off` turns off synchronous replication.

Due to PGLogical applying the transaction before persisting it, the values `on` and `remote_apply` are equivalent (for logical replication).

Limitations

PGLogical uses the same configuration (and internal mechanisms) as Physical Streaming Replication, therefore the needs for (physical, non-BDR) standbys needs to be considered when configuring synchronous replication between BDR nodes using PGLogical. Most importantly, it is not possible to use different synchronization modes for a single transaction.

Eager All-Node Replication

To prevent conflicts after a commit, set the `bdr.commit_scope` parameter to `global`. The default setting of `local` disables eager replication, so BDR will apply changes and resolve potential conflicts post-commit, as described in the Conflicts chapter.

In this mode, BDR uses two-phase commit (2PC) internally to detect and resolve conflicts prior to the local commit. It turns a normal `COMMIT` of the application into an implicit two-phase commit, requiring all peer nodes to prepare the transaction, before the origin node continues to commit it. If at least one node is down or unreachable during the prepare phase, the commit will time out after `bdr.global_commit_timeout`, leading to an abort of the transaction. Note that there is no restriction on the use of temporary tables, as exists in explicit 2PC in PostgreSQL.

Once prepared, Eager All-Node Replication employs Raft to reach a commit decision. In case of failures, this allows a remaining majority of nodes to reach a congruent commit or abort decision so they can finish the transaction. This unblocks the objects and resources locked by the transaction and allows the cluster to proceed.

In case all nodes remain operational, the origin will confirm the commit to the client only after all nodes have committed, to ensure that the transaction is immediately visible on all nodes after the commit.

Requirements

Eager All-Node Replication uses prepared transactions internally; therefore all replica nodes need to have a `max_prepared_transactions` configured high enough to be able to handle all incoming transactions (possibly in addition to local two-phase commit and CAMO transactions; see Configuration: Max Prepared Transactions). We recommend to configure it the same on all nodes, and high enough to cover the maximum number of concurrent transactions across the cluster for which CAMO or Eager All-Node Replication is used. Other than that, no special configuration is required, and every BDR cluster can run Eager All-Node transactions.

Usage

To enable Eager All-Node Replication, the client needs to switch to global commit scope at session level, or for individual transactions as shown here:

```
BEGIN;

... other commands possible...

SET LOCAL bdr.commit_scope = 'global';

... other commands possible...
```

The client can continue to simply issue a `COMMIT` at the end of the transaction and let BDR manage the two phases:

```
COMMIT;
```

Error handling

Given that BDR manages the transaction, the client only needs to check the result of the `COMMIT` (as is advisable in any case, including single-node Postgres).

In case of an origin node failure, the remaining nodes will eventually (after at least `bdr.global_commit_timeout`) decide to rollback the globally prepared transaction. Raft prevents inconsistent commit vs. rollback decisions. This, however, requires a majority of connected nodes. Disconnected nodes keep the transactions prepared to be able to eventually commit them (or rollback) as needed to reconcile with the majority of nodes that may have decided and made further progress.

Eager All-Node Replication with CAMO

Eager All-Node Replication goes beyond CAMO and implies it; there is no need to additionally enable `pg2q.enable_camo`, if `bdr.commit_scope` is set to `global`. Nor does a CAMO partner or origin need to be configured via `bdr.camo_partner_of` or `bdr.camo_origin_for`.

Any other active BDR node may be used in the role of a CAMO partner to query a transaction's status'. However, this non-CAMO usage needs to be indicated to the `bdr.logical_transaction_status` function with a third argument of `require_camo_partner = false`. Otherwise, it may complain about a missing CAMO configuration (which is not required for Eager transactions).

Other than this difference in configuration and invocation of that function, the client needs to adhere to the protocol described for CAMO. See the reference client implementations.

Limitations

Transactions using eager replication cannot yet execute DDL, nor do they support explicit two-phase commit. These may be allowed in later releases. Note that the TRUNCATE command is allowed.

Replacing a crashed and unrecoverable BDR node with its physical standby is not currently supported in combination with Eager All Node transactions.

BDR currently offers a global commit scope only; later releases will support Eager replication with fewer nodes for increased availability.

It is not possible for Eager All Node replication to be combined with `synchronous_replication_availability = 'asy`
Trying to configure both will yield an error.

Effects of Eager Replication in General

Increased Commit Latency

Adding a synchronization step means additional communication between the nodes, resulting in additional latency at commit time. Eager All Node Replication adds roughly two network round trips (to the furthest peer node in the worst case). Logical standby nodes and nodes still in the process of joining or catching up are not included, but will eventually receive changes.

Before a peer node can confirm its local preparation of the transaction, it also needs to apply it locally. This further adds to the commit latency, depending on the size of the transaction. Note that this is independent of the `synchronous_commit` setting and applies whenever `bdr.commit_scope` is set to `global`.

> **Note**
> The performance of Eager replication is currently known to be unexpectedly slow (around 10 TPS only). This is expected to be improved in the next release.

Increased Abort Rate

With single-node Postgres, or even with BDR in its default asynchronous replication mode, errors at `COMMIT` time are rare. The additional synchronization step adds a source of errors, so applications need to be prepared to properly handle such errors (usually by applying a retry loop).

The rate of aborts depends solely on the workload. Large transactions changing many rows are much more likely to conflict with other concurrent transactions.

Commit at Most Once (CAMO)

The objective of the Commit at Most Once (CAMO) feature is to prevent the application from committing more than once.

Without CAMO, when a client loses connection after COMMIT has been submitted, the application might not receive a reply from the server and will therefore be unsure whether the transaction committed or not.

The application cannot easily decide between the two options of:

1) retrying the transaction with the same data, since this can in some cases cause the data to be entered twice, or
2) not retrying the transaction, and risk that the data doesn't get processed at all.

Either of those is a critical error with high value data.

There are two ways to avoid this situation:

One way to avoid this situation is to make sure that the transaction includes at least one `INSERT` into a table with a unique index, but that is dependent upon the application design and requires application-specific error-handling logic, so is not effective in all cases. This is the only option if using BDR-SE.

The CAMO feature in BDR-EE offers a more general solution and does not require an `INSERT` as described above. When activated via `pg2q.enable_camo` or `bdr.commit_scope`, the application will receive a message containing the transaction identifier, if already assigned. Otherwise, the first write statement in a transaction will send that information to the client. If the application sends an explicit COMMIT, the protocol will ensure that the application will have received the notification of the transaction identifier before the COMMIT is sent. If the server does not reply to the COMMIT, the application can handle this error by using the transaction identifier to request the final status of the transaction from another BDR node. If the prior transaction status is known, then the application can safely decide whether or not to retry the transaction.

CAMO works in one of two modes:

- Pair mode
- In combination with Eager All Node Replication

In the Pair mode, CAMO works by assigning to each CAMO node ("origin") a designated **CAMO partner**, which is another BDR master node from the same BDR group. In this operation mode, only the CAMO partner node knows the outcome of any recent transaction that was disconnected during COMMIT on the CAMO origin.

When combined with Eager All Node Replication, CAMO enables every peer (that is a full BDR master node) to act as a CAMO partner. No designated CAMO partner needs to be configured in this mode.

> **Warning**
> CAMO requires changes to the user's application to take advantage of the advanced error handling: it is not sufficient to enable a parameter to gain protection. Reference client implementations are provided in Appendix F.

Requirements

To utilize CAMO, an application must issue an explicit COMMIT message, issued as a separate request (not as part of a multi-statement request). CAMO cannot provide status for transactions issued from within procedures, or from single-statement transactions that use implicit commits.

Configuration

Assuming an existing BDR cluster consisting of the two nodes `node1` and `node2`, both with a BDR enabled database called `bdrdemo`, the following steps will configure `node2` as a CAMO partner for `node1`:

1) Create the BDR cluster including nodes `node1` and `node2`.
2) Ensure the following settings on `node1` (according to 2ndQPostgres Settings for BDR):

   ```
   bdr.camo_origin_for = 'bdrdemo:node2'
   pg2q.enable_camo = 'remote_commit_flush'
   ```

3) Ensure the following setting on `node2`:

   ```
   bdr.camo_partner_of = 'bdrdemo:node1'
   ```

4) Restart PostgreSQL on both nodes for all of these changes to take effect.
5) Adjust the application to use the COMMIT error handling that CAMO suggests.

We do not recommend enabling CAMO at server level, as this imposes higher latency for all transactions, even when not needed. Instead, we recommend to selectively enable it just for individual transactions by turning on CAMO at session or transaction level.

To enable at session level, issue:

```
SET pg2q.enable_camo = 'remote_commit_flush';
```

. . . or to enable for individual transactions, issue this after starting the transaction and before committing it:

```
SET LOCAL pg2q.enable_camo = 'remote_commit_flush';
```

Valid values for `pg2q.enable_camo` that enable CAMO are

- `remote_write`
- `remote_commit_async`
- `remote_commit_flush`

See the Comparison of synchronous replication modes for details about how each mode behaves. Setting `pg2q.enable_camo = off` disables this feature, which is the default.

CAMO with Eager All Node Replication

To use CAMO with Eager All Node Replication, no changes are required on either node. It is sufficient to enable the global commit scope after the start of the transaction - you do not need to set `pg2q.enable_camo` at all.

```
BEGIN;
SET LOCAL bdr.commit_scope = 'global';
...
COMMIT;
```

The application still needs to be adjusted to use COMMIT error handling as specified, but is free to connect to any available BDR node to query the transaction's status.

Failure Scenarios

In this section, we analyze failure scenarios for different configurations. After comparing Local mode with CAMO mode in terms of Availability versus Consistency, we also provide three specific examples.

Data persistence at receiver side

By default, a PGL writer operates in `pglogical.synchronous_commit = off` mode when applying transactions from remote nodes. This holds true for CAMO as well, meaning that transactions are confirmed to the origin node possibly before reaching the disk of the CAMO partner. In case of a crash or hardware failure, it is possible for a confirmed transaction to not be recoverable on the CAMO partner by itself. This is not an issue as long as the CAMO origin node remains operational, as it will redistribute the transaction once the CAMO partner node recovers.

This in turn means CAMO can protect against a single node failure, which is correct for Local mode as well as (or even in combination with) Remote Write.

To cover an outage of both nodes of a CAMO pair, it is possible to use `pglogical.synchronous_commit = local` to enforce a flush prior to the pre-commit confirmation. This does not work in combination with either Remote Write nor Local mode, and has an additional performance impact due to additional I/O requirements on the CAMO partner in the latency sensitive commit path.

Local Mode

When `synchronous_replication_availability = 'async'`, an origin (i.e. master) node will detect whether its CAMO partner is ready; if not, it will temporarily switch to **Local** mode. When in Local mode, an origin commits transactions locally, until switching back to CAMO mode.

This clearly does not allow COMMIT status to be retrieved, but does provide the option to choose availability over consistency. This mode can tolerate a single node failure. In case both nodes of a CAMO pair fail, they may choose incongruent commit decisions to maintain availability, leading to data inconsistencies.

For a CAMO partner to switch to ready, it needs to be connected, and the estimated catchup interval needs to drop below `bdr.global_commit_timeout`. It switches when losing the connection or in case the catchup interval raises above the `bdr.global_commit_timeout`. The current readiness status of a CAMO partner can be checked with `bdr.is_camo_partner_ready`, while `bdr.node_estimates` provides the current estimate of the catchup time.

The detection on the sending node can be configured via PostgreSQL settings controlling keep-alives and timeouts on the TCP connection to the CAMO partner. The `wal_sender_timeout` is the amount of time that an origin waits for a CAMO partner until switching to Local mode. Additionally, the `bdr.global_commit_timeout` setting puts a per-transaction limit on the maximum delay a COMMIT can incur due to the CAMO partner being unreachable. It may well be lower than the `wal_sender_timeout`, which influences synchronous standbys as well, and for which a good compromise between responsiveness and stability needs to be found.

The switch from Local mode to CAMO mode depends on the CAMO partner node, which initiates the connection. The CAMO partner tries to re-connect at least every 30 seconds. After connectivity is reestablished, it may therefore take up to 30 seconds until the CAMO partner connects back to its origin node. Any lag that accumulated on the CAMO partner will further delay the switch back to CAMO protected mode.

Unlike during normal CAMO operation, in Local mode there is no additional commit overhead. This can be problematic, as it allows the origin node to continuously process more transactions than the CAMO pair could normally process. Even if the CAMO partner eventually reconnects and applies transactions, its lag will only ever increase in such a situation, preventing re-establishing the CAMO protection. To artificially throttle transactional throughput, BDR provides the `bdr.camo_local_mode_delay` setting, allowing to delay COMMITs in Local mode by an arbitrary amount of time. We recommend to measure commit times in normal CAMO mode during expected workloads and configure this delay accordingly. The default is 5 ms, which reflects a local network and a relatively quick CAMO partner response.

The choice of whether to allow Local mode should be taken in view of the architecture and the availability requirements. We expand this point by discussing three specific examples in some detail.

Example: Symmetric Node Pair

In this section we consider a setup with two BDR nodes that are the CAMO partner of each other.

This configuration enables CAMO behavior on both nodes; it is therefore suitable for workload patterns where it is acceptable to write concurrently on more than one node, e.g. in cases that are not likely to generate conflicts.

With Local Mode

If Local mode is allowed, there is no single point of failure, and when one node fails:

- The other node can determine the status of all transactions that were disconnected during COMMIT on the failed node.
- New write transactions are allowed:
  - If the second node also fails, then the outcome of those transactions that were being committed at that time will be unknown.

Without Local Mode

If Local mode is not allowed, then each node requires the other node for committing transactions, i.e. each node is a single point of failure. Precisely, when one node fails:

- The other node can determine the status of all transactions that were disconnected during COMMIT on the failed node.
- New write transactions will be prevented until the node recovers.

Example: Origin-Partner Pair

Here we consider two BDR nodes: one CAMO origin, and its CAMO partner.

This configuration enables CAMO only on one node, and is therefore appropriate for those cases where it is recommended to write only on one “main” node, and the other node is intended to be used as a backup in case of failure of the main node.

For instance, this can be the case if the workload pattern causes a relevant amount of conflicts, or if conflicts are unacceptable altogether.

With Local Mode

When Local mode is allowed, the CAMO partner is a single point of failure only in terms of CAMO, and not for writes in general.

More precisely, when the CAMO partner fails:

- The CAMO origin cannot determine the status of any transaction that was disconnected during COMMIT on the CAMO partner.
- The CAMO origin can still commit new transactions:

- If the CAMO origin also fails, then the outcome of those transactions that were being committed at that time will be unknown.

When the CAMO origin fails:

- The CAMO partner can determine the status of all transactions that were disconnected during COMMIT on the CAMO origin.
- The CAMO partner can commit new transactions.
- If the CAMO partner also fails, then the outcome of those transactions that were being committed at that time will be unknown.

Note that the single-node failure scenario is almost the same irrespective of which node fails; the only difference is the ability to determine the outcome of prior transactions that were being committed at the time of the failure.

This symmetry is not surprising, because the effect of Local mode is to suspend CAMO transaction status caching while the CAMO partner node is unavailable.

Without Local Mode

If Local mode is not allowed, then the CAMO partner is a single point of failure, as it is required for the CAMO origin to commit.

When the CAMO partner fails:

- The CAMO origin cannot determine the status of any transaction that was disconnected during COMMIT on the CAMO partner.
- The CAMO origin cannot commit new transactions.

When the CAMO origin fails:

- The CAMO partner can determine the status of all transactions that were disconnected during COMMIT on the CAMO origin.
- The CAMO partner can commit new transactions:
- If the CAMO partner also fails, then the outcome of those transactions that were being committed at that time will be unknown.

Example: Three CAMO Nodes

Here we consider three BDR nodes whose CAMO dependencies are circular:

- A: the CAMO origin for B, and the CAMO partner for C
- B: the CAMO origin for C, and the CAMO partner for A

- C: the CAMO origin for A, and the CAMO partner for B

This arrangement is able to provide High Availability together with CAMO information even when one node fails, thanks to the additional BDR node.

During normal operation, all three nodes can be used for concurrent writes, provided that the workload is appropriate, for instance if conflicts are impossible, or likely to occur only with acceptable frequency.

In case of a single node failure, there remains at least one fully operational CAMO pair, leaving exactly one node that can still apply transactions with CAMO protection.

Local Mode Not Required

It is not necessary to consider or enable Local mode, because the configuration already provides High Availability together with full CAMO functionality.

Without Local Mode

If Node A fails while it is being written:

- Node B can determine the status of all transactions that were disconnected during COMMIT on Node A.
- Node B can commit new transactions:
  - If Node B also fails, the outcome of those transactions that were being committed at that time can be determined by Node C.
- Node C is unable to commit new transactions until Node A is again available.

In particular, the applications that were writing to Node A can perform a fast failover to Node B and access CAMO transaction status.

We do not need to analyze separately the failures of the other two nodes, because this configuration is symmetric.

Application Usage

Overview and Requirements

Commit At Most Once relies on a retry loop and specific error handling on the client side. There are three aspects to it:

- The result of a transaction's COMMIT needs to be checked, and in case of a temporary error, the client must retry the transaction.
- Prior to COMMIT, the client needs to retrieve a global identifier for the transaction, consisting of a **node id** and a **transaction id** (both 32 bit integers).

- Should the current server fail while attempting COMMIT of a transaction, the application must connect to its CAMO partner, retrieve the status of that transaction, and retry depending on the response.

Note that the application needs to store the global transaction identifier only for the purpose of verifying the transaction status in case of disconnection during COMMIT. In particular, the application does not need any additional persistence layer: if the application fails, it only needs the information in the database to restart.

CAMO partner connection status

The function `bdr.is_camo_partner_connected` allows checking the connection status of a CAMO partner node.

Synopsis

```
bdr.is_camo_partner_connected()
```

Return value

A boolean value indicating whether the CAMO partner is currently connected to a walsender process on the local node and therefore able to receive transactional data and send back confirmations.

CAMO partner readiness

The function `bdr.is_camo_partner_ready` allows checking the readiness status of a CAMO partner node, as used for the switch to and from local mode:

Synopsis

```
bdr.is_camo_partner_ready()
```

Return value

A boolean value indicating whether the CAMO partner can reasonably be expected to confirm transactions originating from the local node.

Wait for consumption of the apply queue from the CAMO partner

The function `bdr.wait_for_camo_partner_queue` is a wrapper of the above function defaulting to the CAMO partner node. Unlike the above, if CAMO is not configured, the function yields an error.

Synopsis

```
bdr.wait_for_camo_partner_queue()
```

Transaction status query function

The application should use the function:

```
bdr.logical_transaction_status(node_id, xid, require_camo_partner)
```

. . . to check the status of a transaction which was being committed when the node failed.

In CAMO mode, this function must only ever be used on one of the two nodes of a CAMO pair. It will raise an error on any other node. When using Eager transactions, any other BDR node may be queried with this function, but it then needs a `require_camo_partner := false` argument.

In both cases, the function needs to be called within 15 minutes after the commit was issued, as the CAMO partner needs to regularly purge such meta-information and therefore cannot provide correct answers for older transactions. The client must not call this function before attempting to commit on the origin.

Prior to querying the status of a transaction, this function waits for the receive queue to be consumed and fully applied. This prevents early negative answers for transactions that have already been received, but not applied, yet.

Note that despite its name, it is not always a read-only operation. If the status is unknown, the CAMO partner will decide whether to commit or abort the transaction, storing that decision locally to ensure consistency going forward.

Synopsis

```
bdr.logical_transaction_status(node_id OID,
                               xid     OID,
                               require_camo_partner BOOL DEFAULT true)
```

Parameters

- `node_id` - the node id of the BDR node the transaction originates from, usually retrieved by the client before COMMIT from the PQ parameter `bdr.local_node_id`.
- `xid` - the transaction id on the origin node, usually retrieved by the client before COMMIT from the PQ parameter `transaction_id` (requires `enable_camo` to be set to `remote_write`, `remote_commit_async`, or `remote_commit_flush`. See 2ndQPostgres Settings for BDR)
- `require_camo_partner` - defaults to true and enables configuration checks; may be set to false to disable these checks and query the status of a transaction that was protected by Eager All Node Replication.

Return value

The function will return one of these results:

- `'committed'::TEXT` - the transaction has been committed and will be replicated to all other BDR nodes. No need for the application to retry it.
- `'aborted'::TEXT` - the transaction has been aborted and will not be replicated to any other BDR node. The application needs to either retry it or escalate the failure to commit the transaction.
- `'in progress'::TEXT` - the transaction is still in progress on this local node and has neither been committed nor aborted, yet. Note that the transaction may well be in the COMMIT phase, waiting for the CAMO partner to confirm or deny the commit. The recommended client reaction is to disconnect from the origin node and reconnect to the CAMO partner to query that instead. See the `isTransactionCommitted` method of the reference clients. With a load balancer or proxy in between, where the client lacks control over which node gets queried, the client can only poll repeatedly until the status switches to either `'committed'` or `'aborted'`.

For Eager All Node Replication, peer nodes yield this result for transactions that are not yet committed or aborted. This means that even transactions not yet replicated (or not even started on the origin node) may yield an `in progress` result on a peer BDR node in this case. However, the client should not query the transaction status prior to attempting to commit on the origin.

- `'unknown'::TEXT` - the transaction specified is unknown, either because it is in the future, not replicated to that specific node yet, or too far in the past. The status of such a transaction is not yet or no longer known. This return value is a sign of improper use by the application.

The client must be prepared to retry the function call on error.

Connection pools and proxies

The effect of connection pools and proxies needs to be considered when designing a CAMO cluster. In symmetric cases as well as for Eager All Node Replication, a proxy may freely distribute the load to either of the nodes. Otherwise, the proxy should use the node designated for normal transaction processing, and only fail over to the CAMO partner if the CAMO origin fails.

In both cases, care needs to be taken to ensure that the application fetches the proper node id: when using session pooling, the client remains connected to the same node, so the node id remains constant for the lifetime of the client session. However, with finer-grained transaction pooling, the client needs to fetch the node id for every transaction (as in the example given below).

A client that is not directly connected to the BDR nodes might not even notice a failover or switchover, but can always use the `bdr.local_node_id` parameter to determine which node it is currently connected to. In the crucial situation of a disconnect during COMMIT, the proxy must properly forward that disconnect to the client applying the CAMO procedure.

For CAMO in `remote_write` mode, a proxy that potentially switches between the CAMO pairs must use the `bdr.wait_for_camo_partner_queue` function to prevent stale reads.

The proxies supported by 2ndQuadrant are PgBouncer and HAProxy. Note that neither supports CAMO `remote_write` yet.

Example

The following example demonstrates what a retry loop of a CAMO aware client application should look like in C-like pseudo-code. It expects two DSNs `origin_dsn` and `partner_dsn` providing connection information. These usually are the same DSNs as used for the initial call to `bdr.create_node`, and can be looked up in `bdr.node_summary`, column `interface_connstr`.

```
PGconn *conn = PQconnectdb(origin_dsn);

loop {
    // start a transaction
    PQexec(conn, "BEGIN");

    // apply transactional changes
    PQexec(conn, "INSERT INTO ...");
    ...

    // store a globally unique transaction identifier
    node_id = PQparameterStatus(conn, "bdr.local_node_id");
    xid = PQparameterStatus(conn, "transaction_id");

    // attempt to commit
    PQexec(conn, "COMMIT");
    if (PQresultStatus(res) == PGRES_COMMAND_OK)
        return SUCCESS;
    else if (PQstatus(res) == CONNECTION_BAD)
    {
        // Re-connect to the partner
        conn = PQconnectdb(partner_dsn);
        // Check if successfully reconnected
        if (!connectionEstablished())
            panic();

        // Check the attempted transaction's status
        sql = "SELECT bdr.logical_transaction_status($node_id, $xid)";
        txn_status = PQexec(conn, sql);
        if (txn_status == "committed")
            return SUCCESS;
        else
            continue;   // to retry the transaction on the partner
```

```
    }
    else
    {
        // The connection is intact, but the transaction failed for some
        // other reason.  Differentiate between permanent and temporary
        // errors.
        if (isPermanentError())
            return FAILURE;
        else
        {
            // Determine an appropriate delay to back-off to account for
            // temporary failures due to congestion, so as to decrease
            // the overall load put on the servers.
            sleep(increasing_retry_delay);

            continue;
        }
    }
}
```

This example needs to be extended with proper logic for connecting, including retries and error handling. If using a load balancer (e.g. PgBouncer), re-connecting can be implemented by simply using `PQreset`. Ensure that the load balancer only ever redirects a client to a CAMO partner and not any other BDR node.

In practice, an upper limit of retries is recommended. Depending on the actions performed in the transaction, other temporary errors may be possible and need to be handled by retrying the transaction depending on the error code, similarly to the best practices on deadlocks or on serialization failures while in `SERIALIZABLE` isolation mode.

Please see the reference client implementations provided as part of this documentation.

Interaction with DDL and global locks

Transactions protected by CAMO may contain DDL operations. Note however that DDL uses global locks, which already provide some synchronization among nodes; see DDL Locking Details for more information.

Combining CAMO with DDL not only imposes a higher latency, but also increases the chance of global deadlocks. We therefore recommend using a relatively low `bdr.global_lock_timeout`, which aborts the DDL and therefore resolves a deadlock in a reasonable amount of time.

Non-transactional DDL

The following DDL operations are not allowed within a transaction block and therefore cannot possibly benefit from CAMO protection. For these, CAMO is automatically disabled internally:

- all concurrent index operations (`CREATE`, `DROP`, and `REINDEX`)
- `REINDEX DATABASE`, `REINDEX SCHEMA`, and `REINDEX SYSTEM`
- `VACUUM`
- `CLUSTER` without any parameter
- `ALTER TYPE [enum] ADD VALUE`
- `ALTER SYSTEM`
- `CREATE` and `DROP DATABASE`
- `CREATE` and `DROP TABLESPACE`
- `ALTER DATABASE [db] TABLESPACE`

CAMO Limitations

CAMO is designed to query the results of a recently failed COMMIT on the original node, so in case of disconnection, the application should be coded to immediately request the xid status from the CAMO partner. There should be as little delay as possible after the failure before requesting the status. Applications should not rely on CAMO decisions being stored for extended periods of time.

If the application forgets the xid assigned, for example as a result of an app server restart, there is no easy way to recover that. CAMO partner nodes store only the most recent transactions' states; once the cache is cleared, deriving the xid status would be a manual operation and would be dependent on the backup/archiving mechanisms in use. Applications should shut down only when they have resolved any outstanding transaction status.

For the client to apply proper checks, a transaction protected by CAMO cannot be a single statement with implicit transaction control. Nor is it possible to use CAMO with a transaction-controlling procedure or within a `DO` block that tries to start or end transactions.

A change of `bdr.camo_partner_of` or `bdr.camo_origin_for` to change the CAMO pairing currently requires a restart of the node. A reload of the configuration is not sufficient for CAMO to pick up the change.

CAMO resolves commit status but does not yet resolve pending notifications on commit. CAMO and Eager replication options do not allow the `NOTIFY` SQL command or the `pg_notify()` function, nor do they allow `LISTEN` or `UNLISTEN`.

Replacing a crashed and unrecoverable BDR node with its physical standby is not currently supported in combination with CAMO-protected transactions.

When replaying changes, CAMO transactions may detect conflicts just the same as other transactions. If timestamp conflict detection is used, the CAMO transaction uses the timestamp of the prepare on the origin node.

Performance Implications

CAMO extends the normal synchronous replication protocol by adding an additional message round trip at commit. Applications should expect additional latency, but with enough sessions there should be no

loss of throughput. The additional latency is mostly determined by the round trip time between the origin and its partner node.

The CAMO partner confirming transactions needs to store transaction states. Again, compared to non-CAMO operation, this might require an additional seek for each transaction applied from the origin.

Client Application Testing

Proper use of CAMO on the client side is not trivial; we strongly recommend testing the application behavior in combination with the BDR cluster against failure scenarios such as node crashes or network outages.

Timestamp-Based Snapshots

The Timestamp-Based Snapshots feature of 2ndQPostgres allows reading data in a consistent manner via a user-specified timestamp rather than the usual MVCC snapshot. This can be used to access data on different BDR nodes at a common point-in-time; for example, as a way to compare data on multiple nodes for data quality checking. At this time, this feature does not work with write transactions.

The use of timestamp-based snapshots are enabled via the `snapshot_timestamp` parameter; this accepts either a timestamp value or a special value, 'current', which represents the current timestamp (now). If `snapshot_timestamp` is set, queries will use that timestamp to determine visibility of rows, rather than the usual MVCC semantics.

For example, the following query will return state of the `customers` table at 2018-12-08 02:28:30 GMT:

```
SET snapshot_timestamp = '2018-12-08 02:28:30 GMT';
SELECT count(*) FROM customers;
```

In plain 2ndQPostgres, this only works with future timestamps or the above mentioned special 'current' value, so it cannot be used for historical queries (though that is on the longer-term roadmap).

BDR works with and improves on that feature in a multi-node environment. Firstly, BDR will make sure that all connections to other nodes replicated any outstanding data that were added to the database before the specified timestamp, so that the timestamp-based snapshot is consistent across the whole multi-master group. Secondly, BDR adds an additional parameter called `bdr.timestamp_snapshot_keep`. This specifies a window of time during which queries can be executed against the recent history on that node.

You can specify any interval, but be aware that VACUUM (including autovacuum) will not clean dead rows that are newer than up to twice the specified interval. This also means that transaction ids will not be freed for the same amount of time. As a result, using this can leave more bloat in user tables. Initially, we recommend 10 seconds as a typical setting, though you may wish to change that as needed.

Note that once the query has been accepted for execution, the query may run for longer than `bdr.timestamp_snapshot_keep` without problem, just as normal.

Also please note that info about how far the snapshots were kept does not survive server restart, so the oldest usable timestamp for the timestamp-based snapshot is the time of last restart of the PostgreSQL instance.

One can combine the use of `bdr.timestamp_snapshot_keep` with the `postgres_fdw` extension to get a consistent read across multiple nodes in a BDR group. This can be used to run parallel queries across nodes, when used in conjunction with foreign tables.

There are no limits on the number of nodes in a multi-node query when using this feature.

Use of timestamp-based snapshots does not increase inter-node traffic or bandwidth. Only the timestamp value is passed in addition to query data.

Replication Sets

A replication set is a group of tables which can be subscribed to by a BDR node. Replication sets can be used to create more complex replication topologies than regular symmetric multi-master where each node is exact copy of the other nodes.

Every BDR group automatically creates a replication set with the same name as the group itself. This replication set is the default replication set which is used for all user tables and DDL replication and all nodes are subscribed to it. In other words, by default all user tables are replicated between all nodes.

Behavior of Partitioned Tables

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

> **Note**
> When partitions are replicated through a partitioned table, the statements executed directly on a partition are replicated as they were executed on the parent table. The exception is the `TRUNCATE` command which always replicates with the list of affected tables or partitions.

It's 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 the case partitioning definition is the same on both provider and subscriber, as the partitioning logic does not have to be executed.

> **Note**
> If a 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.

In PostgreSQL 10 and older, BDR only allows replication of partitions directly to other partitions.

Behavior with Foreign Keys

A Foreign Key constraint ensures that each row in the referencing table matches a row in the referenced table. Therefore, if the referencing table is a member of a replication set, the referenced table must also be a member of the same replication set.

The current version of BDR does not automatically check or enforce this condition. It is therefore the responsibility of the database administrator to make sure, when adding a table to a replication set, that all the tables referenced via foreign keys are also added.

The following query can be used to list all the foreign keys and replication sets that do not satisfy this requirement, i.e. such that the referencing table is a member of the replication set, while the referenced table is not:

```
SELECT t1.relname,
       t1.nspname,
       fk.conname,
       t1.set_name
  FROM bdr.tables AS t1
  JOIN pg_catalog.pg_constraint AS fk
    ON fk.conrelid = t1.relid
   AND fk.contype = 'f'
 WHERE NOT EXISTS (
  SELECT *
    FROM bdr.tables AS t2
   WHERE t2.relid = fk.confrelid
     AND t2.set_name = t1.set_name
);
```

The output of this query looks like the following:

```
 relname | nspname |  conname  | set_name
---------+---------+-----------+----------
 t2      | public  | t2_x_fkey | s2
(1 row)
```

This means that table `t2` is member of replication set `s2`, but the table which is referenced by the foreign key `t2_x_fkey` is not.

> **Note**
> The `TRUNCATE CASCADE` command will take into account the replication set membership before replicating the command, e.g.

```
TRUNCATE table1 CASCADE;
```

This will become a `TRUNCATE` without cascade on all the tables that are part of the replication set only:

```
TRUNCATE table1, referencing_table1, referencing_table2 ...
```

Replication Set Management

Management of replication sets, the following functions:

Note that with the exception of `bdr.alter_node_replication_sets`, following functions are considered to be `DDL` so DDL replication and global locking applies to them, if that is currently active. See DDL Replication

bdr.create_replication_set

Creates a replication set.

Replication of this command is affected by DDL replication configuration including DDL filtering settings.

Synopsis

```
bdr.create_replication_set(set_name name,
                           replicate_insert boolean DEFAULT true,
                           replicate_update boolean DEFAULT true,
                           replicate_delete boolean DEFAULT true,
                           replicate_truncate boolean DEFAULT true,
                           autoadd_tables boolean DEFAULT false,
                           autoadd_existing boolean DEFAULT true)
```

Parameters

- `set_name` - name of the new replication set; must be unique across the BDR group
- `replicate_insert` - indicates whether inserts into tables in this replication set should be replicated
- `replicate_update` - indicates whether updates of tables in this replication set should be replicated
- `replicate_delete` - indicates whether deletes from tables in this replication set should be replicated
- `replicate_truncate` - indicates whether truncates of tables in this replication set should be replicated
- `autoadd_tables` - indicates whether newly created (future) tables should be added to this replication set
- `autoadd_existing` - indicates whether all existing user tables should be added to this replication set, this only has effect if `autoadd_tables` is set to true

Notes

By default, new replication sets do not replicate DDL or BDR administration function calls. See ddl filters below on how to set up DDL replication for replication sets. There is a preexisting DDL filter set up for the default group replication set which replicates all DDL and admin function calls, which is created when the

group is created, but can be dropped in case it's not desirable for the BDR group default replication set to replicate DDL or the BDR administration function calls.

This function uses the same replication mechanism as `DDL` statements. This means that the replication is affected by the ddl filters configuration.

The function will take a `DDL` global lock.

This function is transactional - the effects can be rolled back with the `ROLLBACK` of the transaction and the changes are visible to the current transaction.

bdr.alter_replication_set

Modifies options of an existing replication set.

Replication of this command is affected by DDL replication configuration including DDL filtering settings.

Synopsis

```
bdr.alter_replication_set(set_name name,
                          replicate_insert boolean DEFAULT NULL,
                          replicate_update boolean DEFAULT NULL,
                          replicate_delete boolean DEFAULT NULL,
                          replicate_truncate boolean DEFAULT NULL,
                          autoadd_tables boolean DEFAULT NULL)
```

Parameters

- `set_name` - name of an existing replication set
- `replicate_insert` - indicates whether inserts into tables in this replication set should be replicated
- `replicate_update` - indicates whether updates of tables in this replication set should be replicated
- `replicate_delete` - indicates whether deletes from tables in this replication set should be replicated
- `replicate_truncate` - indicates whether truncates of tables in this replication set should be replicated
- `autoadd_tables` - indicates whether newly created (future) tables should be added to this replication set

Any of the options that are set to NULL (the default) will remain the same as before.

Notes

This function uses same replication mechanism as `DDL` statements. This means the replication is affected by the ddl filters configuration.

The function will take a `DDL` global lock.

This function is transactional - the effects can be rolled back with the `ROLLBACK` of the transaction and the changes are visible to the current transaction.

bdr.drop_replication_set

Removes an existing replication set.

Replication of this command is affected by DDL replication configuration including DDL filtering settings.

Synopsis

```
bdr.drop_replication_set(set_name name)
```

Parameters

- `set_name` - name of an existing replication set

Notes

This function uses the same replication mechanism as `DDL` statements. This means the replication is affected by the ddl filters configuration.

The function will take a `DDL` global lock.

This function is transactional - the effects can be rolled back with the `ROLLBACK` of the transaction and the changes are visible to the current transaction.

> **Warning**
> Do not drop a replication set which is being used by at least another node, because this will stop replication on that node. Should this happen, please unsubscribe the affected node from that replication set.
> For the same reason, you should not drop a replication set if there is a join operation in progress, and the node being joined is a member of that replication set; replication set membership is only checked at the beginning of the join.
> This happens because the information on replication set usage is local to each node, so that it can be configured on a node before it joins the group.

You can manage replication set subscription for a node using `alter_node_replication_sets` which is mentioned below.

bdr.alter_node_replication_sets

Changes which replication sets a node publishes and is subscribed to.

Synopsis

```
bdr.alter_node_replication_sets(node_name name,
                                set_names text[])
```

Parameters

- `node_name` - which node to modify; currently has to be local node
- `set_names` - array of replication sets to replicate to the specified node; an empty array will result in the use of the group default replication set

Notes

This function is only executed on the local node and not replicated in any manner.

The replication sets listed are *not* checked for existence, since this function is designed to be executed before the node joins. Be careful to specify replication set names correctly to avoid errors.

This allows for calling the function not only on the node that's part of the BDR group, but also on a node that hasn't joined any group yet in order to limit what data is synchronized during the join. However please note that schema is *always fully synchronized* without regard to the replication sets setting, meaning that all tables are copied across, not just the ones specified in the replication set. Unwanted tables can be dropped by referring to the `bdr.tables` catalog table. These might be removed automatically in later versions of BDR. This is currently true even if the ddl filters configuration would otherwise prevent replication of DDL.

Replication Set Membership

Tables can be added and removed to one or multiple replication sets. This only affects replication of changes (DML) in those tables, schema changes (DDL) handled by DDL replication set filters (see DDL Replication Filtering below).

The replication uses the table membership in replication sets in combination with the node replication sets configuration to determine which actions should be replicated to which node. The decision is done using the union of all the memberships and replication set options. This means that if a table is a member of replication set A which replicates only INSERTs and replication set B which replicates only UPDATEs, both INSERTs and UPDATEs will be replicated if the target node is also subscribed to both replication set A and B.

bdr.replication_set_add_table

Adds a table to a replication set.

This will add a table to replication set and start replication of changes from this moment (or rather transaction commit). Any existing data the table may have on a node will not be synchronized.

Replication of this command is affected by DDL replication configuration including DDL filtering settings.

Synopsis

```
bdr.replication_set_add_table(relation regclass,
                              set_name name DEFAULT NULL,
                              columns text[] DEFAULT NULL,
                              row_filter text DEFAULT NULL)
```

Parameters

- `relation` - name or Oid of a table
- `set_name` - name of the replication set; if NULL (the default) the BDR group default replication set is used
- `columns` - reserved for future use (currently does nothing and must be NULL)
- `row_filter` - SQL expression to be used for filtering the replicated rows; if this expression is not defined (i.e. NULL - the default) all rows are sent

The `row_filter` specifies an expression producing a Boolean result, with NULLs. Expressions evaluating to True or Unknown will replicate the row; a False value will not replicate the row. Expressions cannot contain subqueries nor refer to variables other than columns of the current row being replicated. No system columns may be referenced.

`row_filter` executes on the origin node, not on the target node. This puts an additional CPU overhead on replication for this specific table, but will completely avoid sending data for filtered rows, hence reducing network bandwidth and apply overhead on the target node.

`row_filter` will never remove `TRUNCATE` commands for a specific table. `TRUNCATE` commands can be filtered away at the replication set level; see earlier.

It is possible to replicate just some columns of a table, see Replicating between nodes with differences.

Notes

This function uses same replication mechanism as `DDL` statements. This means the replication is affected by the ddl filters configuration.

The function will take a `DML` global lock on the relation that's being added to the replication set if the `row_filter` is not NULL, otherwise it will take just a `DDL` global lock.

This function is transactional - the effects can be rolled back with the `ROLLBACK` of the transaction and the changes are visible to the current transaction.

bdr.replication_set_remove_table

Removes a table from the replication set.

Replication of this command is affected by DDL replication configuration including DDL filtering settings.

Synopsis

```
bdr.replication_set_remove_table(relation regclass,
                                 set_name name DEFAULT NULL)
```

Parameters

- `relation` - name or Oid of a table
- `set_name` - name of the replication set; if NULL (the default) then the BDR group default replication set is used

Notes

This function uses same replication mechanism as `DDL` statements. This means the replication is affected by the ddl filters configuration.

The function will take a `DDL` global lock.

This function is transactional - the effects can be rolled back with the `ROLLBACK` of the transaction and the changes are visible to the current transaction.

Listing Replication Sets

Existing replication sets can be listed with the following query:

```
SELECT set_name
FROM bdr.replication_sets;
```

This query can be used to list all the tables in a given replication set:

```
SELECT nspname, relname
FROM bdr.tables
WHERE set_name = 'myrepset';
```

In the section Behavior with Foreign Keys above we report a query that lists all the foreign keys whose referenced table is not included in the same replication set as the referencing table.

Use the following SQL to show the replication sets the current node publishes and subscribes from:

```
SELECT s.node_id,
       s.node_name,
       COALESCE(
         i.replication_sets,
         ARRAY[s.default_repset_name]::NAME[]
       ) AS replication_sets
FROM bdr.local_node_summary s
INNER JOIN bdr.node_local_info i ON i.node_id = s.node_id;
```

This produces output like this:

```
  node_id   | node_name |  replication_sets
------------+-----------+--------------------
 1834550102 | s01db01   | {bdrglobal,bdrs01}
(1 row)
```

To get the same query executed on against all nodes in the cluster, thus getting which replication sets are associated to all nodes at the same time, we can use the following query:

```
WITH node_repsets AS (
  SELECT jsonb_array_elements(
    bdr.run_on_all_nodes($$
      SELECT s.node_id,
             s.node_name,
             COALESCE(
               i.replication_sets,
               ARRAY[s.default_repset_name]::NAME[]
             ) AS replication_sets
      FROM bdr.local_node_summary s
      INNER JOIN bdr.node_local_info i ON i.node_id = s.node_id
    $$)::jsonb
  ) AS j
)
SELECT j->'response'->0->>'node_id' AS node_id,
       j->'response'->0->>'node_name' AS node_name,
       j->'response'->0->>'replication_sets' AS replication_sets
FROM node_repsets;
```

This will show, for example:

```
  node_id   | node_name |  replication_sets
------------+-----------+--------------------
 933864801  | s02db01   | {bdrglobal,bdrs02}
 1834550102 | s01db01   | {bdrglobal,bdrs01}
 3898940082 | s01db02   | {bdrglobal,bdrs01}
 1102086297 | s02db02   | {bdrglobal,bdrs02}
(4 rows)
```

DDL Replication Filtering

By default the replication of all supported DDL happens via the default BDR group replication set. This is achieved by the existence of a DDL filter with the same name as the BDR group which is automatically added to the default BDR group replication set when the BDR group is created.

The above can be adjusted by changing the DDL replication filters for all existing replication sets. These filters are independent of table membership in the replication sets. Just like data changes, each DDL statement will be replicated only once no matter if it's matched by multiple filters on multiple replication sets.

You can list existing DDL filters with the following query, which shows for each filter the regular expression applied to the command tag and to the role name:

```
SELECT * FROM bdr.ddl_replication;
```

The following functions can be used to manipulate DDL filters. Note that they are considered to be `DDL`, and therefore subject to DDL replication and global locking.

bdr.replication_set_add_ddl_filter

Adds a DDL filter to a replication set.

Any DDL that matches the given filter will be replicated to any node which is subscribed to that set. This also affects replication of BDR admin functions.

Note that this does not prevent execution of DDL on any node, it only alters whether DDL is replicated, or not, to other nodes. So if two nodes have a replication filter between them that excludes all index commands, then index commands can still be executed freely by directly connecting to each node and executing the desired DDL on that node.

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 for regular PostgreSQL commands. A typical example might be to create a filter that prevents additional index commands on a logical standby from being replicated to all other nodes.

The BDR admin functions use can be filtered using a tagname matching the qualified function name (for example `bdr.replication_set_add_table` will be the command tag for the function of the same name). For example, this allows all BDR functions to be filtered using `bdr.*`.

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.

Synopsis

```
bdr.replication_set_add_ddl_filter(set_name name,
                                   ddl_filter_name text,
                                   command_tag text,
                                   role_name text DEFAULT NULL)
```

Parameters

- `set_name` - name of the replication set; if NULL the BDR group default replication set is used
- `ddl_filter_name` - name of the DDL filter; this must be unique across the whole BDR group
- `command_tag` - regular expression for matching command tags, NULL means match everything
- `role_name` - regular expression for matching role name, NULL means match all roles

Notes

This function uses same replication mechanism as `DDL` statements. This means the replication is affected by the ddl filters configuration. Please note that this means that replication of changes to ddl filter configuration is affected by existing ddl filter configuration!

The function will take a `DDL` global lock.

This function is transactional - the effects can be rolled back with the `ROLLBACK` of the transaction and the changes are visible to the current transaction.

To view which replication filters are defined, use the view `bdr.ddl_replication`.

Examples

To include only BDR admin functions, define a filter like this

```
SELECT bdr.replication_set_add_ddl_filter('mygroup', 'mygroup_admin', $$bdr\..*$$);
```

To exclude everything apart from index DDL

```
SELECT bdr.replication_set_add_ddl_filter('mygroup', 'index_filter',
                '^(?!(CREATE INDEX|DROP INDEX|ALTER INDEX)).*');
```

To include all operations on tables and indexes, but exclude all others, add two filters, one for tables, one for indexes. This illustrates that multiple filters provide the union of all allowed DDL commands.

```
SELECT bdr.replication_set_add_ddl_filter('bdrgroup','index_filter', '^((?!INDEX).)*$');
SELECT bdr.replication_set_add_ddl_filter('bdrgroup','table_filter', '^((?!TABLE).)*$');
```

bdr.replication_set_remove_ddl_filter

Remove DDL filter from a replication set.

Replication of this command is affected by DDL replication configuration including DDL filtering settings themselves!

Synopsis

```
bdr.replication_set_remove_ddl_filter(set_name name,
                                      ddl_filter_name text)
```

Parameters

- `set_name` - name of the replication set, if NULL the BDR group default replication set is used
- `ddl_filter_name` - name of the DDL filter to remove

Notes

This function uses the same replication mechanism as `DDL` statements. This means that the replication is affected by the ddl filters configuration. Please note that this means that replication of changes to the DDL filter configuration is affected by the existing DDL filter configuration.

The function will take a `DDL` global lock.

This function is transactional - the effects can be rolled back with the `ROLLBACK` of the transaction and the changes are visible to the current transaction.

Stream Triggers

BDR introduces new types of triggers which can be used for additional data processing on the downstream/target node.

- Conflict Triggers
- Transform Triggers

Together, these types of triggers are known as Stream Triggers.

Stream Triggers are designed to be trigger-like in syntax, they leverage the PostgreSQL BEFORE trigger architecture, and are likely to have similar performance characteristics as PostgreSQL BEFORE Triggers.

One trigger function can be used by multiple trigger definitions, just as with normal PostgreSQL triggers. A trigger function is simply a program defined in this form: `CREATE FUNCTION ... RETURNS TRIGGER`. Creating the actual trigger does not require use of the CREATE TRIGGER command. Instead, stream triggers are created using the special BDR functions `bdr.create_conflict_trigger()` and `bdr.create_transform_trigger()`.

Once created, the trigger will be visible in the catalog table `pg_trigger`. The stream triggers will be marked as `tgisinternal = true` and `tgenabled = false` and will have name suffix '_bdrc' or '_bdrt'. The view `bdr.triggers` provides information on the triggers in relation to the table, the name of the procedure that is being executed, the event that triggers it and, the trigger type.

Note that stream triggers are NOT therefore enabled for normal SQL processing. Because of this the `ALTER TABLE ... ENABLE TRIGGER` is blocked for stream triggers in both its specific name variant and the ALL variant, to prevent the trigger from executing as a normal SQL trigger.

Note that these triggers execute on the downstream or target node. There is no option for them to execute on the origin node, though may wish to consider the use of `row_filter` expressions on the origin.

Also, any DML which is applied during the execution of a stream trigger will not be replicated to other BDR nodes, and will not trigger the execution of standard local triggers. This is intentional, and can be used for instance to log changes or conflicts captured by a stream trigger into a table that is crash-safe and specific of that node; a working example is provided at the end of this chapter.

Trigger execution during Apply

Transform triggers execute first, once for each incoming change in the triggering table. These triggers fire before we have even attempted to locate a matching target row, allowing a very wide range of transforms to be applied efficiently and consistently.

Next, for UPDATE and DELETE changes we locate the target row. If there is no target row then there is no further processing for those change types.

We then execute any normal triggers that previously have been explicitly enabled as replica triggers at table-level:

```
ALTER TABLE tablename
ENABLE REPLICA TRIGGER trigger_name;
```

We then decide whether a potential conflict exists and if so, we then call any conflict trigger that exists for that table.

Missing Column Conflict Resolution

Before transform triggers are executed, PostgreSQL tries to match the incoming tuple against the rowtype of the target table.

Any column that exists on the input row but not on the target table will trigger a conflict of type `target_column_missing`; conversely, a column existing on the target table but not in the incoming row triggers a `source_column_missing` conflict. The default resolutions for those two conflict types are respectively `ignore_if_null` and `use_default_value`.

This is relevant in the context of rolling schema upgrades, for instance if the new version of the schema introduces a new column. When replicating from an old version of the schema to a new one, the source column is missing, and the `use_default_value` strategy is appropriate, as it populates the newly introduced column with the default value.

However, when replicating from a node having the new schema version to a node having the old one, the column is missing from the target table, and the `ignore_if_null` resolver is not appropriate for a rolling upgrade, because it will break replication as soon as the user inserts, in any of the upgraded nodes, a tuple with a non-NULL value in the new column.

In view of this example, the appropriate setting for rolling schema upgrades is to configure each node to apply the `ignore` resolver in case of a `target_column_missing` conflict.

This is done with the following query, that must be **executed separately on each node**, after replacing `node1` with the actual node name:

```
SELECT bdr.alter_node_set_conflict_resolver('node1',
    'target_column_missing', 'ignore');
```

Data Loss and Divergence Risk

In this section we show how setting the conflict resolver to `ignore` can lead to data loss and cluster divergence.

Consider the following example: table `t` exists on nodes 1 and 2, but its column `col` only exists on node 1.

If the conflict resolver is set to `ignore`, then there can be rows on node 1 where `c` is not null, e.g. `(pk=1, col=100)`. That row will be replicated to node 2, and the value in column `c` will be discarded, e.g. `(pk=1)`.

If column `c` is then added to the table on node 2, it will initially be set to NULL on all existing rows, and the row considered above becomes `(pk=1, col=NULL)`: the row having `pk=1` is no longer identical on all nodes, and the cluster is therefore divergent.

Note that the default `ignore_if_null` resolver is not affected by this risk, because any row that is replicated to node 2 will have `col=NULL`.

Based on this example, we recommend running LiveCompare against the whole cluster at the end of a rolling schema upgrade where the `ignore` resolver was used, to make sure that any divergence is detected and fixed.

Terminology of row-types.

This document uses these row-types:

- `SOURCE_OLD` is the row before update, i.e. the key.
- `SOURCE_NEW` is the new row coming from another node.
- `TARGET` is row that exists on the node already, i.e. conflicting row.

Conflict Triggers

Conflict triggers are executed when a conflict is detected by BDR and are used to decide what happens when the conflict has occurred.

- If the trigger function returns a row the action will be applied to the target.
- If the trigger function returns NULL row the action will be skipped.

To clarify, if the trigger is called for a `DELETE`, the trigger should return NULL if it wants to skip the `DELETE`. If you wish the DELETE to proceed then return a row value - either `SOURCE_OLD` or `TARGET` will work. When the conflicting operation is either `INSERT` or `UPDATE`, and the chosen resolution is the deletion of the conflicting row, the trigger must explicitly perform the deletion and return NULL. The trigger function may perform other SQL actions as it chooses, but those actions will only be applied locally, not replicated.

When a real data conflict occurs between two or more nodes there will be two or more concurrent changes occurring. When we apply those changes the conflict resolution occurs independently on each node. This means the conflict resolution will occur once on each node and can occur with a significant time difference between then. As a result there is no possibility of communication between the multiple executions of the conflict trigger. It is the responsibility of the author of the conflict trigger to ensure that the trigger gives exactly the same result for all related events otherwise data divergence will occur. 2ndQuadrant recommends that all conflict triggers are formally tested using the isolationtester tool, supplied with BDR.

> **Warning**
> - Multiple conflict triggers can be specified on a single table, but they should match distinct event, i.e. each conflict should only match a single conflict trigger.
> Multiple triggers matching the same event on the same table are not recommended; they might result in inconsistent behaviour, and will be forbidden in a future release.

If the same conflict trigger matches more than one event, the `TG_OP` variable can be used within the trigger to identify the operation that produced the conflict.

By default, BDR detects conflicts by observing a change of replication origin for a row, hence it is possible for a conflict trigger to be called even when there is only one change occurring. Since in this case there is no real conflict we say that this conflict detection mechanism can generate false positive conflicts. The conflict trigger must handle all of those identically, as mentioned above.

Note that in some cases, timestamp conflict detection will not detect a conflict at all. For example, in a concurrent UPDATE/DELETE where the DELETE occurs just after the UPDATE, any nodes that see first the UPDATE and then the DELETE will not see any conflict. If no conflict is seen, the conflict trigger will never be called. The same situation, but using row version conflict detection *will* see a conflict, which can then be handled by a conflict trigger.

The trigger function has access to additional state information as well as the data row involved in the conflict, depending upon the operation type:

- On `INSERT`, conflict triggers would be able to access `SOURCE_NEW` row from source and `TARGET` row
- On `UPDATE`, conflict triggers would be able to access `SOURCE_OLD` and `SOURCE_NEW` row from source and `TARGET` row
- On `DELETE`, conflict triggers would be able to access `SOURCE_OLD` row from source and `TARGET` row

The function `bdr.trigger_get_row()` can be used to retrieve `SOURCE_OLD`, `SOURCE_NEW` or `TARGET` rows, if a value exists for that operation.

Changes to conflict triggers happen transactionally and are protected by Global DML Locks during replication of the configuration change, similarly to how some variants of `ALTER TABLE` are handled.

If primary keys are updated inside a conflict trigger, it can sometimes leads to unique constraint violations error due to difference in timing of execution. Hence users should avoid updating primary keys within conflict triggers.

Transform Triggers

These triggers are similar to the Conflict Triggers, except they are executed for every row on the data stream against the specific table. The behavior of return values and the exposed variables are similar, but transform triggers execute before a target row is identified, so there is no `TARGET` row.

Specify multiple Transform Triggers on each table in BDR, if desired. Transform triggers execute in alphabetical order.

A transform trigger can filter away rows, and it can do additional operations as needed. It can alter the values of any column, or set them to `NULL`. The return value decides what further action is taken:

- If the trigger function returns a row it will be applied to the target.

- If the trigger function returns a `NULL` row there is no further action to be performed and as-yet unexecuted triggers will never execute.
- The trigger function may perform other actions as it chooses.

The trigger function has access to additional state information as well as rows involved in the conflict:

- On `INSERT`, transform triggers would be able to access `SOURCE_NEW` row from source.
- On `UPDATE`, transform triggers would be able to access `SOURCE_OLD` and `SOURCE_NEW` row from source.
- On `DELETE`, transform triggers would be able to access `SOURCE_OLD` row from source.

The function `bdr.trigger_get_row()` can be used to retrieve `SOURCE_OLD` or `SOURCE_NEW` rows; `TARGET` row is not available since this type of trigger executes before such a target row is identified, if any.

Transform Triggers look very similar to normal BEFORE row triggers, but have these important differences:

- Transform trigger gets called for every incoming change. BEFORE triggers will not be called at all for UPDATE and DELETE changes if we don't find matching row in a table.
- Transform triggers are called before partition table routing occurs
- Transform triggers have have access to the lookup key via SOURCE_OLD, which is not available to normal SQL triggers

Stream Triggers Variables

Both conflict trigger and transform triggers have access to information about rows and metadata via the predefined variables provided by trigger API and additional information functions provided by BDR.

In PL/pgSQL the following predefined variables exist:

TG_NAME

Data type name; variable that contains the name of the trigger actually fired. Note that the actual trigger name has a '_bdrt' or '_bdrc' suffix (depending on trigger type) compared to the name provided during trigger creation.

TG_WHEN

Data type text; this will say `BEFORE` for both Conflict and Transform triggers. The stream trigger type, can be obtained by calling `bdr.trigger_get_type()` information function (see below).

TG_LEVEL

Data type text; a string of `ROW`.

TG_OP

Data type text; a string of `INSERT`, `UPDATE` or `DELETE` telling for which operation the trigger was fired.

TG_RELID

Data type oid; the object ID of the table that caused the trigger invocation.

TG_TABLE_NAME

Data type name; the name of the table that caused the trigger invocation.

TG_TABLE_SCHEMA

Data type name; the name of the schema of the table that caused the trigger invocation. For partitioned tables, this is the name of the root table.

TG_NARGS

Data type integer; the number of arguments given to the trigger function in the `bdr.create_conflict_trigger()` or `bdr.create_transform_trigger()` statement.

TG_ARGV[]

Data type array of text; the arguments from the `bdr.create_conflict_trigger()` or `bdr.create_transform_trigg` statement. The index counts from 0. Invalid indexes (less than 0 or greater than or equal to `TG_NARGS`) result in a `NULL` value.

Information functions

bdr.trigger_get_row

Returns contents of a trigger row specified by an identifier as a `RECORD`. This function returns NULL if called inappropriately, i.e. called with SOURCE_NEW when the operation type (TG_OP) is DELETE.

Synopsis

```
bdr.trigger_get_row(row_id text)
```

Parameters

- `row_id` - identifier of the row, can be any of SOURCE_NEW, SOURCE_OLD and TARGET depending on the trigger type and operation (see documentation of individual trigger types).

bdr.trigger_get_committs

Returns commit timestamp of a trigger row specified by an identifier. If not available because a row is frozen or row is not available, this will return NULL. Always returns NULL for row identifier SOURCE_OLD.

Synopsis

```
bdr.trigger_get_committs(row_id text)
```

Parameters

- `row_id` - identifier of the row, can be any of SOURCE_NEW, SOURCE_OLD and TARGET depending on trigger type and operation (see documentation of individual trigger types).

bdr.trigger_get_xid

Returns local transaction id of a TARGET row specified by an identifier. If not available because a row is frozen or row is not available, this will return NULL. Always returns NULL for SOURCE_OLD and SOURCE_NEW row identifiers.

This is only available for conflict triggers.

Synopsis

```
bdr.trigger_get_xid(row_id text)
```

Parameters

- `row_id` - identifier of the row, can be any of SOURCE_NEW, SOURCE_OLD and TARGET depending on trigger type and operation (see documentation of individual trigger types).

bdr.trigger_get_type

Returns the current trigger type which can be either `CONFLICT` or `TRANSFORM`. Returns null if called outside a Stream Trigger.

Synopsis

```
bdr.trigger_get_type()
```

bdr.trigger_get_conflict_type

Returns the current conflict type if called inside a conflict trigger, or `NULL` otherwise.

See Conflict Types for possible return values of this function.

bdr.trigger_get_origin_node_id

Returns the node id corresponding to the origin for the trigger row_id passed in as argument. If the origin is not valid which means the row has originated locally, return the node id of the source or target node depending on the trigger row argument. Always returns NULL for row identifier SOURCE_OLD. This can be used to define conflict triggers to always favour a trusted source node. See the example given below.

Synopsis

```
bdr.trigger_get_origin_node_id(row_id text)
```

Parameters

- `row_id` - identifier of the row, can be any of SOURCE_NEW, SOURCE_OLD and TARGET depending on trigger type and operation (see documentation of individual trigger types).

Synopsis

```
bdr.trigger_get_conflict_type()
```

bdr.ri_fkey_on_del_trigger

When called as a BEFORE trigger, will use FOREIGN KEY information to avoid FK anomalies.

Synopsis

```
bdr.ri_fkey_on_del_trigger()
```

Row Contents

The SOURCE_NEW, SOURCE_OLD and TARGET contents depend on the operation, REPLICA IDENTITY setting of a table and the contents of the target table.

The TARGET row is only available in conflict triggers. The TARGET row only contains data if a row was found when applying UPDATE or DELETE in the target table; if the row is not found, the TARGET will be NULL.

Triggers Notes

Execution order for triggers:

- Transform triggers - execute once for each incoming row on the target
- Normal triggers - execute once per row
- Conflict triggers - execute once per row where a conflict exists

Stream Triggers Manipulation Interfaces

Stream Triggers are managed using SQL interfaces provided as part of BDR-Enterprise extension.

Stream Triggers can only be created on tables with `REPLICA IDENTITY FULL` or tables without any `TOAST`able columns.

bdr.create_conflict_trigger

Creates a new conflict trigger.

Synopsis

```
bdr.create_conflict_trigger(trigger_name text,
                            events text[],
                            relation regclass,
                            function regprocedure,
                            args text[] DEFAULT '{}')
```

Parameters

- `trigger_name` - name of the new trigger
- `events` - array of events on which to fire this trigger; valid values are '`INSERT`', '`UPDATE`' and '`DELETE`'
- `relation` - for which relation to fire this trigger
- `function` - which function to execute
- `args` - optional, specify array of parameters the trigger function will receive upon execution (contents of `TG_ARGV` variable)

Notes

This function uses the same replication mechanism as `DDL` statements. This means that the replication is affected by the ddl filters configuration.

The function will take a global DML lock on the relation on which the trigger is being created.

This function is transactional - the effects can be rolled back with the `ROLLBACK` of the transaction and the changes are visible to the current transaction.

Similarly to normal PostgreSQL triggers, the `bdr.create_conflict_trigger` function requires `TRIGGER` privilege on the `relation` and `EXECUTE` privilege on the function. This applies with a `bdr.backwards_compatibility` of 30619 or above. Additional security rules apply in BDR to all triggers including conflict triggers, see the security chapter ontriggers.

bdr.create_transform_trigger

Creates a new transform trigger.

Synopsis

```
bdr.create_transform_trigger(trigger_name text,
                             events text[],
                             relation regclass,
                             function regprocedure,
                             args text[] DEFAULT '{}')
```

Parameters

- `trigger_name` - name of the new trigger
- `events` - array of events on which to fire this trigger, valid values are '`INSERT`', '`UPDATE`' and '`DELETE`'
- `relation` - for which relation to fire this trigger
- `function` - which function to execute
- `args` - optional, specify array of parameters the trigger function will receive upon execution (contents of `TG_ARGV` variable)

Notes

This function uses the same replication mechanism as `DDL` statements. This means that the replication is affected by the ddl filters configuration.

The function will take a global DML lock on the relation on which the trigger is being created.

This function is transactional - the effects can be rolled back with the `ROLLBACK` of the transaction and the changes are visible to the current transaction.

Similarly to normal PostgreSQL triggers, the `bdr.create_transform_trigger` function requires `TRIGGER` privilege on the `relation` and `EXECUTE` privilege on the function. Additional security rules apply in BDR to all triggers including transform triggers, see the security chapter on triggers.

bdr.drop_trigger

Removes an existing stream trigger (both conflict and transform).

Synopsis

```
bdr.drop_trigger(trigger_name text,
                 relation regclass,
                 ifexists boolean DEFAULT false)
```

Parameters

- `trigger_name` - name of an existing trigger
- `relation` - which relation is the trigger defined for
- `ifexists` - when set to true `true`, this command will ignore missing triggers

Notes

This function uses the same replication mechanism as `DDL` statements. This means that the replication is affected by the ddl filters configuration.

The function will take a global DML lock on the relation on which the trigger is being created.

This function is transactional - the effects can be rolled back with the `ROLLBACK` of the transaction and the changes are visible to the current transaction.

The `bdr.drop_trigger` function can be only executed by the owner of the `relation`.

Stream Triggers Examples

A conflict trigger which provides similar behavior as the update_if_newer conflict resolver:

```
CREATE OR REPLACE FUNCTION update_if_newer_trig_func
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF (bdr.trigger_get_committs('TARGET') >
        bdr.trigger_get_committs('SOURCE_NEW')) THEN
    RETURN TARGET;
    ELSIF
```

```
        RETURN SOURCE;
    END IF;
END;
$$;
```

A conflict trigger which applies a delta change on a counter column and uses SOURCE_NEW for all other columns:

```
CREATE OR REPLACE FUNCTION delta_count_trg_func
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    DELTA bigint;
    SOURCE_OLD record;
    SOURCE_NEW record;
    TARGET record;
BEGIN
    SOURCE_OLD := bdr.trigger_get_row('SOURCE_OLD');
    SOURCE_NEW := bdr.trigger_get_row('SOURCE_NEW');
    TARGET := bdr.trigger_get_row('TARGET');

    DELTA := SOURCE_NEW.counter - SOURCE_OLD.counter;
    SOURCE_NEW.counter = TARGET.counter + DELTA;

    RETURN SOURCE_NEW;
END;
$$;
```

A transform trigger which logs all changes to a log table instead of applying them:

```
CREATE OR REPLACE FUNCTION log_change
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    SOURCE_NEW record;
    SOURCE_OLD record;
    COMMITTS timestamptz;
BEGIN
    SOURCE_NEW := bdr.trigger_get_row('SOURCE_NEW');
    SOURCE_OLD := bdr.trigger_get_row('SOURCE_OLD');
    COMMITTS := bdr.trigger_get_committs('SOURCE_NEW');

    IF (TG_OP = 'INSERT') THEN
        INSERT INTO log SELECT 'I', COMMITTS, row_to_json(SOURCE_NEW);
    ELSIF (TG_OP = 'UPDATE') THEN
```

```
        INSERT INTO log SELECT 'U', COMMITTS, row_to_json(SOURCE_NEW);
    ELSIF (TG_OP = 'DELETE') THEN
        INSERT INTO log SELECT 'D', COMMITTS, row_to_json(SOURCE_OLD);
    END IF;

    RETURN NULL; -- do not apply the change
END;
$$;
```

A conflict trigger that implements Trusted Source conflict detection, also known as trusted site, preferred node or Always Wins resolution. This uses the bdr.trigger_get_origin_node_id() function to provide a solution that works with 3 or more nodes.

```
CREATE OR REPLACE FUNCTION test_conflict_trigger()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    SOURCE  record;
    TARGET  record;

    TRUSTED_NODE    bigint;
    SOURCE_NODE     bigint;
    TARGET_NODE     bigint;
BEGIN
    TARGET := bdr.trigger_get_row('TARGET');
    IF (TG_OP = 'DELETE')
        SOURCE := bdr.trigger_get_row('SOURCE_OLD');
    ELSE
        SOURCE := bdr.trigger_get_row('SOURCE_NEW');
    END IF;

    TRUSTED_NODE := current_setting('customer.trusted_node_id');

    SOURCE_NODE := bdr.trigger_get_origin_node_id('SOURCE_NEW');
    TARGET_NODE := bdr.trigger_get_origin_node_id('TARGET');

    IF (TRUSTED_NODE = SOURCE_NODE) THEN
        RETURN SOURCE;
    ELSIF (TRUSTED_NODE = TARGET_NODE) THEN
        RETURN TARGET;
    ELSE
        RETURN NULL; -- do not apply the change
    END IF;
END;
$$;
```

Monitoring

Monitoring replication setups is important to ensure that your system performs optimally and does not run out of disk space or encounter other faults that may halt operations.

It is important to have automated monitoring in place to ensure that if, for example, replication slots start falling badly behind, the administrator is alerted and can take proactive action.

BDR Cluster Manager, a plugin for OmniDB, provides a packaged monitoring solution that is available via the 2ndQuadrant portal.

In addition, tools or users can make their own calls into BDR using the facilities discussed below. When you wish to make the same request of multiple BDR nodes, consider using the `bdr.run_on_all_nodes()` function to simplify the task.

Monitoring Node Join and Removal

By default, the node management functions wait for the join or part operation to complete. This can be turned off using the respective `wait_for_completion` function argument. If waiting is turned off, then to see when a join or part operation finishes, check the node state indirectly via `bdr.node_summary` and `bdr.state_journal_details`.

When called, the helper function `bdr.wait_for_join_completion()` will cause a PostgreSQL session to pause until all outstanding node join operations complete.

Here is an example output of a `SELECT` query from `bdr.node_summary` that indicates that two nodes are active and another one is joining:

```
# SELECT node_name, interface_connstr, peer_state_name,
#     node_seq_id, node_local_dbname
# FROM bdr.node_summary;
-[ RECORD 1 ]-----+-----------------------------------------
node_name         | node1
interface_connstr | host=localhost dbname=postgres port=7432
peer_state_name   | ACTIVE
node_seq_id       | 1
node_local_dbname | postgres
-[ RECORD 2 ]-----+-----------------------------------------
node_name         | node2
interface_connstr | host=localhost dbname=postgres port=7433
peer_state_name   | ACTIVE
node_seq_id       | 2
node_local_dbname | postgres
-[ RECORD 3 ]-----+-----------------------------------------
node_name         | node3
interface_connstr | host=localhost dbname=postgres port=7434
```

```
peer_state_name   | JOINING
node_seq_id       | 3
node_local_dbname | postgres
```

Also, the table `bdr.node_catchup_info` will give information on the catch-up state, which can be relevant to joining nodes or parting nodes.

When a node is parted, it could be that some nodes in the cluster did not receive all the data from that parting node. So it will create a temporary slot from a node that already received that data and can forward it.

The `catchup_state` can be one of the following:

```
10 = setup
20 = start
30 = catchup
40 = done
```

Monitoring Replication Peers

There are two main views used for monitoring of replication activity:

- `bdr.node_slots` for monitoring outgoing replication
- `bdr.subscription_summary` for monitoring incoming replication

Most of the information provided by `bdr.node_slots` can be also obtained by querying the standard PostgreSQL replication monitoring views `pg_catalog.pg_stat_replication` and `pg_catalog.pg_replication_slots`.

Each node has one BDR group slot which should never have a connection to it and will very rarely be marked as active. This is normal, and does not imply something is down or disconnected. See `Replication Slots created by BDR`.

Monitoring Outgoing Replication

There are two additional views used for monitoring of outgoing replication activity:

- `bdr.node_replication_rates` for monitoring outgoing replication
- `bdr.node_estimates` for estimates on catchup interval of peer nodes

The `bdr.node_replication_rates` view gives an overall picture of the outgoing replication activity whereas the `bdr.node_estimates` focuses on the catchup estimates for peer nodes, specifically.

```
# SELECT * FROM bdr.node_replication_rates;
-[ RECORD 1 ]----+-----------
target_name      | node1
sent_lsn         | 0/28AF99C8
replay_lsn       | 0/28AF99C8
replay_lag       | 00:00:00
replay_lag_bytes | 0
replay_lag_size  | 0 bytes
apply_rate       | 822
catchup_interval | 00:00:00
-[ RECORD 2 ]----+-----------
target_name      | node3
sent_lsn         | 0/28AF99C8
replay_lsn       | 0/28AF99C8
replay_lag       | 00:00:00
replay_lag_bytes | 0
replay_lag_size  | 0 bytes
apply_rate       | 853
catchup_interval | 00:00:00
```

The `apply_rate` above refers to the rate in bytes per second. It is the rate at which the peer is consuming data from the local node. The `replay_lag` when a node reconnects to the cluster is immediately set to zero. We are working on fixing this information; as a workaround, we suggest you use the `catchup_interval` column that refers to the time required for the peer node to catch up to the local node data. The other fields are also available via the `bdr.node_slots` view, as explained below.

Administrators may query `bdr.node_slots` for outgoing replication from the local node. It shows information about replication status of all other nodes in the group that are known to the current node, as well as any additional replication slots created by BDR on the current node.

```
# SELECT node_group_name, target_dbname, target_name, slot_name, active_pid,
#     catalog_xmin, client_addr, sent_lsn, replay_lsn, replay_lag,
#     replay_lag_bytes, replay_lag_size
# FROM bdr.node_slots;
-[ RECORD 1 ]---+--------------------------
node_group_name | bdrgroup
target_dbname   | postgres
target_name     | node3
slot_name       | bdr_postgres_bdrgroup_node3
active_pid      | 15089
catalog_xmin    | 691
client_addr     | 127.0.0.1
sent_lsn        | 0/23F7B70
replay_lsn      | 0/23F7B70
replay_lag      | [NULL]
replay_lag_bytes| 120
replay_lag_size | 120 bytes
```

```
-[ RECORD 2 ]---+--------------------------
node_group_name | bdrgroup
target_dbname   | postgres
target_name     | node2
slot_name       | bdr_postgres_bdrgroup_node2
active_pid      | 15031
catalog_xmin    | 691
client_addr     | 127.0.0.1
sent_lsn        | 0/23F7B70
replay_lsn      | 0/23F7B70
replay_lag      | [NULL]
replay_lag_bytes| 84211
replay_lag_size | 82 kB
```

Note that because BDR is a mesh network, to get full view of lag in the cluster, this query has to be executed on all nodes participating.

`replay_lag_bytes` reports the difference in WAL positions between the local server's current WAL write position and `replay_lsn`, the last position confirmed replayed by the peer node. `replay_lag_size` is just a human-readable form of the same. It is important to understand that WAL usually contains a lot of writes that are not replicated but still count in `replay_lag_bytes`, including `VACUUM` activity, index changes, writes associated with other databases on the same node, writes for tables that are not part of a replication set, etc. So the lag in bytes reported here is not the amount of data that must be replicated on the wire to bring the peer node up to date, only the amount of server-side WAL that must be processed.

Similarly, `replay_lag` is not a measure of how long the peer node will take to catch up, or how long it will take to replay from its current position to the write position at the time `bdr.node_slots` was queried. It measures the delay between when the peer confirmed the most recent commit and the current wall-clock time. We suggest that you monitor `replay_lag_bytes` and `replay_lag_size` or `catchup_interval` in `bdr.node_replication_rates`, as this column is set to zero immediately after the node reconnects.

The lag in both bytes and time does not advance while logical replication is streaming a transaction. It only changes when a commit is replicated. So the lag will tend to "sawtooth", rising as a transaction is streamed, then falling again as the peer node commits it, flushes it, and sends confirmation. The reported LSN positions will "stair-step" instead of advancing smoothly, for similar reasons.

When replication is disconnected (`active = 'f'`), the `active_pid` column will be `NULL`, as will `client_addr` and other fields that only make sense with an active connection. The `state` field will be `'disconnected'`. The `_lsn` fields will be the same as the `confirmed_flush_lsn`, since that is the last position that the client is known for certain to have replayed to and saved. The `_lag` fields will show the elapsed time between the most recent confirmed flush on the client and the current time, and the `_lag_size` and `_lag_bytes` fields will report the distance between `confirmed_flush_lsn` and the local server's current WAL insert position.

Note: It is normal for `restart_lsn` to be behind the other `lsn` columns; this does not indicate a problem with replication or a peer node lagging. The `restart_lsn` is the position that PostgreSQL's internal logical decoding must be reading WAL at if interrupted, and generally reflects the position of the oldest transaction that is not yet replicated and flushed. A very old `restart_lsn` can make replication slow

to restart after disconnection and force retention of more WAL than is desirable, but will otherwise be harmless. If you are concerned, look for very long running transactions and forgotten prepared transactions.

Monitoring Incoming Replication

Incoming replication (also called subscription) can be monitored by querying the `bdr.subscription_summary` view. This shows the list of known subscriptions to other nodes in the BDR cluster and the state of the replication worker, e.g.:

```
# SELECT node_group_name, origin_name, sub_enabled, sub_slot_name,
#     subscription_status
# FROM bdr.subscription_summary;
-[ RECORD 1 ]-------+---------------------------
node_group_name     | bdrgroup
origin_name         | node2
sub_enabled         | t
sub_slot_name       | bdr_postgres_bdrgroup_node1
subscription_status | replicating
-[ RECORD 2 ]-------+---------------------------
node_group_name     | bdrgroup
origin_name         | node3
sub_enabled         | t
sub_slot_name       | bdr_postgres_bdrgroup_node1
subscription_status | replicating
```

Monitoring BDR Replication Workers

All BDR workers show up in the system view pg_stat_activity. So this view offers some insight into the state of a BDR system.

The view `bdr.worker_errors` shows errors (if any) reported by any worker. BDR 3.6 depended explicitly on pglogical 3.6 as a separate extension. While pglogical deletes older worker errors, BDR does not aim to, given the additional complexity of bi-directional replication. A side effect of this dependency is that in BDR 3.6 some worker errors are deleted over time, while others are retained indefinitely. Because of this it's important to note the time of the error and not just the existence of one. Starting from BDR 4, there is a single extension, and dependency on pglogical as a separate extension has been removed, meaning that all worker errors are now retained indefinitely.

Monitoring Global Locks

The global lock, which is currently only used for DDL replication, is a heavyweight lock that exists across the whole BDR group.

There are currently two types of global locks:

- DDL lock, used for serializing all DDL operations on permanent (not temporary) objects (i.e. tables) in the database
- DML relation lock, used for locking out writes to relations during DDL operations that change the relation definition

Either or both entry types may be created for the same transaction, depending on the type of DDL operation and the value of the `bdr.ddl_locking` setting.

Global locks held on the local node are visible in the `bdr.global_locks` view. This view shows the type of the lock; for relation locks it shows which relation is being locked, the PID holding the lock (if local), and whether the lock has been globally granted or not.

The following is an example output of `bdr.global_locks` while running an `ALTER TABLE` statement with `bdr.ddl_locking = on`:

```
# SELECT lock_type, relation, pid, granted FROM bdr.global_locks;
-[ RECORD 1 ]--------------
lock_type | GLOBAL_LOCK_DDL
relation  | [NULL]
pid       | 15534
granted   | t
-[ RECORD 2 ]--------------
lock_type | GLOBAL_LOCK_DML
relation  | someschema.sometable
pid       | 15534
granted   | t
```

See the catalog documentation for details on all fields including lock timing information.

Monitoring Conflicts

Replication conflicts can arise when multiple nodes make changes that affect the same rows in ways that can interact with each other. The BDR system should be monitored to ensure that conflicts are identified and, where possible, application changes are made to eliminate them or make them less frequent.

By default, all conflicts are logged to the PostgreSQL server log. It is however possible to change the configuration to log only some conflicts, and to log both to the server log and/or a table.

The configuration of logging is defined by the `bdr.alter_node_add_log_config` and `bdr.alter_node_remove_log_c` functions.

Apply Statistics

BDR collects statistics about replication apply, both for each subscription and for each table.

Two monitoring views exist: `bdr.stat_subscription` for subscription statistics and `bdr.stat_relation` for relation statistics. These views both provide:

- Number of INSERTs/UPDATEs/DELETEs/TRUNCATEs replicated
- Block accesses and cache hit ratio
- Total I/O time for read/write

and for relations only, these statistics:

- Total time spent processing replication for the relation
- Total lock wait time to acquire lock (if any) for the relation (only)

and for subscriptions only, these statistics:

- Number of COMMITs/DDL replicated for the subscription
- Number of times this subscription has connected upstream

Tracking of these statistics is controlled by the pglogical GUCs `pglogical.track_subscription_apply` and `pglogical.track_relation_apply` respectively - for details, see pglogical Settings for BDR.

The example output from these would look like this:

```
# SELECT sub_name, nconnect, ninsert, ncommit, nupdate, ndelete, ntruncate, nddl
FROM pglogical.stat_subscription;
-[ RECORD 1 ]----------------------------------
sub_name  | bdr_regression_bdrgroup_node1_node2
nconnect  | 3
ninsert   | 10
ncommit   | 5
nupdate   | 0
ndelete   | 0
ntruncate | 0
nddl      | 2
```

In this case the subscription connected 3 times to the upstream, inserted 10 rows and did 2 DDL commands inside 5 transactions.

Standard PostgreSQL Statistics Views

Statistics on table and index usage are updated normally by the downstream master. This is essential for the correct function of autovacuum. If there are no local writes on the downstream master and statistics have not been reset, these two views should show corresponding results between upstream and downstream:

- `pg_stat_user_tables`
- `pg_statio_user_tables`

> **Note**
> We don't necessarily expect the upstream table statistics to be *similar* to the downstream ones; we only expect them to *change* by the same amounts. Consider the example of a table whose statistics show 1M inserts and 1M updates; when a new node joins the BDR group, the statistics for the same table in the new node will show 1M inserts and zero updates. However, from that moment, the upstream and downstream table statistics will change by the same amounts, because all changes on one side will be replicated to the other side.

Since indexes are used to apply changes, the identifying indexes on the downstream side may appear more heavily used with workloads that perform `UPDATE`s and `DELETE`s than non-identifying indexes are.

The built-in index monitoring views are:

- `pg_stat_user_indexes`
- `pg_statio_user_indexes`

All these views are discussed in detail in the PostgreSQL documentation on the statistics views.

Monitoring BDR Versions

BDR allows running different Postgres versions as well as different BDR versions across the nodes in the same cluster. This is useful for upgrading.

The view `bdr.monitor_group_versions_details` uses the function `bdr.run_on_all_nodes()` to retrieve BDR version, edition, and pglogical version from all nodes at the same time. For example:

```
bdrdb=# SELECT node_name, postgres_version, pglogical_version,
bdr_version FROM bdr.monitor_group_versions_details;
 node_name | postgres_version | pglogical_version | bdr_version
-----------+------------------+-------------------+-------------
 node1     | 11.12            | 3.6.26            | 3.6.26
 node2     | 11.12            | 3.6.26            | 3.6.26
```

BDR and pglogical versions should be the same in the same node. It is recommended that the cluster does not run different BDR versions for too long. The recommended setup is to try to have all nodes running the same latest BDR version as soon as possible.

For monitoring purposes, we recommend the following alert levels:

- status=UNKNOWN, message=This node is not part of any BDR group
- status=OK, message=All nodes are running same pglogical and BDR versions
- status=WARNING, message=There is at least 1 node that is not accessible
- status=WARNING, message=There are node(s) running different BDR versions when compared to other nodes

- status=WARNING, message=There are node(s) running different BDR editions when compared to other nodes

The described behavior is implemented in the function `bdr.monitor_group_versions()`, which uses BDR/pglogical version information returned from the view `bdr.monitor_group_version_details` to provide a cluster-wide version check. For example:

```
bdrdb=# SELECT * FROM bdr.monitor_group_versions();
 status |                         message
--------+-----------------------------------------------------
 OK     | All nodes are running same pglogical and BDR versions
```

Monitoring Raft Consensus

Raft Consensus should be working cluster-wide at all times. The impact of running a BDR cluster without Raft Consensus working might be as follows:

- BDR replication might still be working correctly
- Global DDL/DML locks will not work
- Galloc sequences will eventually run out of chunks
- Eager Replication (EE only) will not work
- Cluster maintenance operations (join node, part node, promote standby) are still allowed but they might not finish (simply hang)
- Node statuses might not be correctly synced among the BDR nodes
- BDR group replication slot does not advance LSN, thus keeps WAL files on disk

The view `bdr.monitor_group_raft_details` uses the functions `bdr.run_on_all_nodes()` and `bdr.get_raft_status()` to retrieve Raft Consensus status from all nodes at the same time. For example:

```
bdrdb=# SELECT node_id, node_name, state, leader_id
FROM bdr.monitor_group_raft_details;
  node_id   | node_name |     state      | leader_id
------------+-----------+----------------+------------
 1148549230 | node1     | RAFT_LEADER    | 1148549230
 3367056606 | node2     | RAFT_FOLLOWER | 1148549230
```

We can say that Raft Consensus is working correctly if all below conditions are met:

- A valid state (`RAFT_LEADER` or `RAFT_FOLLOWER`) is defined on all nodes
- Only one of the nodes is the `RAFT_LEADER`
- The `leader_id` is the same on all rows and must match the `node_id` of the row where `state = RAFT_LEADER`

From time to time, Raft Consensus will start a new election to define a new `RAFT_LEADER`. During an election, there might be an intermediary situation where there is no `RAFT_LEADER` and some of the nodes consider themselves as `RAFT_CANDIDATE`. The whole election should not take longer than `bdr.raft_election_timeout` (by default it is set to 6 seconds). If the query above returns an in-election situation, then simply wait for `bdr.raft_election_timeout` and run the query again. If after `bdr.raft_election_timeout` has passed and some the conditions above are still not met, then Raft Consensus is not working.

Raft Consensus might not be working correctly on a single node only; for example one of the nodes does not recognize the current leader and considers itself as a `RAFT_CANDIDATE`. In this case, it is important to make sure that:

- All BDR nodes are accessible to each other through both regular and replication connections (check file `pg_hba.conf`)
- BDR and pglogical versions are the same on all nodes
- `bdr.raft_election_timeout` is the same on all nodes

In some cases, especially if nodes are geographically distant from each other and/or network latency is high, the default value of `bdr.raft_election_timeout` (6 seconds) might not be enough. If Raft Consensus is still not working even after making sure everything is correct, consider increasing `bdr.raft_election_timeout` to, say, 30 seconds on all nodes. From BDR 3.6.11 onwards, setting `bdr.raft_election_timeout` requires only a server reload.

Given how Raft Consensus affects cluster operational tasks, and also as Raft Consensus is directly responsible for advancing the group slot, we can define monitoring alert levels as follows:

- status=UNKNOWN, message=This node is not part of any BDR group
- status=OK, message=Raft Consensus is working correctly
- status=WARNING, message=There is at least 1 node that is not accessible
- status=WARNING, message=There are node(s) as RAFT_CANDIDATE, an election might be in progress
- status=WARNING, message=There is no RAFT_LEADER, an election might be in progress
- status=CRITICAL, message=There is a single node in Raft Consensus
- status=CRITICAL, message=There are node(s) as RAFT_CANDIDATE while a RAFT_LEADER is defined
- status=CRITICAL, message=There are node(s) following a leader different than the node set as RAFT_LEADER

The described behavior is implemented in the function `bdr.monitor_group_raft()`, which uses Raft Consensus status information returned from the view `bdr.monitor_group_raft_details` to provide a cluster-wide Raft check. For example:

```
bdrdb=# SELECT * FROM bdr.monitor_group_raft();
 status |               message
--------+-----------------------------------
 OK     | Raft Consensus is working correctly
```

Monitoring Replication Slots

Each BDR node keeps:

- One replication slot per active BDR peer
- One group replication slot

For example:

```
bdrdb=# SELECT slot_name, database, active, confirmed_flush_lsn
FROM pg_replication_slots ORDER BY slot_name;
        slot_name         | database | active | confirmed_flush_lsn
--------------------------+----------+--------+---------------------
 bdr_bdrdb_bdrgroup       | bdrdb    | f      | 0/3110A08
 bdr_bdrdb_bdrgroup_node2 | bdrdb    | t      | 0/31F4670
 bdr_bdrdb_bdrgroup_node3 | bdrdb    | t      | 0/31F4670
 bdr_bdrdb_bdrgroup_node4 | bdrdb    | t      | 0/31F4670
```

Peer slot names follow the convention `bdr_<DATABASE>_<GROUP>_<PEER>`, while the BDR group slot name follows the convention `bdr_<DATABASE>_<GROUP>`.

Peer replication slots should be active on all nodes at all times. If a peer replication slot is not active, then it might mean:

- The corresponding peer is shutdown or not accessible; or
- BDR replication is broken. Grep the log file for `ERROR` or `FATAL` and also check `bdr.worker_errors` on all nodes. The root cause might be, for example, an incompatible DDL was executed with DDL replication disabled on one of the nodes.

The BDR group replication slot, on the other hand, is inactive most of the time. BDR keeps this slot and advances LSN, as all other peers have already consumed the corresponding transactions. So it is not possible to monitor the status (active or inactive) of the group slot.

We recommend the following monitoring alert levels:

- status=UNKNOWN, message=This node is not part of any BDR group
- status=OK, message=All BDR replication slots are working correctly
- status=CRITICAL, message=There is at least 1 BDR replication slot which is inactive
- status=CRITICAL, message=There is at least 1 BDR replication slot which is missing

The described behavior is implemented in the function `bdr.monitor_local_replslots()`, which uses replication slot status information returned from view `bdr.node_slots` (slot active or inactive) to provide a local check considering all BDR node replication slots, except the BDR group slot.

```
bdrdb=# SELECT * FROM bdr.monitor_local_replslots();
 status |                    message
--------+-----------------------------------------------
 OK     | All BDR replication slots are working correctly
```

Tracing Transaction COMMITs

By default, BDR transactions commit only on the local node. In that case, transaction `COMMIT` will be processed quickly.

BDR can be used with standard PostgreSQL synchronous replication, while BDR also provides two new transaction commit modes: CAMO and Eager replication. Each of these modes provides additional robustness features, though at the expense of additional latency at `COMMIT`. The additional time at `COMMIT` can be monitored dynamically using the `pg_stat_activity` catalog, where processes report different `wait_event` states. A transaction in `COMMIT` waiting for confirmations from one or more synchronous standbys reports a `SyncRep` wait event, whereas the two new modes report `EagerRep`.

Also, transaction `COMMIT` can be traced in more detail using the `tracelog_timeout` parameter. If the `COMMIT` takes longer than this time, additional log messages will show what steps occurred at which point. All such messages are prefixed with `TRACELOG:` and are disabled until the timeout is reached. Thus the first line per transaction describes the activity during which the timeout triggered. Timestamps on the following log lines allow us to determine the duration of all following actions during the `COMMIT`. As an example, the tracelog lines for a transaction that took too long to write to disk look similar to:

```
LOG:  00000: TRACELOG: Flushed WAL to disk
LOG:  00000: TRACELOG: End of critical section for commit
LOG:  00000: TRACELOG: Finished waiting for SyncRep
LOG:  00000: TRACELOG: XIDs marked as committed in pg_xact
LOG:  00000: TRACELOG: Marked transaction as no longer running
LOG:  00000: TRACELOG: Commit XactCallbacks fired
LOG:  00000: TRACELOG: Released before locks resources
LOG:  00000: TRACELOG: Released all buffer pins
LOG:  00000: TRACELOG: Cleaned up relation cache
LOG:  00000: TRACELOG: Catalog changes visible to all backends
LOG:  00000: TRACELOG: Transaction end for multiXact
```

Note that this can cause substantial additional logs, so it should be enabled with care. If used in production, it should be set to catch outliers. It is not intended for regular performance monitoring. Please start with high timeouts and decrease, until a useful amount of log is available for analysis, to minimize its impact on performance.

Backup and Recovery

In this chapter we discuss the backup and restore of a BDR 3.x cluster.

BDR is designed to be a distributed, highly available system. If one or more nodes of a cluster are lost, the best way to replace them is to clone new nodes directly from the remaining nodes.

The role of backup and recovery in BDR is to provide for Disaster Recovery (DR), such as in the following situations:

- Loss of all nodes in the cluster
- Significant, uncorrectable data corruption across multiple nodes as a result of data corruption, application error or security breach

Backup

`pg_dump`

`pg_dump`, sometimes referred to as "logical backup", can be used normally with BDR.

Note that `pg_dump` dumps both local and global sequences as if they were local sequences. This is intentional, to allow a BDR schema to be dumped and ported to other PostgreSQL databases. This means that sequence kind metadata is lost at the time of dump, so a dump and restore would effectively reset all sequence kinds to the value of `bdr.default_sequence_kind`.

2ndQuadrant recommends the use of physical backup techniques for backup and recovery.

Physical Backup

Physical backups of a node in a BDR cluster can be taken using standard PostgreSQL software, such as Barman.

A physical backup of a BDR node can be performed with the same procedure that applies to any PostgreSQL node: a BDR node is just a PostgreSQL node running the BDR extension.

There are some specific points that must be considered when applying PostgreSQL backup techniques to BDR:

- BDR operates at the level of a single database, while a physical backup includes all the databases in the instance; you should plan your databases to allow them to be easily backed-up and restored.
- Backups will make a copy of just one node. In the simplest case, every node has a copy of all data, so you would need to backup only one node to capture all data. However, the goal of DR will not be met if the site containing that single copy goes down, so the minimum should be at least one node backup per site (obviously with many copies etc.).

- However, each node may have un-replicated local data, and/or the definition of replication sets may be complex so that all nodes do not subscribe to all replication sets. In these cases, backup planning must also include plans for how to backup any unreplicated local data and a backup of at least one node that subscribes to each replication set.

Eventual Consistency

The nodes in a BDR cluster are *eventually consistent*, but not entirely *consistent*; a physical backup of a given node will provide Point-In-Time Recovery capabilities limited to the states actually assumed by that node (see the Example below).

The following example shows how two nodes in the same BDR cluster might not (and usually do not) go through the same sequence of states.

Consider a cluster with two nodes `N1` and `N2`, which is initially in state `S`. If transaction `W1` is applied to node `N1`, and at the same time a non-conflicting transaction `W2` is applied to node `N2`, then node `N1` will go through the following states:

```
(N1)   S  -->  S + W1  -->  S + W1 + W2
```

. . . while node `N2` will go through the following states:

```
(N2)   S  -->  S + W2  -->  S + W1 + W2
```

That is: node `N1` will *never* assume state `S + W2`, and node `N2` likewise will never assume state `S + W1`, but both nodes will end up in the same state `S + W1 + W2`. Considering this situation might affect how you decide upon your backup strategy.

Point-In-Time Recovery (PITR)

In the example above, the changes are also inconsistent in time, since `W1` and `W2` both occur at time `T1`, but the change `W1` is not applied to `N2` until `T2`.

PostgreSQL PITR is designed around the assumption of changes arriving from a single master in COMMIT order. Thus, PITR is possible by simply scanning through changes until one particular point-in-time (PIT) is reached. With this scheme, you can restore one node to a single point-in-time from its viewpoint, e.g. `T1`, but that state would not include other data from other nodes that had committed near that time but had not yet arrived on the node. As a result, the recovery might be considered to be partially inconsistent, or at least consistent for only one replication origin.

To request this, use the standard syntax:

```
recovery_target_time = T1
```

BDR allows for changes from multiple masters, all recorded within the WAL log for one node, separately identified using replication origin identifiers.

BDR allows PITR of all or some replication origins to a specific point in time, providing a fully consistent viewpoint across all subsets of nodes. Thus for multi-origins, we view the WAL stream as containing

multiple streams all mixed up into one larger stream. There is still just one PIT, but that will be reached as different points for each origin separately.

We read the WAL stream until requested origins have found their PIT. We apply all changes up until that point, except that we do not mark as committed any transaction records for an origin after the PIT on that origin has been reached.

We end up with one LSN "stopping point" in WAL, but we also have one single timestamp applied consistently, just as we do with "single origin PITR".

In the specific example above, `N1` would be restored to `T1`, but would also include changes from other nodes that have been committed by `T1`, even though they were not applied on `N1` until later.

To request multi-origin PITR, use the standard syntax in the recovery.conf file:

```
recovery_target_time = T1
```

The list of replication origins which would be restored to `T1` need either to be specified in a separate multi_recovery.conf file via the use of a new parameter `recovery_target_origins`:

```
recovery_target_origins = '*'
```

...or one can specify the origin subset as a list in `recovery_target_origins`.

```
recovery_target_origins = '1,3'
```

Note that the local WAL activity recovery to the specified `recovery_target_time` is always performed implicitly. For origins that are not specified in `recovery_target_origins`, recovery may stop at any point, depending on when the target for the list mentioned in `recovery_target_origins` is achieved.

In the absence of the `multi_recovery.conf` file, the recovery defaults to the original PostgreSQL PITR behaviour that is designed around the assumption of changes arriving from a single master in COMMIT order.

Note that Barman does not yet automatically create a `multi_recovery.conf` file.

Restore

While you can take a physical backup with the same procedure as a standard PostgreSQL node, what is slightly more complex is **restoring** the physical backup of a BDR node.

BDR Cluster Failure or Seeding a New Cluster from a Backup

The most common use case for restoring a physical backup involves the failure or replacement of all the BDR nodes in a cluster, for instance in the event of a datacentre failure.

You may also want to perform this procedure to clone the current contents of a BDR cluster to seed a QA or development instance.

In that case, BDR capabilities can be restored based on a physical backup of a single BDR node, optionally plus WAL archives:

- If you still have some BDR nodes live and running, fence off the host you restored the BDR node to, so it cannot connect to any surviving BDR nodes. This ensures that the new node does not confuse the existing cluster.
- Restore a single PostgreSQL node from a physical backup of one of the BDR nodes.
- If you have WAL archives associated with the backup, create a suitable `recovery.conf` and start PostgreSQL in recovery to replay up to the latest state. You can specify a alternative `recovery_target` here if needed.
- Start the restored node, or promote it to read/write if it was in standby recovery. Keep it fenced from any surviving nodes!
- Clean up any leftover BDR metadata that was included in the physical backup, as described below.
- Fully stop and restart the PostgreSQL instance.
- Add further BDR nodes with the standard procedure based on the `bdr.join_node_group()` function call.

Cleanup BDR Metadata

The cleaning of leftover BDR metadata is achieved as follows:

1. Drop the BDR node using `bdr.drop_node`
2. Fully stop and re-start PostgreSQL (important!).

Cleanup of Replication Origins

Replication origins must be explicitly removed with a separate step because they are recorded persistently in a system catalog, and therefore included in the backup and in the restored instance. They are not removed automatically when dropping the BDR extension, because they are not explicitly recorded as its dependencies.

BDR creates one replication origin for each remote master node, to track progress of incoming replication in a crash-safe way. Therefore we need to run:

```
SELECT pg_replication_origin_drop('bdr_dbname_grpname_nodename');
```

. . . once for each node in the (previous) cluster. Replication origins can be listed as follows:

```
SELECT * FROM pg_replication_origin;
```

. . . and those created by BDR are easily recognized by their name, as in the example shown above.

Cleanup of Replication Slots

If a physical backup was created with `pg_basebackup`, replication slots will be omitted from the backup.

Some other backup methods may preserve replications slots, likely in outdated or invalid states. Once you restore the backup, just:

```
SELECT pg_replication_slot_drop(slot_name)
FROM pg_replication_slots;
```

. . . to drop *all* replication slots. If you have a reason to preserve some, you can add a `WHERE slot_name LIKE 'bdr%'` clause, but this is rarely useful.

> **Warning**
> Never run this on a live BDR node.

Upgrading

In this chapter we discuss upgrading the BDR 3.x cluster and how to minimize downtime for applications during the upgrade.

Database Encoding

We recommend using `UTF-8` encoding in all replicated databases. BDR does not support replication between databases with different encoding. There is currently no supported path to upgrade/alter encoding.

Server Software Upgrade

The upgrade of BDR software on individual nodes happens in-place. There is no need for backup and restore when upgrading the BDR extension.

The first step in the upgrade is to install the new version of the BDR packages, which will install both the new binary and the extension SQL script. This step depends on the operating system used.

Upgrading the binary and extension scripts by itself does not upgrade BDR in the running instance of PostgreSQL. To do that, the PostgreSQL instance needs to be restarted so that the new BDR binary can be loaded (the BDR binary is loaded at the start of the PostgreSQL server). After that, the node is upgraded. The extension SQL upgrade scripts are executed automatically as needed.

> **Warning**
> It's important to never run the `ALTER EXTENSION ... UPDATE` command before the PostgreSQL instance is restarted, as that will only upgrade the SQL-visible extension but keep the old binary, which can cause unpredictable behaviour or even crashes. The `ALTER EXTENSION ... UPDATE` command should never be needed; BDR3 maintains the SQL-visible extension automatically as needed.

After this procedure, your BDR node is upgraded. You can verify the current version of BDR3 binary like this:

```
SELECT bdr.bdr_version();
```

The upgrade of BDR3 will usually also upgrade the version of pglogical 3 installed in the system. The current version of pglogical can be checked using:

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

Always check the monitoring after upgrade of a node to confirm that the upgraded node is working as expected.

Rolling Server Software Upgrades

BDR3 supports rolling upgrades from version 3.5 onwards. A rolling upgrade is the process where the above Server Software Upgrade are done on each node in the BDR Group one by one, while keeping the replication working.

During the upgrade process, the application can be switched over to a node which is currently not being upgraded to provide continuous availability of the BDR group for applications.

While the cluster is going through a rolling upgrade, replication happens between mixed versions of BDR3. For example, nodeA will have BDR 3.5.3, while nodeB and nodeC will have 3.6.21. In this state, the replication and group management will use the protocol and features from the oldest version (3.5.3 in case of this example), so any new features provided by the newer version which require changes in the protocol will be disabled. Once all nodes are upgraded to the same version, the new features are automatically enabled.

A BDR cluster is designed to be easily upgradeable. Most BDR releases support rolling upgrades, which means running part of the cluster on one release level and the remaining part of the cluster on a second, compatible, release level.

An rolling upgrade starts with a cluster with all nodes at a prior release, then proceeds by upgrading one node at a time to the newer release, until all nodes are at the newer release. Should problems occur, do not attempt to downgrade without contacting 2ndQuadrant support to discuss and provide options.

An upgrade process may take an extended period of time when the user decides caution is required to reduce business risk, though this should not take any longer than 30 days without discussion and explicit agreement from 2ndQuadrant Support to extend the period of coexistence of two release levels.

In case of problems during upgrade, do not initiate a second upgrade to a newer/different release level. Two upgrades should never occur concurrently in normal usage. Nodes should never be upgraded to a third release without specific and explicit instructions from 2ndQuadrant Support. A case where that might occur is if an upgrade failed for some reason and a Hot Fix was required to continue the current cluster upgrade process to successful conclusion. BDR has been designed and tested with more than 2 release levels, but this cannot be relied upon for production usage except in specific cases.

Upgrading a CAMO-Enabled cluster

CAMO protection requires at least one of the nodes of a CAMO pair to be operational. For upgrades, we recommend to ensure that no CAMO protected transactions are running concurrent to the upgrade, or use a rolling upgrade strategy, giving the nodes enough time to reconcile in between the upgrades and the corresponding node downtime due to the upgrade.

With 3.6.11, a third synchronization level for CAMO has been introduced, so there now are `remote_write`, `remote_commit_async`, and `remote_commit_flush`. The value `on` is supported for backwards compatibility and defaults to the safest, but slowest choice: `remote_commit_flush`. This may lead to increased transaction commit times compared to 3.6.10, where `on` was equivalent to the mode now known as `remote_commit_async`.

From 3.6.6 to 3.6.7, the internals of CAMO changed to use two-phase commit. Nodes that run CAMO-protected transactions therefore need to have a `max_prepared_transactions` set high enough to cover the maximum number of concurrent-prepared transactions at any point in time, possibly caused by CAMO, Eager or explicit two-phase commit. Ensure `max_prepared_transactions` is set prior to restart/upgrade.

From 3.5.3 to 3.6.0, configuration of CAMO has been simplified and checks have been added to prevent misconfiguration. To upgrade a BDR cluster with CAMO pairs, please disable CAMO, upgrade, and then re-configure using the following new or changed settings:

- `bdr.camo_partner_of` and `bdr.camo_origin_for` need to be configured on both nodes of a CAMO pair. This replaces the former requirement to set the application name in `synchronous_standby_names`. This separates CAMO configuration from the configuration of physical standbys and simplifies configuration, as the new settings accept BDR node names.
- `synchronous_commit` is no longer required to be set to `remote_apply`. Instead CAMO internally uses a mode equivalent to `remote_write` whenever it is enabled, independent of the `synchronous_commit` setting (note that this also means that setting `synchronous_commit = 'local'` does no longer disable CAMO).
- `pg2q.enable_camo` has been added as an alias with simpler on/off values in place of `synchronous_replication_timing` and is now recommended. The previous setting remains compatible and setting it to `remote_first` has the same effect as setting `pg2q.enable_camo = on`.
- `bdr.global_commit_timeout` now allows the abort of transactions with CAMO protection earlier than `wal_sender_timeout`, which it may not be feasible to set to very short intervals, see CAMO Local Mode for details.

Rolling Application Schema Upgrades

By default, DDL will automatically be sent to all nodes. This can be controlled manually, as described in DDL Replication, which could be used to create differences between database schemas across nodes. BDR is designed to allow replication to continue even while minor differences exist between nodes. These features are designed to allow application schema migration without downtime, or to allow logical standby nodes for reporting or testing.

> **Warning**
> Application Schema Upgrades are managed by the user, not by BDR. Careful scripting will be required to make this work correctly on production clusters. Extensive testing is advised.

Details of this are covered here Replicating between nodes with differences.

When one node runs DDL that adds a new table, nodes that have not yet received the latest DDL will need to cope with the extra table. In view of this, the appropriate setting for rolling schema upgrades is to

configure all nodes to apply the `skip` resolver in case of a `target_table_missing` conflict. This must be performed before any node has additional tables added, and is intended to be a permanent setting.

This is done with the following query, that must be **executed separately on each node**, after replacing `node1` with the actual node name:

```
SELECT bdr.alter_node_set_conflict_resolver('node1',
        'target_table_missing', 'skip');
```

When one node runs DDL that adds a column to a table, nodes that have not yet received the latest DDL will need to cope with the extra columns. In view of this, the appropriate setting for rolling schema upgrades is to configure all nodes to apply the `ignore` resolver in case of a `target_column_missing` conflict. This must be performed before one node has additional columns added and is intended to be a permanent setting.

This is done with the following query, that must be **executed separately on each node**, after replacing `node1` with the actual node name:

```
SELECT bdr.alter_node_set_conflict_resolver('node1',
        'target_column_missing', 'ignore');
```

When one node runs DDL that removes a column from a table, nodes that have not yet received the latest DDL will need to cope with the missing column. This situation will cause a `source_column_missing` conflict, which uses the `use_default_value` resolver. Thus, columns that neither accept NULLs nor have a DEFAULT value will require a two step process:

1. Remove NOT NULL constraint or add a DEFAULT value for a column on all nodes.
2. Remove the column.

Constraints can be removed in a rolling manner. There is currently no supported way for coping with adding table constraints in a rolling manner, one node at a time.

When one node runs a DDL that changes the type of an existing column, depending on the existence of binary coercibility between the current type and the target type, the operation may not rewrite the underlying table data. In that case, it will be only a metadata update of the underlying column type. Rewrite of a table is normally restricted. However, in controlled DBA environments, it is possible to change the type of a column to an automatically castable one by adopting a rolling upgrade for the type of this column in a non-replicated environment on all the nodes, one by one. More details are provided in the ALTER TABLE section.

Explicit Two-Phase Commit (2PC)

An application may opt to use two-phase commit explicitly with BDR. See Distributed Transaction Processing: The XA Specification.

The X/Open Distributed Transaction Processing (DTP) model envisages three software components:

- An application program (AP) that defines transaction boundaries and specifies actions that constitute a transaction.
- Resource managers (RMs, such as databases or file access systems) that provide access to shared resources.
- A separate component called a transaction manager (TM) that assigns identifiers to transactions, monitors their progress, and takes responsibility for transaction completion and for failure recovery.

BDR supports explicit external 2PC using the PREPARE TRANSACTION and COMMIT PREPARED/ROLLBACK PREPARED commands. Externally, a BDR cluster appears to be a single Resource Manager to the Transaction Manager for a single session.

When `bdr.commit_scope` is `local`, the transaction is prepared only on the local node. Once committed, changes will be replicated, and BDR then applies post-commit conflict resolution.

Using `bdr.commit_scope` set to `local` may seem nonsensical with explicit two-phase commit, but the option is offered to allow the user to control the trade-off between transaction latency and robustness.

Explicit two-phase commit does not work in combination with either CAMO or the global commit scope. Future releases may enable this combination.

Usage

Two-phase commits with a local commit scope work exactly like standard PostgreSQL. Please use the local commit scope and disable CAMO.

```
BEGIN;

SET LOCAL pg2q.enable_camo = 'off';
SET LOCAL bdr.commit_scope = 'local';

... other commands possible...
```

To start the first phase of the commit, the client must assign a global transaction id, which can be any unique string identifying the transaction:

```
PREPARE TRANSACTION 'some-global-id';
```

After a successful first phase, all nodes have applied the changes and are prepared for committing the transaction. The client must then invoke the second phase from the same node:

```
COMMIT PREPARED 'some-global-id';
```

Catalogs and Views

This section contains a listing of system catalogs and views used by BDR in alphabetical order.

bdr.apply_log

This table provides all the columns that are recognized when logging row conflict details to a table.

It can be used either as a target for row-level conflict detail logging, or just as an example of which columns are supported.

bdr.apply_log Columns

The columns of bdr.apply_log are documented in Logging to a Table, together with more details on row conflict logging.

bdr.apply_log_summary

A view containing user-readable details on row conflict.

bdr.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

bdr.camo_decision_journal

A persistent journal of decisions resolved by a CAMO partner node after a failover, in case bdr.logical_transaction_status got invoked. Unlike bdr.node_pre_commit, this does not cover transactions processed under normal operational conditions (i.e. both nodes of a CAMO pair are running and connected). Entries in this journal are not ever cleaned up automatically. This is a purely diagnostic tool that the system does not depend on in any way.

bdr.camo_decision_journal Columns

Name	Type	Description
origin_node_id	oid	OID of the node where the transaction executed
origin_xid	oid	Transaction Id on the remote origin node
decision	char	'c' for commit, 'a' for abort
decision_ts	timestamptz	Decision time

bdr.crdt_handlers

This table lists merge ("handlers") functions for all CRDT data types.

bdr.crdt_handlers Columns

Name	Type	Description
crdt_type_id	regtype	CRDT data type id
crdt_merge_id	regproc	Merge function for this data type

bdr.ddl_epoch

An internal catalog table holding state per DDL epoch.

bdr.ddl_epoch Columns

Name	Type	Description
ddl_epoch	int8	Monotonically increasing epoch number
origin_node_id	oid	Internal node id of the node that requested creation of this epoch
epoch_consume_timeout	timestamptz	Timeout of this epoch
epoch_consumed	boolean	Switches to true as soon as the local node has fully processed the epoch

bdr.ddl_replication

This view lists DDL replication configuration as set up by current DDL filters.

bdr.ddl_replication Columns

Name	Type	Description
set_ddl_name	name	Name of DDL filter
set_ddl_tag	text	Which command tags it applies on (regular expression)

Name	Type	Description
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

bdr.global_consensus_journal

This catalog table logs all the Raft messages that were sent while managing global consensus.

As for the `bdr.global_consensus_response_journal` catalog, the payload is stored in a binary encoded format, which can be decoded with the `bdr.decode_message_payload()` function; see the `bdr.global_consensus_journal_details` view for more details.

`bdr.global_consensus_journal` Columns

Name	Type	Description
log_index	int8	Id of the journal entry
term	int8	Raft term
origin	oid	Id of node where the request originated
req_id	int8	Id for the request
req_payload	bytea	Payload for the request
trace_context	bytea	Trace context for the request

bdr.global_consensus_journal_details

This view presents Raft messages that were sent, and the corresponding responses, using the `bdr.decode_message_payload()` function to decode their payloads.

`bdr.global_consensus_journal_details` Columns

Name	Type	Description
log_index	int8	Id of the journal entry
term	int8	Raft term
request_id	int8	Id of the request
origin_id	oid	Id of the node where the request originated
req_payload	bytea	Payload of the request
origin_node_name	name	Name of the node where the request originated
message_type_no	oid	Id of the BDR message type for the request
message_type	text	Name of the BDR message type for the request
message_payload	text	BDR message payload for the request

Name	Type	Description
response_message_type_no	oid	Id of the BDR message type for the response
response_message_type	text	Name of the BDR message type for the response
response_payload	text	BDR message payload for the response
response_errcode_no	text	SQLSTATE for the response
response_errcode	text	Error code for the response
response_message	text	Error message for the response

bdr.global_consensus_response_journal

This catalog table collects all the responses to the Raft messages that were received while managing global consensus.

As for the `bdr.global_consensus_journal` catalog, the payload is stored in a binary-encoded format, which can be decoded with the `bdr.decode_message_payload()` function; see the `bdr.global_consensus_journal_details` view for more details.

bdr.global_consensus_response_journal Columns

Name	Type	Description
log_index	int8	Id of the journal entry
res_status	oid	Status code for the response
res_payload	bytea	Payload for the response
trace_context	bytea	Trace context for the response

bdr.global_lock

This catalog table stores the information needed for recovering the global lock state on server restart.

For monitoring usage, operators should prefer the `bdr.global_locks` view, because the visible rows in `bdr.global_lock` do not necessarily reflect all global locking activity.

Do not modify the contents of this table: it is an important BDR catalog.

bdr.global_lock Columns

Name	Type	Description
ddl_epoch	int8	DDL epoch for the lock
origin_node_id	oid	OID of the node where the global lock has originated
lock_type	oid	Type of the lock (DDL or DML)
nspname	name	Schema name for the locked relation

Name	Type	Description
relname	name	Relation name for the locked relation

bdr.global_locks

A view containing active global locks on this node. The `bdr.global_locks` view exposes BDR's shared-memory lock state tracking, giving administrators a greater insight into BDR's global locking activity and progress.

See Monitoring Global Locks for more information about global locking.

`bdr.global_locks` Columns

Name	Type	Description
`origin_node_id`	oid	The OID of the node where the global lock has originated
`origin_node_name`	name	Name of the node where the global lock has originated
`lock_type`	text	Type of the lock (DDL or DML)
`relation`	text	Locked relation name (for DML locks)
`pid`	int4	PID of the process holding the lock
`acquire_stage`	text	Internal state of the lock acquisition process
`waiters`	int4	List of backends waiting for the same global lock
`global_lock_request_time`	timestamptz	Time this global lock acquire was initiated by origin node
`local_lock_request_time`	timestamptz	Time the local node started trying to acquire the local-lock
`last_state_change_time`	timestamptz	Time `acquire_stage` last changed

Column details:

- `origin_node_id` and `origin_node_name`: If these are the same as the local node's ID and name, then the local node is the initiator of the global DDL lock, i.e. it is the node running the acquiring transaction. If these fields specify a different node, then the local node is instead trying to acquire its local DDL lock to satisfy a global DDL lock request from a remote node.
- `pid`: The process ID of the process that requested the global DDL lock, if the local node is the requesting node. Null on other nodes; query the origin node to determine the locker pid.

- `global_lock_request_time`: The timestamp at which the global-lock request initiator started the process of acquiring a global lock. May be null if unknown on the current node. This time is stamped at the very beginning of the DDL lock request, and includes the time taken for DDL epoch management and any required flushes of pending-replication queues. Currently only known on origin node.
- `local_lock_request_time`: The timestamp at which the local node started trying to acquire the local lock for this global lock. This includes the time taken for the heavyweight session lock acquire, but does NOT include any time taken on DDL epochs or queue flushing. If the lock is re-acquired after local node restart, this will be the node restart time.
- `last_state_change_time`: The timestamp at which the `bdr.global_locks.acquire_stage` field last changed for this global lock entry.

bdr.local_consensus_snapshot

This catalog table contains consensus snapshots created or received by the local node.

`bdr.local_consensus_snapshot` Columns

Name	Type	Description
log_index	int8	Id of the journal entry
log_term	int8	Raft term
snapshot	bytea	Raft snapshot data

bdr.local_consensus_state

This catalog table stores the current state of Raft on the local node.

`bdr.local_consensus_state` Columns

Name	Type	Description
node_id	oid	Id of the node
current_term	int8	Raft term
apply_index	int8	Raft apply index
voted_for	oid	Vote cast by this node in this term
last_known_leader	oid	node_id of last known Raft leader

bdr.local_node_summary

A view containing the same information as `bdr.node_summary` but only for the local node.

bdr.node

This table lists all the BDR nodes in the cluster.

bdr.node Columns

Name	Type	Description
node_id	oid	Id of the node
node_group_id	oid	Id of the node group
source_node_id	oid	Id of the source node
node_state	oid	Consistent state of the node
target_state	oid	State that the node is trying to reach (during join or promotion)
seq_id	int4	Sequence identifier of the node used for generating unique sequence numbers
dbname	name	Database name of the node
proto_version_min	int2	Minimum protocol version supported by the node
proto_version_max	int2	Maximum protocol version supported by the node
synchronize_structure	“char”	Schema synchronization done during the join

bdr.node_catchup_info

This catalog table records relevant catch-up information on each node, either if it is related to the join or part procedure.

bdr.node_catchup_info Columns

Name	Type	Description
node_id	oid	Id of the node
node_source_id	oid	Id of the node used as source for the data
slot_name	name	Slot used for this source
min_node_lsn	pg_lsn	Minimum LSN at which the node can switch to direct replay from a peer node
catchup_state	oid	Status code of the catchup state
origin_node_id	oid	Id of the node from which we want transactions

If a node(node_id) needs missing data from a parting node(origin_node_id), it can get it from a node that already has it(node_source_id) via forwarding. The records in this table will persist until the node(node_id) is a member of the BDR cluster.

bdr.node_conflict_resolvers

Currently configured conflict resolution for all known conflict types.

bdr.node_conflict_resolvers Columns

Name	Type	Description
conflict_type	text	Type of the conflict
conflict_resolver	text	Resolver used for this conflict type

bdr.node_group

This catalog table lists all the BDR node groups.

bdr.node_group Columns

Name	Type	Description
node_group_id	oid	ID of the node group
node_group_name	name	Name of the node group
node_group_default_repset	oid	Default replication set for this node group
node_group_insert_to_update	bool	On conflict, whether INSERT should be converted to UPDATE
node_group_update_to_insert	bool	On conflict, whether UPDATE should be converted to INSERT
node_group_ignore_redundant_updates	bool	Whether an UPDATE that does not change any value can be ignored
node_group_check_full_tuple	bool	Whether on UPDATE all attributes should be used and compared; requires REPLICA IDENTITY FULL
node_group_apply_delay	interval	How long a subscriber waits before applying changes from the provider
node_group_check_constraints	bool	Whether the apply process should check constraints when applying data

bdr.node_group_replication_sets

A view showing replication sets used by the BDR group. See also `bdr.replication_sets`.

bdr.node_group_replication_sets Columns

Name	Type	Description
node_group_name	name	Name of the BDR group
node_name	name	Name of the local node
set_name	name	Name of the replication set

bdr.node_local_info

A catalog table used to store per-node information that changes less frequently than peer progress.

bdr.node_local_info Columns

Name	Type	Description
node_id	oid	The OID of the node (including the local node)
applied_state	oid	Internal id of the node state
ddl_epoch	int8	Last epoch number processed by the node
replication_sets	text[]	List of replication sets subscribed to (only for the local node)
slot_name	name	Name of the slot used to connect to that node (NULL for the local node)

bdr.node_log_config

A catalog view that stores information on the conflict logging configurations.

bdr.node_log_config Columns

Name	Description
log_name	name of the logging configuration
log_to_file	whether it logs to the server log file
log_to_table	whether it logs to a table, and which table is the target
log_conflict_type	which conflict types it logs, if NULL means all
log_conflict_res	which conflict resolutions it logs, if NULL means all

bdr.node_peer_progress

Catalog used to keep track of every node's progress in the replication stream. Every node in the cluster regularly broadcasts its progress every `bdr.replay_progress_frequency` milliseconds to all other nodes (default is 60000 ms - i.e 1 minute). Expect N * (N-1) rows in this relation.

You may be more interested in the `bdr.node_slots` view for monitoring purposes. See also Monitoring.

bdr.node_peer_progress Columns

Name	Type	Description
node_id	oid	The OID of the originating node which reported this position info
peer_node_id	oid	The OID of the node's peer (remote node) for which this position info was reported
last_update_sent_time	timestamptz	The time at which the report was sent by the originating node
last_update_recv_time	timestamptz	The time at which the report was received by the local server
last_update_node_lsn	pg_lsn	LSN on the originating node at the time of the report
peer_position	pg_lsn	Latest LSN of the node's peer seen by the originating node
peer_replay_time	timestamptz	Latest replay time of peer seen by the reporting node
last_update_horizon_xid	oid	Internal resolution horizon: all lower xids are known resolved on the reporting node
last_update_horizon_lsn	pg_lsn	Internal resolution horizon: same in terms of an LSN of the reporting node

bdr.node_pre_commit

Used internally on a node configured as a Commit At Most Once (CAMO) partner. Shows the decisions a CAMO partner took on transactions in the last 15 minutes.

bdr.node_pre_commit Columns

Name	Type	Description
origin_node_id	oid	OID of the node where the transaction executed
origin_xid	oid	Transaction Id on the remote origin node
decision	char	'c' for commit, 'a' for abort
local_xid	xid	Transaction Id on the local node
commit_ts	timestamptz	commit timestamp of the transaction
decision_ts	timestamptz	decision time

bdr.internal_node_pre_commit

Internal catalog table; please use the `bdr.node_pre_commit` view.

bdr.node_estimates

This view contains information about how much time a peer node will take to apply all the pending WAL from the local node.

bdr.node_estimates Columns

Name	Type	Description
peer_node_id	oid	The OID of node's peer (remote node) for which this info was reported
peer_node_nai	name	Name of the target peer node
apply_rate	bigint	Number of WAL bytes being applied per second at the peer node
catchup_interv	interval	Approximate time required for the peer node to catchup to all the local WAL that is yet to be applied

bdr.node_replication_rates

This view contains information about outgoing replication activity from the local node.

bdr.node_replication_rates Columns

Name	Type	Description
target_name	name	Name of the target peer node
sent_lsn	pg_lsn	Latest sent position
replay_lsn	pg_lsn	Latest position reported as replayed (visible)
replay_lag	interval	Approximate lag time for reported replay
replay_lag_byt	int8	Bytes difference between replay_lsn and current WAL write position
replay_lag_size	text	Human-readable bytes difference between replay_lsn and current WAL write position
apply_rate	bigint	Number of WAL bytes being applied per second at the peer node
catchup_interv	interval	Approximate time required for the peer node to catchup to all the local WAL that is yet to be applied

Note
The `replay_lag` is set immediately to zero after reconnect; we suggest as a workaround to use `replay_lag_bytes`, `replay_lag_size` or `catchup_interval`.

bdr.node_slots

This view contains information about replication slots used in the current database by BDR.

See Monitoring Outgoing Replication for guidance on the use and interpretation of this view's fields.

bdr.node_slots Columns

Name	Type	Description
target_dbname	name	Database name on the target node
node_group_name	name	Name of the BDR group
node_group_id	oid	The OID of the BDR group
origin_name	name	Name of the origin node
target_name	name	Name of the target node
origin_id	oid	The OID of the origin node
target_id	oid	The OID of the target node
local_slot_name	name	Name of the replication slot according to BDR
slot_name	name	Name of the slot according to Postgres (should be same as above)
plugin	name	Logical decoding plugin using this slot (should be pglogical_output)
slot_type	text	Type of the slot (should be logical)
datoid	oid	The OID of the current database
database	name	Name of the current database
temporary	bool	Is the slot temporary
active	bool	Is the slot active (does it have a connection attached to it)
active_pid	int4	The PID of the process attached to the slot
xmin	xid	The XID needed by the slot
catalog_xmin	xid	The catalog XID needed by the slot
restart_lsn	pg_lsn	LSN at which the slot can restart decoding
confirmed_flush_lsn	pg_lsn	Latest confirmed replicated position
usesysid	oid	sysid of the user the replication session is running as
username	name	username of the user the replication session is running as
application_name	text	Application name of the client connection (used by `synchronous_standby_names`)
client_addr	inet	IP address of the client connection
client_hostname	text	Hostname of the client connection
client_port	int4	Port of the client connection
backend_start	timestamptz	When the connection started
state	text	State of the replication (catchup, streaming, . . .) or 'disconnected' if offline
sent_lsn	pg_lsn	Latest sent position

Name	Type	Description
write_lsn	pg_lsn	Latest position reported as written
flush_lsn	pg_lsn	Latest position reported as flushed to disk
replay_lsn	pg_lsn	Latest position reported as replayed (visible)
write_lag	interval	Approximate lag time for reported write
flush_lag	interval	Approximate lag time for reported flush
replay_lag	interval	Approximate lag time for reported replay
sent_lag_bytes	int8	Bytes difference between sent_lsn and current WAL write position
write_lag_bytes	int8	Bytes difference between write_lsn and current WAL write position
flush_lag_bytes	int8	Bytes difference between flush_lsn and current WAL write position
replay_lag_bytes	int8	Bytes difference between replay_lsn and current WAL write position
sent_lag_size	text	Human-readable bytes difference between sent_lsn and current WAL write position
write_lag_size	text	Human-readable bytes difference between write_lsn and current WAL write position
flush_lag_size	text	Human-readable bytes difference between flush_lsn and current WAL write position
replay_lag_size	text	Human-readable bytes difference between replay_lsn and current WAL write position

Note
The `replay_lag` is set immediately to zero after reconnect; we suggest as a workaround to use `replay_lag_bytes` or `replay_lag_size`.

bdr.node_summary

This view contains summary information about all BDR nodes known to the local node.

`bdr.node_summary` Columns

Name	Type	Description
node_name	name	Name of the node
node_group_name	name	Name of the BDR group the node is part of
interface_name	name	Name of the connection interface used by the node

Name	Type	Description
interface_connstr	text	Connection string to the node
peer_state_name	text	Consistent state of the node in human readable form
peer_target_state_name	text	State which the node is trying to reach (during join or promotion)
node_seq_id	int4	Sequence identifier of the node used for generating unique sequence numbers
node_local_dbname	name	Database name of the node
default_repset_name	name	Name of the default replication set
set_repl_ops	text	Which operations does the default replication set replicate
node_id	oid	The OID of the node
node_group_id	oid	The OID of the BDR node group
default_repset_id	oid	The OID of the default replication set
if_id	oid	The OID of the connection interface used by the node

bdr.replication_sets

A view showing replication sets defined in the BDR group, even if they are not currently used by any node.

bdr.replication_sets Columns

Name	Type	Description
set_id	Oid	The OID of the replication set
set_name	name	Name of the replication set
replicate_insert	boolean	Indicates if the replication set replicates INSERTs
replicate_update	boolean	Indicates if the replication set replicates UPDATEs
replicate_delete	boolean	Indicates if the replication set replicates DELETEs
replicate_truncate	boolean	Indicates if the replication set replicates TRUNCATEs
set_autoadd_tables	boolean	Indicates if new tables will be automatically added to this replication set
set_autoadd_seqs	boolean	Indicates if new sequences will be automatically added to this replication set
node_name	name	Name of the local node if the replication set is used by it, otherwise `NULL`
node_group_name	name	Name of the BDR group if the replication set is used by local node, otherwise `NULL`

bdr.schema_changes

A simple view to show all the changes to schemas within BDR.

bdr.schema_changes Columns

Name	Type	Description
schema_changes_ts	timestampstz	The ID of the trigger
schema_changes_change	char	A flag of change type
schema_changes_classid	oid	Class ID
schema_changes_objectid	oid	Object ID
schema_changes_subid	smallint	The subscription
schema_changes_descr	text	The object changed
schema_changes_addrnames	text[]	Location of schema change

bdr.sequence_alloc

A view to see the sequences allocated.

bdr.sequence_alloc Columns

Name	Type	Description
seqid	regclass	The ID of the sequence
seq_chunk_size	bigint	A sequence number for the chunk within its value
seq_allocated_up_to	bigint	
seq_nallocs	bigint	
seq_last_alloc	timestamptz	Last sequence allocated

bdr.sequence_kind

An internal state table storing the type of each non-local sequence. The view `bdr.sequences` is recommended for diagnostic purposes.

bdr.sequence_kind Columns

Name	Type	Description
seqid	oid	Internal OID of the sequence
seqkind	char	Internal sequence kind identifier

bdr.sequences

This view lists all sequences with their kind, excluding sequences for internal BDR book-keeping.

bdr.sequences Columns

Name	Type	Description
nspname	name	Namespace containing the sequence
relname	name	Name of the sequence
seqkind	text	Type of the sequence (local, timeshard)

bdr.stat_relation

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

bdr.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)

bdr.stat_subscription

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

bdr.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)

bdr.state_journal

An internal node state journal. Please use `bdr.state_journal_details` for diagnostic purposes instead.

bdr.state_journal_details

Every change of node state of each node is logged permanently in `bdr.state_journal` for diagnostic purposes. This view provides node names and human-readable state names and carries all of the information in that journal. Once a node has successfully joined, the last state entry will

be `BDR_PEER_STATE_ACTIVE`. This differs from the state of each replication connection listed in `bdr.node_slots.state`.

`bdr.state_journal_details` Columns

Name	Type	Description
state_counter	oid	Monotonically increasing event counter, per node
node_id	oid	Internal node id
node_name	name	Name of the node
state	oid	Internal state id
state_name	text	Human-readable state name
entered_time	timestamptz	Point in time the current node observed the state change

bdr.subscription

This catalog table lists all the subscriptions owned by the local BDR node, and which mode they are in.

`bdr.subscription` Columns

Name	Type	Description
pgl_subscription_id	oid	Subscription in pglogical
nodegroup_id	oid	Id of nodegroup
origin_node_id	oid	Id of origin node
target_node_id	oid	Id of target node
subscription_mode	char	Mode of subscription
source_node_id	oid	Id of source node

bdr.subscription_summary

This view contains summary information about all BDR subscriptions that the local node has to other nodes.

`bdr.subscription_summary` Columns

Name	Type	Description
node_group_name	name	Name of the BDR group the node is part of
sub_name	name	Name of the subscription

Name	Type	Description
origin_name	name	Name of the origin node
target_name	name	Name of the target node (normally local node)
sub_enabled	bool	Is the subscription enabled
sub_slot_name	name	Slot name on the origin node used by this subscription
sub_replication_sets	text[]	Replication sets subscribed
sub_forward_origins	text[]	Does the subscription accept changes forwarded from other nodes besides the origin
sub_apply_delay	interval	Delay transactions by this much compared to the origin
sub_origin_name	name	Replication origin name used by this subscription
bdr_subscription_mode	char	Subscription mode
subscription_status	text	Status of the subscription worker
node_group_id	oid	The OID of the BDR group the node is part of
sub_id	oid	The OID of the subscription
origin_id	oid	The OID of the origin node
target_id	oid	The OID of the target node
last_xact_replay_timestamp	timestamptz	Timestamp of last transaction replayed on this subscription

bdr.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.

bdr.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

(*) These columns are reserved for future use and should currently be NULL

bdr.trigger

Within this view, you can see all the stream triggers created. Often triggers here are created from `bdr.create_conflict_trigger`.

`bdr.trigger` Columns

Name	Type	Description
trigger_id	oid	The ID of the trigger
trigger_reloid	regclass	Name of the relating function
trigger_pgtgid	oid	Postgres trigger ID
trigger_type	char	Type of trigger call
trigger_name	name	Name of the trigger

bdr.triggers

An expanded view of `bdr.trigger` with more easy to read columns.

Name	Type	Description
trigger_name	name	The name of the trigger
event_manipulation	text	The operation(s)
trigger_type	bdr.trigger_type	Type of trigger
trigger_table	bdr.trigger_reloid	The table that calls it
trigger_function	name	The function used

bdr.worker_errors

A persistent log of errors from BDR background worker processes, which includes errors from the underlying pglogical worker processes.

`bdr.worker_errors` Columns

Name	Type	Description
sub_name	name	Name of the subscription
worker_role	int4	Internal identifier of the role of this worker (1: manager, 2: receive, 3: writer, 4: output, 5: extension)
worker_pid	int4	Process id of the worker causing the error
error_time	timestamptz	Date and time of the error

Name	Type	Description
error_message	text	Description of the error
error_context_message	text	Context in which the error happened

bdr.monitor_group_versions_details

Uses `bdr.run_on_all_nodes` to gather BDR/pglogical information from all nodes.

`bdr.monitor_group_versions_details` Columns

Name	Type	Description
node_id	oid	Internal node id
node_name	name	Name of the node
postgres_version	text	PostgreSQL version on the node
pglogical_version	text	Pglogical version on the node
bdr_version	text	BDR version on the node
bdr_edition	text	BDR edition (SE or EE) on the node

bdr.monitor_group_raft_details

Uses `bdr.run_on_all_nodes` to gather Raft Consensus status from all nodes.

`bdr.monitor_group_raft_details` Columns

Name	Type	Description
node_id	oid	Internal node id
node_name	name	Name of the node
state	text	Raft worker state on the node
leader_id	oid	Node id of the RAFT_LEADER
current_term	int	Raft election internal id
commit_index	int	Raft snapshot internal id

BDR System Functions

BDR management is primarily accomplished via SQL-callable functions. All functions in BDR are exposed in the `bdr` schema. Any calls to these functions should be schema-qualified, rather than putting `bdr` in the `search_path`.

This page contains additional system functions that are not documented in the other sections of the documentation.

Note that you cannot manipulate BDR-owned objects using pglogical functions; only using the following supplied functions.

Version Information Functions

bdr.bdr_edition

This function returns a textual representation of the BDR edition. BDR3 is distributed in either Standard Edition (`SE`) and Enterprise Edition (`EE`); this function can be used to check which of those is currently installed.

The Standard Edition runs on the community version of PostgreSQL 10 and 11, while the Enterprise Edition requires 2ndQPostgres 11. The Enterprise Edition includes more advanced features than the Standard Edition provides.

bdr.bdr_version

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

bdr.bdr_version_num

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

The following formula is used to turn the version number consisting of major version, minor verion and patch release into a single numerical value:

```
MAJOR_VERSION * 10'000 + MINOR_VERSION * 100 + PATCH_RELEASE
```

System Information Functions

bdr.get_relation_stats

Returns the relation information.

bdr.get_subscription_stats

Returns the current subscription statistics.

System and Progress Information Parameters

BDR exposes some parameters that can be queried via `SHOW` in `psql` or using `PQparameterStatus` (or equivalent) from a client application. This section lists all such parameters BDR reports to.

bdr.local_node_id

Upon session initialization, this is set to the node id the client is connected to. This allows an application to figure out what node it is connected to even behind a transparent proxy. It is also used in combination with CAMO, see the CAMO.md#connection-pools-and-proxies section.

bdr.last_committed_lsn

After every `COMMIT` of an asynchronous transaction, this parameter is updated to point to the end of the commit record on the origin node. In combination with `bdr.wait_for_apply_queue`, this allows applications to perform causal reads across multiple nodes, i.e. to wait until a transaction becomes remotely visible.

transaction_id

As soon as Postgres assigns a transaction id, this parameter is updated to show the transaction id just assigned, if CAMO is enabled.

Consensus Function

bdr.consensus_disable

Disables the consensus worker on the local node until server restart or until it's re-enabled using `bdr.consensus_enable` (whichever happens first).

> **Warning**
> Disabling consensus will disable some features of BDR and eventually will impact availability of the BDR cluster if left disabled for prolonged periods of time. This function should only be used in coordination with 2ndQuadrant Support.

bdr.consensus_enable

Re-enabled disabled consensus worker on local node.

bdr.consensus_proto_version

Returns currently used consensus protocol version by the local node.

Needed by the BDR group reconfiguration internal mechanisms.

bdr.consensus_snapshot_export

Generate a new BDR consensus snapshot from the currently committed-and-applied state of the local node and return it as bytea.

The exporting node does not have to be the current Raft leader, nor does it need to be completely up to date with the latest state on the leader. However, such snapshot might not be accepted by `bdr.consensus_snapshot_import()` (see bellow).

The new snapshot is not automatically stored to the local node's `bdr.local_consensus_snapshot` table. It's only returned to the caller.

The generated snapshot may be passed to `bdr.consensus_snapshot_import()` on any other node(s) in the same BDR nodegroup that is behind the exporting node's raft log position.

The local BDR consensus worker must be disabled for this function to work. Typical usage is:

```
SELECT bdr.bdr_consensus_disable();
\copy (SELECT * FROM bdr.consensus_snapshot_export()) TO 'my_node_consensus_snapshot.data'
SELECT bdr.bdr_consensus_enable();
```

While the BDR consensus worker is disabled, DDL locking attempts on the node will fail or time out, galloc sequences will not get new values, Eager and CAMO transactions will pause or ERROR, and other functionality that needs the distributed consensus system will be disrupted. The required downtime is generally very brief.

Depending on the use case, it may be practical to extract a snapshot that already exists from the `snapshot` field of the `bdr.local_consensus_snapshot` table and use that instead. Doing so does not require that the consensus worker be stopped.

bdr.consensus_snapshot_import

Synopsis

```
bdr.consensus_snapshot_import(IN snapshot bytea)
```

Import a consensus snapshot which was exported by `bdr.consensus_snapshot_export()`, usually from another node in the same BDR nodegroup.

It's also possible to use a snapshot extracted directly from the `snapshot` field of the `bdr.local_consensus_snapshot` table on another node.

This function is useful for resetting a BDR node's catalog state to known good state in case of corruption or user mistake.

The snapshot can be imported if the importing node's `apply_index` is less than or equal to the snapshot-exporting node's `commit_index` at the time the snapshot was generated. See `bdr.get_raft_status()`. A node that cannot accept the snapshot because its logs is already too far ahead will raise an `ERROR` and make no changes. The imported snapshot does not have to be completely up-to-date, as once the snapshot is imported the node will fetch the remaining changes from the current leader.

The BDR consensus worker must be disabled on the importing node for this function to work. See notes on `bdr.consensus_snapshot_export()` for details.

It's possible to use this to force the local node to generate a new Raft snapshot by running:

```
SELECT bdr.consensus_snapshot_import(bdr.consensus_snapshot_export());
```

This may also cause it to truncate its Raft logs up to the current applied log position.

bdr.get_consensus_status

Returns status information about the current consensus (Raft) worker.

bdr.get_raft_status

Returns status information about the current consensus (Raft) worker. Alias for `bdr.get_consensus_status`.

Utility Functions

bdr.wait_slot_confirm_lsn

Allows the user to wait until the last write on this session has been replayed to one or all nodes.

Waits until a slot passes certain LSN. If no position is supplied, the current write position is used on the local node.

If no slot name is passed, it will wait until all BDR slots pass the LSN. This is a separate function from the one provided by pglogical so that we can only wait for slots registered for other BDR nodes, not all pglogical slots and, more importantly, not our BDR group slot.

The function polls every 1000ms for changes from other nodes.

If a slot is dropped concurrently the wait will end for that slot. If a node is currently down and is not updating its slot then the wait will continue. You may wish to set `statement_timeout` to complete earlier in that case.

Synopsis

```
bdr.wait_slot_confirm_lsn(slot_name text DEFAULT NULL, target_lsn pg_lsn DEFAULT NULL)
```

Parameters

- `slot_name` - name of replication slot, or if NULL, all BDR slots (only)
- `target_lsn` - LSN to wait for, or if NULL, use the current write LSN on the local node

bdr.wait_for_apply_queue

The function `bdr.wait_for_apply_queue` allows a BDR node to wait for the local application of certain transactions originating from a given BDR node. It will return only after all transactions from that peer node are applied locally. An application or a proxy can use this function to prevent stale reads. For convenience, BDR provides a special variant of this function for CAMO and the CAMO partner node, see bdr.wait_for_camo_partner_queue.

In case a specific LSN is given, that's the point in the recovery stream from the peer to wait for. This can be used in combination with `bdr.last_committed_lsn` retrieved from that peer node on a previous or concurrent connection.

If the given `target_lsn` is NULL, this function checks the local receive buffer and uses the LSN of the last transaction received from the given peer node. Effectively waiting for all transactions already received to be applied. This is especially useful in case the peer node has failed and it's not known which transactions have been sent. Note that in this case, transactions that are still in transit or buffered on the sender side are not waited for.

Synopsis

```
bdr.wait_for_apply_queue(peer_node_name TEXT, target_lsn pg_lsn)
```

Parameters

- `peer_node_name` - the name of the peer node from which incoming transactions are expected to be queued and which should be waited for. If NULL, waits for all peer node's apply queue to be consumed.
- `target_lsn` - the LSN in the replication stream from the peer node to wait for, usually learned via `bdr.last_committed_lsn` from the peer node.

bdr.get_node_sub_receive_lsn

This function can be used on a subscriber to get the last LSN that has been received from the given origin. Either filtered to take into account only relevant LSN increments for transactions to be applied or unfiltered.

The difference between the output of this function and the output of `bdr.get_node_sub_apply_lsn()` measures the size of the corresponding apply queue.

Synopsis

```
bdr.get_node_sub_receive_lsn(node_name name, committed bool default true)
```

Parameters

- `node_name` - the name of the node which is the source of the replication stream whose LSN we are retrieving
- `committed` - the default (true) makes this function take into account only commits of transactions received, rather than the last LSN overall, including actions that have no effect on the subscriber node.

bdr.get_node_sub_apply_lsn

This function can be used on a subscriber to get the last LSN that has been received and applied from the given origin.

Synopsis

```
bdr.get_node_sub_apply_lsn(node_name name)
```

Parameters

- `node_name` - the name of the node which is the source of the replication stream whose LSN we are retrieving

bdr.set_ddl_replication

This function allows us to turn off the DDL replication within a session locally.

Synopsis

```
bdr.set_ddl_replication(ddl_replication text, local boolean DEFAULT false)
```

Parameters

- `ddl_replication` - value of ddl replication ('on', 'off')
- `local` - boolean value for the change; if true, it lasts' only for the current transaction (true, false)

bdr.set_ddl_locking

Set the DDL locking to apply, either locally or globally on all nodes as explained in Executing DDL on BDR systems.

Synopsis

```
bdr.set_ddl_locking(ddl_locking text, local boolean DEFAULT false)
```

Parameters

- `ddl_locking` - value for the DDL locking ('on', 'off', 'DML')
- `local` - boolean value for the change; if true, it lasts' only for the current transaction (true, false)

bdr.run_on_all_nodes

Function to run a query on all nodes.

> **Warning**
> This function will run an arbitrary query on a remote node with the privileges of the user used for the internode connections as specified in the node's DSN. Caution needs to be taken when granting privileges to this function.

Synopsis

```
bdr.run_on_all_nodes(query text)
```

Parameters

- `query` - arbitrary query to be executed

Notes

This function will connect to other nodes and execute the query, returning a result from each of them in json format. Multiple rows may be returned from each node, encoded as a json array. Any errors, such as being unable to connect because a node is down, will be shown in the response field. No explicit statement_timeout or other runtime parameters are set, so defaults will be used.

This function does not go through normal replication, it uses direct client connection to all known nodes.

Don't use this function for running DDL, otherwise you risk breaking replication and inconsistencies between nodes. Use either transparent DDL replication or `bdr.bdr_replicate_ddl_command()` to replicate DDL. DDL may be blocked in a future release.

Example

It's useful to use this function in monitoring, for example in the following query:

```
SELECT bdr.run_on_all_nodes($$
    SELECT local_slot_name, origin_name, target_name, replay_lag_size
      FROM bdr.node_slots
     WHERE origin_name IS NOT NULL
$$);
```

. . . will return something like this on a two node cluster:

```
[
    {
        "dsn": "host=192.168.0.90 dbname=testdb port=7432",
        "node_id": "2079384130",
        "response": [
            {
                "origin_name": "dc1n1",
                "target_name": "dc1n2",
                "local_slot_name": "bdr_testdb_bdrgroup_dc1n2",
                "replay_lag_size": "168 bytes"
            }
        ],
        "node_name": "dc1n1"
    },
    {
        "dsn": "host=192.168.0.91 dbname=testdb port=7433",
        "node_id": "4121887394",
        "response": [
            {
                "origin_name": "dc1n2",
                "target_name": "dc1n1",
                "local_slot_name": "bdr_testdb_bdrgroup_dc1n1",
                "replay_lag_size": "0 bytes"
            }
        ],
        "node_name": "dc1n2"
    }
]
```

bdr.global_lock_table

This function will acquire a global DML locks on a given table. See DDL Locking Details for information about global DML lock.

Synopsis

```
bdr.global_lock_table(relation regclass)
```

Parameters

- `relation` - name or Oid of the relation to be locked

Notes

This function will acquire the global DML lock independently of the `ddl_locking` setting.

The `bdr.global_lock_table` function requires `UPDATE`, `DELETE`, or `TRUNCATE` privilege on the locked `relation`, unless `bdr.backwards_compatibility` is set is set to 30618 or below.

bdr.monitor_group_versions

This function uses BDR/pglogical version information returned from view `bdr.monitor_group_version_details` to provide a cluster-wide version check.

Synopsis

```
bdr.monitor_group_versions()
```

Notes

This function returns a record with fields `status` and `message`, as explained in Monitoring.

This function calls `bdr.run_on_all_nodes()`.

bdr.monitor_group_raft

This function uses BDR/pglogical Raft information returned from view `bdr.monitor_group_raft_details` to provide a cluster-wide Raft check.

Synopsis

```
bdr.monitor_group_raft()
```

Notes

This function returns a record with fields `status` and `message`, as explained in Monitoring.

This function calls `bdr.run_on_all_nodes()`.

bdr.monitor_replslots

This function uses replication slot status information returned from view `pg_replication_slots` (slot active or inactive) to provide a local check considering all replication slots, except the BDR group slots.

Synopsis

```
bdr.monitor_replslots()
```

Notes

This function returns a record with fields `status` and `message`, as explained in Monitoring.

Internal Functions

BDR message payload functions

bdr.decode_message_response_payload and bdr.decode_message_payload

These functions decode the consensus payloads to a more human-readable output.

Used primarily by the `bdr.global_consensus_journal_details` debug view.

bdr.get_global_locks

This function shows information about global locks held on the local node.

Used to implement the `bdr.global_locks` view, to provide a more detailed overview of the locks.

bdr.get_slot_flush_timestamp

Retrieves the timestamp of the last flush position confirmation for a given replication slot.

Used internally to implement the `bdr.node_slots` view.

bdr internal function replication functions

bdr.internal_alter_sequence_set_kind, internal_replication_set_add_table, internal_replication_set_remove_table

Functions used internally for replication of the various function calls.

No longer used by the current version of BDR. Only exists for backwards compatibility during rolling upgrades.

bdr.internal_submit_join_request

Submits a consensus request for joining a new node.

Needed by the BDR group reconfiguration internal mechanisms.

bdr.isolation_test_session_is_blocked

A helper function, extending (and actually invoking) the original `pg_isolation_test_session_is_blocked` with an additional check for blocks on global locks.

Used for isolation/concurrency tests.

bdr.local_node_info

This function displays information for the local node, needed by the BDR group reconfiguration internal mechanisms.

The view `bdr.local_node_summary` provides similar information useful for user consumption.

bdr.msgb_connect

Function for connecting to the connection pooler of another node, used by the consensus protocol.

bdr.msgb_deliver_message

Function for sending messages to another node's connection pooler, used by the consensus protocol.

bdr.peer_state_name

This function transforms the node state (`node_state`) into a textual representation, and is used mainly to implement the `bdr.node_summary` view.

bdr.request_replay_progress_update

Requests the immediate writing of a 'replay progress update' Raft message. It is used mainly for test purposes, but can be also used to test if the consensus mechanism is working.

bdr.seq_nextval

Internal implementation of sequence increments.

This function will be used instead of standard `nextval` in queries which interact with BDR Global Sequences.

bdr.show_subscription_status

Retrieves information about the subscription status, and is used mainly to implement the `bdr.subscription_summary` view.

bdr.conflict_resolution_to_string

Transforms the conflict resolution from oid to text.

The view `bdr.apply_log_summary` is using it to give user-friendly information for the conflict resolution.

bdr.conflict_type_to_string

Transforms the conflict type from oid to text.

The view `bdr.apply_log_summary` is using it to give user-friendly information for the conflict type.

bdr.reset_subscription_stats

A function that resets the statistics created by subscriptions. Simply returns a boolean result.

bdr.reset_relation_stats

A function that resets the relation stats. Simply returns a boolean result.

bdr.pg_xact_origin

Return origin id of a given transaction.

Synopsis

```
bdr.pg_xact_origin(xmin xid)
```

Parameters

- `xid` - Transaction id whose origin is returned

bdr.difference_fix_origin_create

Creates a replication origin with a given name passed as argument, but adding a `bdr_` prefix. It returns the internal id of the origin. This performs same functionality as `pg_replication_origin_create()`, except this requires `bdr_superuser` rather than postgres superuser permissions.

Synopsis

bdr.difference_fix_session_setup

Marks the current session as replaying from the current origin. The function uses the pre-created `bdr_local_only_origin` local replication origin implicitly for the session. It allows replay progress to be reported. It returns void. This performs the same functionality as `pg_replication_origin_session_setup()`, except that this requires `bdr_superuser` rather than postgres superuser permissions. Note that the earlier form of the function: `bdr.difference_fix_session_setup(text` has been deprecated and will be removed in upcoming releases.

Synopsis

```
bdr.difference_fix_session_setup()
```

bdr.difference_fix_session_reset

Marks the current session as not replaying from any origin, essentially resetting the effect of `bdr.difference_fix_session_setup()`. It returns void. This performs the same functionality as `pg_replication_origin_session_reset()`, except this requires `bdr_superuser` rather than postgres superuser permissions.

Synopsis

```
bdr.difference_fix_session_reset()
```

bdr.difference_fix_xact_set_avoid_conflict

Marks the current transaction as replaying a transaction that has committed at LSN '0/0' and timestamp '2000-01-01'. This performs the same functionality as `pg_replication_origin_xact_setup('0/0', '2000-01-01'` except this requires `bdr_superuser` rather than postgres superuser permissions.

Synopsis

```
bdr.difference_fix_xact_set_avoid_conflict()
```

bdr.resynchronize_table_from_node(node_name name, relation regclass)

Resynchronizes the relation from a remote node.

Synopsis

```
bdr.resynchronize_table_from_node(node_name name, relation regclass)
```

Parameters

- `node_name` - the node from which to copy/resync the relation data.
- `relation` - the relation to be copied from the remote node.

Notes

This acquires a global DML lock on the relation, truncates the relation locally, and copies data into it from the remote node.

The relation must exist on both nodes with the same name and definition.

Resynchronization of partitioned tables with identical partition definitions, resynchronization partitioned table to non-partitioned table and vice-versa and resynchronization of referenced tables by temporarily dropping and recreating foreign key constraints are all supported.

After running the function on a referenced table, if the referenced column data no longer matches the referencing column values, it throws an error and function should be rerun after resynchronizing the referencing table data.

Currently, row_filters are ignored by this function.

The `bdr.resynchronize_table_from_node` function can be only executed by the owner of the table, provided the owner has bdr_superuser privileges.

bdr.alter_subscription_skip_changes_upto

This does the same as `pglogica.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 target 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. 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 target side, so it's important to save the problem transaction whenever possible. See the example below.

It's possible to skip over changes without `bdr.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:

- 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 source node* (if possible)
- `bdr.alter_subscription_skip_changes_upto` on the target node
- apply repaired or equivalent changes on the target 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

```
bdr.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 on the target 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:  XX000: CONFLICT: target_table_missing; resolver skip_if_recently_dropped returned an
CONTEXT:  during apply of INSERT from remote relation public.break_me in xact with commit-end
committs 2021-02-02 15:11:03.913792+01 (action #2) (effective sess origin id=2 lsn=0/300AC18)
while consuming 'I' message from receiver for subscription bdr_regression_bdrgroup_node1_node
on node node2 (id=3367056606) from upstream node node1 (id=1148549230, reporiginid=2)
```

In this portion of log we have the information we need: the_target_lsn: **0/300AC18** the_subscription: **bdr_regression_bdrgroup_node1_node2**

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

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

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 COPYing it out on the provider side first, using the subscription's slot name.

```
\\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 `bdr....skip_changes_upto` does a faster skip that avoids decoding and outputting all the data:

```
SELECT bdr.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 target node, using the dumped transaction contents as a guide.

Finally, re-enable the subscription:

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

Credits and Licence

BDR3 has been designed, developed and tested by the 2ndQuadrant team:

- Petr Jelinek
- Craig Ringer
- Markus Wanner
- Pavan Deolasee
- Tomas Vondra
- Simon Riggs
- Nikhil Sontakke
- Pallavi Sontakke
- Amruta Deolasee
- Rahila Syed

Appendix A: Release Notes for BDR3

BDR 3.6.32

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

Resolved Issues

- Catchup replication slot cleanup during PARTing of a node (BDR-2368, RT82884)
  When parting a node, the catcup replication slot may be left behind if the source node used for catching up changes. Clean up these replication slots once the catchup finishes. Similarly clean up the catchup information from BDR catalogs.
- Cleanup replication slot when bdr_init_physical fails (BDR-2364, RT74789)
  If bdr_init_physical aborts without being able to join the node, it will leave behind an inactive replication slot. Remove such a replication slot when it is inactive before an irregular exit.

Improvements

- Allow consumption of the reserved galloc sequence slot (BDR-2367, RT83437, RT68255)
  The galloc sequence slot reserved for future use by background allocator can be consumed in the presece of consensus failure.

Upgrades

This release supports upgrading from the following versions of BDR:

- 3.6.20 and higher

BDR 3.6.31

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

Resolved Issues

- Make ALTER TABLE lock the underlying relation only once (RT80204)
  This avoids the ALTER TABLE operation falling behind in the queue when it released the lock in between internal operations. With this fix, concurrent transactions trying to acquire the same lock after the ALTER TABLE command will properly wait for the ALTER TABLE to finish.
- Reduce log for bdr.run_on_all_nodes (BDR-2153, RT80973)
  Don't log when setting `bdr.ddl_replication` to off if it's done with the "run_on_all_nodes" variants of function. This eliminates the flood of logs for monitoring functions.

Improvements

- Use 64 bits for calculating lag size in bytes (BDR-2215)

Upgrades

This release supports upgrading from the following versions of BDR:

- 3.6.20 and higher

BDR 3.6.30

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

Resolved Issues

- Ensure loss of CAMO partner connectivity switches to Local Mode immediately
  This prevents disconnected partner from being reported as CAMO ready.
- Fix the cleanup of `bdr.node_pre_commit` for async CAMO configurations (BDR-1808)
  Previously, the periodic cleanup of commit decisions on the CAMO partner checked the readiness of it's partner, rather than the origin node. This is the same node for symmetric CAMO configurations, so those were not affected. This release corrects the check for asymmetric CAMO pairings.

Upgrades

This release supports upgrading from the following versions of BDR:

- 3.6.20 and higher

BDR 3.6.29

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

Resolved Issues

- Switch from CAMO to Local Mode only after timeouts (EE, RT74892)
  Do not use the `catchup_interval` estimate when switching from CAMO protected to Local Mode, as that could induce inadvertent switching due to load spikes. Use the estimate only when switching from Local Mode back to CAMO protected (to prevent toggling forth and back due to lag on the CAMO partner).

- Prevent duplicate values generated locally by galloc sequence in high concurrency situations when the new chunk is used (RT76528)
  The galloc sequence could have temporarily produce duplicate value when switching which chunk is used locally (but not across nodes) if there were multiple sessions waiting for the new value. This is now fixed.
- Ensure that the group slot is moved forward when there is only one node in the BDR group
  This prevents disk exhaustion due to WAL accumulation when the group is left running with just single BDR node for prolonged period of time. This is not recommended setup but the WAL accumulation was not intentional.
- Advance Raft protocol version when there is only one node in the BDR group
  Single node clusters would otherwise always stay on oldest support protocol until another node was added. This could limit available feature set on that single node.

Improvements

- Reduce frequency of CAMO partner connection attempts (EE)
  In case of a failure to connect to a CAMO partner do not retry immediately (leading to a fully busy pglogical manager process), but throttle down repeated attempts to reconnect to once per minute.
- Ensure CAMO configuration is checked again after a reconnect (EE)

Upgrades

This release supports upgrading from the following versions of BDR:

- 3.6.20 and higher

BDR 3.6.28.1

This is a hotfix release for BDR 3.6.28.

Resolved Issues

- Fix potential FATAL error when using global DML locking with CAMO (BDR-1675, BDR-1655)
- Fix lag calculation for CAMO local mode delay (BDR-1681)

BDR 3.6.28

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

Please make sure you read the pglogical 3.6.28 release notes for the important security and physical promotion data loss fixes provided there.

Resolved Issues

- Ensure `bdr.camo_local_mode_delay` is actually being applied (BDR-803)
  This artificial delay allows throttling a CAMO node that is not currently connected to its CAMO partner to prevent it from producing transactions faster than the CAMO partner can possibly apply. In previous versions, it did not properly kick in after `bdr.global_commit_timeout` amount of lag, but only 1000 times later (due to erroneously comparing seconds to milliseconds).
- Fix `bdr.alter_sequence_set_kind` to accept a bigint as a start value (RT74294)
  The function was casting the value to an `int` thus getting bogus values when `bigint` was used.
- Fix node replication slot handling on node name reuse (RT71888)
  Cleanup local info slot name when the slot is dropped, this will prevent data loss by not dropping the slot during `bdr_init_physical` when it joins a node re-using the same name.
- Don't stop consensus worker when manager exists (RT73539)
  Killing the consensus after configuration changes would destabilize Raft.
- Fix ignored `--recovery-conf` option in `bdr_init_physical`.
  `bdr_init_physical` was completely ignoring the value of the –recovery-conf option if provided. It would append to a recovery.conf if one already existed in the target data dir but totally ignored the supplementary config provided on the command line, if any.

Improvements

- Allow user to specify a different postgres.auto.conf file. (BDR-1400)
  Added command-line argument `--postgresql-auto-conf` to be used when the user needs to specify a different postgres.auto.conf file.
- Log LSN when advancing replication origin during join
  Make `bdr_process_node_state_join_start()` log the LSN it advances the replication origin for the catchup join-target node to when BDR is started by `bdr_init_physical` with a `bdr.init_physical_lsn` set. This is critical information for diagnosing issues with physical join.
- Replication slot monitoring improvements (BDR-720)
  Differentiate decoder slot in `bdr.node_slots`.
  Include node and group information in `bdr.node_slots` when origin and target are in different node group.
  Teach `bdr.monitor_local_replslots` what slots to expect.
- Improve documentation of the backup/restore procedure (RT72503, BDR-1340) Recommend against dropping the extension with cascade because it may drop user columns that are using CRDT types and break the sequences. It's better do use `drop_node` function instead.

Upgrades

This release supports upgrading from following versions of BDR:

- 3.6.20 and higher

BDR 3.6.27

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

Please make sure you read the pglogical 3.6.27 release notes for the important security and physical promotion data loss fixes provided there.

Resolved Issues

- Make sure `bdr_init_physical` does not leave temporary replication slots behind (BDR-191, RT70355)
  We now use native temporary slots in PostgreSQL which are automatically cleaned up.
- Make the consensus worker exit if postmaster dies (BDR1063, RT70024)
  This solves issues with consensus worker hanging after crash of other PostgreSQL process.
- Fix reuse of internal connection pool connections
  Improves behavior with larger number of nodes.

Upgrades

This release supports upgrading from following versions of BDR:

- 3.6.20 and higher

BDR 3.6.26

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

Please make sure you read the pglogical 3.6.26 release notes for the important security and physical promotion data loss fixes provided there.

Resolved Issues

- Fix crash in `bdr.run_on_all_nodes` when remote node returned an error

Other Changes

- Add `bdr.alter_subscription_skip_changes_upto()` (BDR-195) This function allows moving subscription ahead without replicating the changes which is useful for error recovery.
- Add a new documentation appendix which shows example of how to do table rewriting DDL safely with BDR (RTRT69514, RT69340).
- Improve coexistence with BDR 3.7 in the same BDR group (during upgrades).

Upgrades

This release supports upgrading from following versions of BDR:

- 3.6.20 and higher

BDR 3.6.25

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

Resolved Issues

- Don't display additional node connection string in node_summary and local_node_summary views (RT69564)
  BDR only supports one connection string for each node.
- Fix "bdr node . . . not found" in replay progress update (RT69779)
  Nodes that have pending progress info globally might no longer exist locally, and if that's the case is safe to ignore the progress update for them.
- Update state journal on node creation (RM20111)
  This solves corner cases where we could sometimes fail to join or part a node if the node with same name previously existed and was removed.
- Fix issues with stopping supervisor process when blocked by network call (RM20311)
  This issue could have resulted in PostgreSQL refusing to stop if the supervisor process was stuck on a network call.

Improvements

- Add new param detector_args to bdr.alter_table_conflict_detection (RT69677)
  To allow additional parameters for individual detectors.

Currently the only additional parameter is `atttype` for the row_version detection method which allows using smallint and bigint, rather than just the default integer for the column type.

BDR 3.6.24

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

Resolved Issues

- Unblock replay progress updates after a failing CAMO connection (EE) (RT69493, RM19924)
  After a connection to the configured CAMO partner (or origin) failed, for example if the CAMO partner simply has not been started at the start of the origin node, it was not properly reset. This not only prevented further connection attempts and prevented CAMO from operating, but also blocked replay progress updates from the blocked node. Which in turn prevents the group slot from advancing, meaning WAL data piled up on all nodes.

This release not only corrects the cleanup of the CAMO connection state to enable retries, but also decouples replay progress updates from the CAMO connection. Should there ever be another issue with CAMO connections, replay progress updates and the group slot should not be affected.

- Ensure `bdr_init_physical` works on matching BDR versions (RT69520, RM19975)
  To initialize a new node from a physical copy, the new node must use the same BDR version as the node the physical backup was copied from. This release ensures `bdr_init_physical` checks and gives a useful error otherwise.
- Relax the safety checks in `bdr.drop_node` a bit again (RT69639)
  The check to prevent premature removal introduced in 3.6.23 turned out to be too strict and prevented dropping nodes as soon as any one peer had properly dropped the node already. This release relaxes the check again to allow dropping on all peer nodes.

Improvements

- Local node information in `bdr.node_summary` (RM20002, RT69541) Allow user to see information related to the local node even before the node is part of a node_group. This extends also to the `bdr.local_node_summary` view.

BDR 3.6.23

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

Resolved Issues

- Resolve CAMO transactions within a local transaction (RT69404)
  On rare occasions, the final commit stage of a CAMO or Eager All Node transaction still needs to do lookups in the system catalog. This led to a segfault, because BDR did not properly wrap this operation in a (read-only) transaction with full access to system catalogs.
- Better validate inputs to administration functions (RM19276, RM18108, RM17994, RT69013)
  When intentionally fed with invalid input (mostly `NULL`), some of the administrative functions of BDR crashed, leading to a restart of the Postgres node. This would constitute a denial of service attack for roles with either `bdr_application` or `bdr_superuser` privileges.
- Add a warning when trying to join known broken BDR versions (RT69088)
  BDR versions 3.6.19 and older are susceptible to data-loss when joining a BDR cluster. BDR now identifies this situation and logs a warning at join time.
- Handle missing column gracefully for `ALTER COLUMN TYPE` (RM19389, RT69114)
  The fix just throws same error Postgres would when `ALTER COLUMN TYPE` with non-existent column is executed.
- Fix `JOINING` state handling on consensus request timeout (RT69076)
  The timeout during `JOINING` state handling could result in node unable to join the BDR group. The retry logic now handles this state correctly.
- Change the snapshot restore to follow Raft paper more closely
  Fixes potential metadata consistency issues when node returns after longer period of downtime.
- Fix potential crash in `bdr.resynchronize_table_from_node` (RM19527)
- Extend bdr.drop_node with a check preventing premature removal (RM19280)
  The function `bdr.drop_node` by default now checks whether the node to drop has been fully parted on all nodes prior to allowing to drop its metadata. An additional `force` argument allows to disable this check and effectively exposes the previous behavior.
- Validation of `replication_set_remove_table` input (RT69248, RM19620)
  Check if the relation Oid passed as the function argument is valid.

Other Changes

- Improve error messages (RM19483)
- Minor documentation clarifications and language fixes (RM19825, RM19296, RT69401, RT69044)

BDR 3.6.22

This is a security and maintenance release for BDR 3.6 which also includes various minor features.

Resolved Issues

- Correct a hang in `bdr.wait_for_apply_queue` (RM11416, also affects CAMO)
  Keepalive messages possibly move the LSN forward. In an otherwise quiescent system (without any transactions processed), this may have led to a hang in `bdr.wait_for_apply_queue`, because there may not be anything to apply for the corresponding PGL writer, so the `apply_lsn` doesn't ever reach the `receive_lsn`. A proper CAMO client implementation uses `bdr.logical_transaction_status`, which in turn uses the affected function internally. Thus a CAMO switch- or fail-over could also have led to a hang. This release prevents the hang by discarding LSN increments for which there is nothing to apply on the subscriber.

- Correct a problem with Raft snapshot writing when a 3.6.21 node became leader of a cluster that also included nodes on version 3.6.20 or earlier. Nodes that were parted but not subsequently dropped (so they still appear in the `bdr.node` table) can cause it as well. The issue can cause DDL locking to time out, galloc sequence allocation to stop, and other issues related to loss of functioning Raft-based consensus. If this problem is encountered, the following error will be seen in logs:

  ERROR: invalid snapshot: record length [...] doesn't match expected length [...] for chunk "ddl_epoch"

and `SELECT bdr.get_raft_status()` will intermittently report `ERROR: could not find consensus worker` when called.

To check if this issue could be affecting you, check `bdr.node` for any parted nodes with `proto_version_max` less than 13. Also query

```
SELECT status->>'protocol_version'
FROM bdr.get_raft_status() status;
```

to see if the result is less than 13. If either is true and your cluster has 3.6.21 nodes you could be affected by the issue. To prevent or solve it you are affected by it, upgrade immediately to 3.6.22.

This issue does not arise when all nodes are on 3.6.20 or older.

- Fix a problem with NULL values in `bdr.ddl_epoch` catalog (RM19046). Release 3.6.21 added a new `epoch_consumed_lsn` column to `bdr.ddl_epoch` catalog. Adding a new column would set the column value to NULL in all existing rows in the table. But the code failed to handle the NULL values properly. This could lead to reading garbage values or even memory access errors. The garbage values can potentially lead to global lock timeouts as a backend may wait on a LSN which is far into the future.

We fix this by updating all NULL values to '0/0' LSN, which is an invalid value representation for LSN. The column is marked NOT NULL explicitly and the code is fixed to never generate new NULL values for the column.

- Allow consensus protocol version upgrades despite parted nodes (RM19041)
  Exclude already parted nodes from the consensus protocol version negotiation, as such nodes do not participate in the consensus protocol any more. Ensures the newest protocol version among the set of active nodes is used.
- Check relation oid in `bdr.drop_trigger` (RM19276, RT69013) Error instead of segfault when we pass an invalid relation oid to the function bdr.drop_trigger.

Improvements

- Numerous fixes for galloc sequences (RM18519, RM18512) The "nextval" code for galloc sequences had numerous issues:
- Large INCREMENT BY values (+ve or -ve) were not working correctly
- Large CACHE values were not handled properly
- MINVAL/MAXVAL not honored in some cases The crux of the issue was that large increments or cache calls would need to make multiple RAFT fetch calls. This caused the loop retry code to be invoked multiple times. The various variables to track the loops needed adjustment.
- Error out if INCREMENT BY is more than galloc chunk range (RM18519) The smallint, int and bigint galloc sequences get 1000, 1000000, 1000000000 values allocated in each chunk respectively. We error out if the INCREMENT value is more than these ranges.
- Extend `bdr.get_node_sub_receive_lsn` with an optional `committed` argument
  The default behaviour has been corrected to return only the last received LSN for a committed transaction to apply (filtered), which is the original intent and use of the function (e.g. by HARP). Passing a `false` lets this function return the unfiltered most recent LSN received, matching the previous version's behavior. This change is related to the hang in `bdr.wait_for_apply_queue` mentioned above.
- Fix tracking of the last committed LSN for CAMO and Eager transactions (RM13509)
  The GUC `bdr.last_committed_lsn` was only updated for standard asynchronous BDR transactions, not for CAMO or Eager ones.

BDR 3.6.21

This is a security and maintenance release for BDR 3.6 which also includes various minor features.

Resolved Issues

- SECURITY: Qualify calls to `unnest` from `bdr.logical_transaction_status` (RM18359)
  Required to protect CAMO users from attack risks identified in CVE-2018-1058, when the user application avoids the insecure coding practices identified there. See BDR Security chapter for

further explanation. `bdr.logical_transaction_status` is typically only used by an application taking advantage of the CAMO feature.

- SECURITY: Qualify a call to an inequality operator in `bdr_init_physical` (RM18359)
  Since 3.6.20, the `bdr_init_physical` utility uses the <> inequality operator. Similar to the above entry, the operator is now called by its fully qualified name to avoid attack risks identified in CVE-2018-1058.

- SECURITY: Recent PostgreSQL 11.9 et al reported CVE-2020-14349. The related risk is the same as CVE-2018-1058 and was already handled in BDR3.6.19, so no further action has been taken in this release.

- Re-add the "local_only" replication origin (RT68021)
  Using `bdr_init_physical` may have inadvertently removed it due to a bug that existing up until release 3.6.19. This release ensures to recreate it, if it's missing.

- Eliminate SQL calls to `pg_switch_wal` from BDR (RT68159, RM17356)
  The helper function `bdr.move_group_slot_all_nodes` as well internal replay progress update handling invoked the Postgres function `pg_switch_wal` via SQL, thus requiring explicit execute permissions. Instead use direct internal calls from the BDR extension to avoid that requirement.

- Handle NULL arguments passed to functions in BDR schema gracefully (RT68375, RM17994)
  Some of the functions in BDR schema, e.g. bdr.alter_node_add_log_config() caused segmentation fault when NULL arguments were passed to those. Those are fixed to handle the NULL arguments appropriately. Those functions which do not expect NULL arguments now throw an error mentioning so. Others handle NULL arguments according to the functionality offered by individual function.

- Prevent violation of NOT NULL constraints in BDR-internal catalogs (RM18335)
  Postgres 11.9 is more strict with `NOT NULL` constraints for catalog tables. Correct and relax constraints for two internal tables for which BDR stores `NULL` values in.

- Wait for replication changes triggered by prior epoch (RM17594, RM17802)
  When a node requests a global DDL lock, it now waits until it has seen the replication changes associated with the previous DDL. If the previous DDL was run on the same node, then there won't be any additional wait. But if the previous DDL was run on some other node, then this node must wait until it catches up replication changes from the other node. This improves handling of multiple concurrent DDL operations across the BDR Group which would previously result in global lock timeout, but now may cause additional wait in case of a large replication lag between the pair of nodes.

- Fix incorrect handling of MAXVALUE and MINVALUE for galloc sequences (RM14596)

- Fix concurrent action of nextval/currval when changing sequence kind (RT68438)
  During nextval/currval calls, the sequence kind was fetched prior to locking the sequence, which allowed a concurrently executed call to bdr.alter_sequence_set_kind() to influence the returned value incorrectly.

- Disallow SCHEMA and SEQUENCE RENAME for galloc sequences (RM15554)
  The consistent range allocator for galloc sequences identifies the sequence using schema name and sequence name, so changing the name of either of those could break it.
- Adjust chunk size of a galloc seq chunk if type was changed (RT68470, RM18297)
- Don't drop `bdr_local_only_origin` in `bdr_init_physical`
  Also recreate it in upgrade script in case previous version of BDR has dropped it.
- Ensure there is at least one transaction after wal switch in `bdr.move_group_slot_all_nodes` (RT67591, RM17356)
  This will move the group slot forward more hastily on servers with low concurrent activity.
- Fix `PART_CATCHUP` node state handling (RM17418)
  In 3.6.20 different nodes might have had different ideas about what to do when there is not fully joined node during part of another node which could cause the part operation to stall indefinitely with only possible solution being forced part. 3.6.21 leaves the decision on what to do next on Raft leader and other nodes just follow, so the decision making is consistent across whole BDR group.
- Fix incorrect cache handling for sessions established prior to BDR node creation (RT68499).

Improvements

- Allow use of CRDTs when BDR extension installed but without any node (RM17470)
  Previously, restoring CRDT values on a node with BDR extension installed, but without a node created, would throw an ERROR when the CRDT data type requests the node id. We now store an `Invalid0id` value when the node id is not available. If the node is subsequently added to a BDR cluster, when the CRDT column is updated, `Invalid0id` will be replaced by a normal node id.
- Check validity of options when altering a galloc sequence (RM18301, RT68470)
  Galloc sequences do not accept some options, so warn the user in case invalid options are used, as already occurs with timeshard sequences.
- `resynchronize_table_from_node()` freezes the table on target node (RM15987) When we use this function the target table is truncated first and then copied into on the destination node. This activity additionally FREEZEs the tuples as the data is loaded. This then avoids a rewriting the target table to set hint bits, as well as avoiding high volume WAL writes which would occur when the table was first used after resyncronizing.
- Create a virtual sequence record on other nodes (RM16008, RT68438, RT68432)
  If we create a galloc sequence and try to use its value in the same transaction block, then because it does not exist yet on other nodes, it used to error out with "could not fetch next sequence chunk" on the other nodes. We solve this by creating a virtual record on the other nodes.
- Add `start` parameter to `bdr.alter_sequence_set_kind()` (RT68430)
  Allows specifying new starting point for galloc and local sequenes in single step with changing the sequence kind.

- Add an internal consistent consensus-based key-value store for use by HARP (RM17825)
  This is not meant for direct use by users, but enables features for additional tooling.
- Improve latency of global locks in cases where other nodes respond very quickly, such as when there is no replication lag
- Throw errors for configuration paarmeters that were supported by BDR1 and BDR2, but are no longer supported by BDR3 (RT68529). Since PostgreSQL does not throw an ERROR/WARNING while setting an extension qualified, non-existent parameter, users with BDR1/BDR2 experience may fail to notice that the parameter is not supported by BDR3. We now explicitly catch all attempts to use parameter names that match unsupported BDR1/BDR2 configuration parameters and throw an ERROR.
- Document known issues in “Appendix C” section of documentation (RM18169)

BDR 3.6.20

This is a security and maintenance release for BDR 3.6 which also includes various minor features.

Additional Actions

- Run LiveCompare following logical join to handle rare divergent errors (RT67307)
  When running a logical join when nodes are down or have large replication lag can cause divergence of concurrent transactions that conflict during join. When running a logical join from a source to a target node, changes made on other nodes that were still in-flight could cause conflicts when applied; if conflicts occurs, conflict resolution might diverge for those rows. Running LiveCompare immediately following the join will clean up any such issues; this can be carefully targeted using a conflict logging table. Required actions are now fully documented.

Resolved Issues

- Ignore self-conflicts during logical join that could lead to data loss (RT67858)
  Changes made to tables during logical join could conflict with the original rows, as a result of rewriting the commit timestamp onto the target node. Explicitly ignore such conflicts seen while in join catchup mode, avoiding the associated data loss.
- Ensure origin position messages cannot be skipped during join (RT67858)
  Progress watermark messages are now written to WAL transaction stream rather than being issued asynchronously via the messaging layer. During join we now wait for the joining node to see at least one watermark to ensure no timing window exists that could lead to skipping the origin position during join, thus avoiding data loss.
- Resilience against `idle_in_transaction_session_timeout` (RM13649, RT67029, RT67688)
  Set `idle_in_transaction_session_timeout` to 0 so we avoid any user setting that could close the connection and invalidate the snapshot.

- Allow early `part_node` to interrupt a hanging join (RT67980, RM16659)
  In case a new node failing to join and not making any progress, attempting to remove it with plain (non-forced) `bdr.part_node` could lead to a hang. Resolve this by skipping the `PART_CATCHUP` phase for such a node, as it did not possibly produce any transactions to catch up to.
- Warn when joining to BDR group with versions of BDR which have known consistency issues
  The above fixes for join process only work if whole BDR node is upgraded to the 3.6.20 or higher (protocol version 12+), so we warn about joining to groups with older versions. In-place upgrades work normally.
- Fix minor issues detected by Coverity scanner.

Improvements

- Ignore bdr.assess_update_replica_identity parameter in writer worker processes (RM15983) (EE)
  Setting this in postgresql.conf can cause a potential outage if executed by writer worker processes. Hence, writer worker processes ignore this parameter.
- CentOS 8 is now supported, starting with this release.
- Improve diagnostic logging of the join process (RT67858)
  Include the fact that we retry the join state requests after failure in the error message and raise the log level of slot/origin creation/movement from DEBUG to LOG so that it's always logged by default.

BDR 3.6.19

This is a security and maintenance release for BDR 3.6 which also includes various minor features.

Resolved Issues

- SECURITY: Set search_path to empty for internal BDR SQL statements (RM15373)
  Also, fully qualify all operators used internally. BDR is now protected from attack risks identified in CVE-2018-1058, when the user application avoids the insecure coding practices identified there. See BDR Security chapter for further explanation.
- SECURITY: Raise required privileges for BDR admin functions (RM15542)
  When executed by roles other than superuser or bdr_superuser:
  - bdr.alter_table_conflict_detection needs table owner
  - bdr.column_timestamps_enable needs table owner
  - bdr.column_timestamps_disable needs table owner
  - bdr.drop_trigger needs table owner
  - bdr.alter_sequence_set_kind needs sequence owner
  - bdr.global_lock_table needs UPDATE, DELETE or TRUNCATE (like LOCK TABLE)

  - bdr.create_conflict_trigger needs TRIGGER permission on the table and EXECUTE permission on the function
  - bdr.create_transform_trigger needs TRIGGER permission on the table and EXECUTE permission on the function

A new GUC `bdr.backwards_compatibility` allows to skip this newly introduced check for existing clients requiring the former behavior.

- Resolve a hang possible after multiple global lock releases (RT67570, RM14994)
  A bug in the code path for releasing a global lock after a timeout led to overriding the backend's PID with a value of `-1`, also showing up in the `waiters` list of `bdr.global_locks`. This in turn crippled the waiters list and ultimately led to an infinite loop. This release fixes the override, which is the original cause of this hang and correctly removes entries from the lock wait list.
- Correct parsing of BDR WAL messages (RT67662)
  In rare cases a DDL which is replicated across a BDR cluster and requires a global lock may cause errors such as "invalid memory alloc request size" or "insufficient data left in message" due to incorrect parsing of direct WAL messages. The code has been fixed to parse and handle such WAL messages correctly.
- Fix locking in ALTER TABLE with multiple sub commands (RM14771) Multiple ALTER TABLE sub-commands should honor the locking requirements of the overall set. If one sub-command needs the locks, then the entire ALTER TABLE command needs it as well.
- Fix bug in example of ALTER TABLE ... ADD COLUMN workaround (RT67668)
  Explain why bdr.global_lock_table() is needed to avoid concurrent changes that cause problems, in that case.
- Fix a hang after promotion of a physical standby (RM15728)
  A physical standby promoted to a BDR node may have failed to start replicating due to the use of stale data from an internal catalog cache.
- Fix crash when bdr.trigger_get_type() is called by itself (RM15592) Calling bdr.trigger_get_type() outside a streaming trigger function would cause a crash. Fixed the function to return NULL when called outside a streaming trigger function.

Improvements

- bdr.trigger_get_origin_node_id() allows preferred-node resolution (RM15105, RT67601)
  Some customers have a requirement to resolve conflicts based upon the node that is the source of the change. This is also known as trusted source, trusted site or AlwaysWins resolution. Previous versions of BDR allowed these mechanisms with 2 nodes; this new function allows this option with any number of nodes. Examples are documented.
- BDR now accepts URIs in connection strings (RM14588)
  All connection strings can now use the format URI "postgresql://..."

- New function bdr.resynchronize_table_from_node() (RM13565, RT67666, RT66968) allows a single table to be truncated and then resynced from a chosen node, while holding a global dml lock. This allows a table to be resynchronized following a data divergence or data corruption without needing to regenerate the whole node. Foreign Keys are removed and re-enabled afterwards.
- Improve filtering of changes made by explicitly unreplicated transactions (RM15557)
  Previously changes made by transactions using bdr.xact_replication = off or by bdr.difference_fix transactions would be sent to the remote node, generating spurious conflicts and wasting effort. Changes are now filtered on the source node instead, improving performance.
- Initial and periodic transaction status checks use async libpq (RM13504) (EE)
  With CAMO enabled, the status of in-flight transactions is checked against a partner node. This uses an standard Postgres connection via libpq, which used to block the PGL manager process. This release changes the logic to use asynchronous libpq to allow the PGL manager to perform other tasks (e.g. process Raft messages) while that status check is performed. This reduces chances of timeouts or deadlocks due to a more responsive PGL manager process.
- Additional message fields assist diagnosis of DDL replication issues (RM15292)
- Clarify documentation regarding privileges required for BDR users (RT67259, RM15533)

BDR 3.6.18

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

Improvements

- Add synchronize_structure option to join_node_group (RM14200, RT67243)
  New synchronize_structure option can be set to either ‘all’ or ‘none’, which either sychronizes the whole schema or copies no DDL. This allows for rolling application schema upgrades to be performed with a user-managed schema (DDL) change step.
- Make bdr_difference_fix_* functions use pre-created local origin (RM14189)
  The bdr_difference_fix_* family of functions used to create a local origin to carry out conflict fixes. We now pre-create “bdr_local_only_origin” local origin at extension creation time. This same local origin is used by the above functions now.
- Adjust monitored values in bdr.monitor_group_versions() (RM14494)
  We no longer report CRITICAL when pglogical version different to bdr version, which is actually not important. We now report WARNING if BDR editions differ between nodes.
- Substantial formatting corrections and spelling check of documentation

Resolved Issues

- Fix node join so it uses only `bdr_superuser` permissions (RM14121, RT67259)
  This affects the `join_target_dsn` connection of the `join_node_group` function, which has been fixed to work with only `bdr_superuser` right for the role used to connect.
- GRANT EXECUTE on bdr.show_subscription_status TO bdr_real_all_stats (RT67360, RM14624)
  This allows both bdr_read_all_stats and bdr_monitor roles to access the bdr.subscription_summary view
- Fix failure of bdr_init_physical to copy data columns using BDR types (RM14522)
  bdr_init_physical now uses bdr.drop_node() rather than DROP EXTENSION, which caused all columns using BDR datatypes such as CRDTs to be silently dropped from tables.
- Fix failure in 3.6.17 upgrade script caused by views referencing CRDTs (RT67505)
  Upgrade script now executed only on tables and mat views. Upgrade failure may give a spurious error such as "ERROR: BDR global lock manager not initialized yet"
- Set non-join subscriptions to CATCHUP state rather than INIT state at startup
  Avoids a rare but possible case of copying metadata twice during node join.
- Fix lookup for a galloc sequence when BDR catalogs are absent. (RT67455, RM14564)
  This might cause a query on a sequence to throw an error like "cache lookup failed for relation . . . " when bdr library is added to shared_preload_libraries but BDR extension is not created.
- Allow ALTER TABLE ALTER COLUMN with BDR loaded but not initialized (RM14435)
  With the BDR extension loaded, but no local BDR node created, the DDL replication logic now still allows execution of an ALTER TABLE ALTER COLUMN operation.
- LOCK TABLE warning not shown when BDR node is not created (RM14613)
  Assess LOCK TABLE statement does not show when bdr.assess_lock_statement is set to a value other than 'ignore' until BDR node is created.
- Prevent a NULL dereference in consensus_disable (RM14618)
  `bdr.consensus_disable` expected the consensus process to be running. Fix it to prevent a segfault if that's not the case when the function is called.

BDR 3.6.17

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

Improvements

- Allow ALTER TABLE ALTER COLUMN TYPE with rewrite when not replicating DDL (EE) (RM13244)
  In some cases, in controlled DBA environments, it is possible to change the type of a column to an

implicitly castable one by adopting a rolling upgrade for the type of this column in a non replicated environment on all the nodes one by one. We allow concurrent activity on this table on other nodes during the rewrite. Also note that such ALTER commands cannot be run within transaction blocks.

- Add conflict logging configuration view (RM13691, RT66898)
  Add `bdr.node_log_config` view that shows information on the conflict logging configuration.
- Add new group monitoring views and functions (RM14014)
  These views and functions report the state of the BDR installation, replication slots and consensus across all nodes in the BDR group.
- Add current state of DDL replication related configuration parameters to log context (RM13637)
  Improves troubleshooting.

Resolved Issues

- Don't drop existing slot for a joining node (RM13310, RT67289, RT66797)
  This could have caused inconsistencies when node was joined using `bdr_init_physical` because it precreated the slot for new node which was supposed to be reused during join, instead it was dropped and recreated. We now keep the slot correctly which ensures there are no inconsistencies.
- Fix restart of CAMO node despite missing partner node (EE) (RM13899, RT67161)
  Prevent an error looking up the BDR node configured as a CAMO origin. In case the node got dropped, it does not exist, but might still be configured for CAMO.
- Fix locking in `bdr.column_timestamps_enable()` (EE) (RT67150)
  Don't hold same transaction and session level locks otherwise `PREPARE`, CAMO and Eager replication can't work for transactions where this is used.
- Don't try to apply BDR conflict resolution to PGL-only subscriptions (RT67178)
  BDR should only be active on BDR subscriptions, not pglogical ones.
- Let the CAMO partner return the final decision, once learned (RM13520)
  If an origin node switches to Local mode, temporarily dropping CAMO protections, it's possible for the CAMO partner to provisionally abort the transaction internally, but actually commit it eventually (to be in sync with the origin node). In earlier releases, this was not recorded leading to the status query function to continue to return an "aborted" result for the transaction. This release allows the final commit decision to override the provisional abort internally (catalog table bdr.node_pre_commit).
- Make CLCD/CRDT data types properly TOAST-able (EE) (RM13689)
  CLCD/CRDT data types were defined as using PLAIN storage. This can become as issue with a table with too many columns or if a large number of nodes are involved. This is now solved by converting these data types to use EXTENDED storage thus allowing for large sized values.

- Ensure duplicate messages are not received during node promotion (RM13972) Send a watermark from join source to the joining node during catchup phase of join to ensure it learns about current replication positions of all other nodes even if there are no data to forward from them during the catchup. Otherwise we might ask for older lsns during the promotion and receive duplicate messages and fail the join.
- Fix errors when `bdr.move_group_slot_all_nodes` is called with no BDR node present in the database (RT67245)
  Allows setting up physical standbys of future BDR master before creating the BDR node.
- Make sure table has a `PRIMARY KEY` when CLCD is turned on (EE)
  This is sanity check that prevents user from enabling CLCD on tables without a `PRIMARY KEY` as that would break the conflict detection for such tables.
- Automatically disable CAMO for non-transactional DDL operations (EE)
  Several DDL operations are not allowed within a transaction block and as such cannot reasonably benefit from the protection that CAMO offers. Automatically disable CAMO for these, so as to avoid "cannot PREPARE" errors at COMMIT time.

BDR 3.6.16

BDR 3.6.16 is the sixteenth minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified previously.

Improvements

- Add `bdr.alter_table_conflict_detection()` (RM13631)
  This function unifies the UI for changing conflict detection method for individual tables. Allows choice between origin based, row_version based and column level based (EE-only) conflict detection using same interface. The old functions are still supported, although they should be considered deprecated and will be removed in BDR 3.7.
- Add `bdr.default_conflict_detection` configuration option (RM13631)
  Related to the above `bdr.alter_table_conflict_detection()` function, the new configuration option allows setting the default conflict detection method for newly created tables.
- Change how forced part node works (RM13447)
  Forced node part will now first try to get consensus for parting and only do the local change if the consensus fails or if it's called for node which already started consensus based part process but the process has stuck on one of the steps.
- Automatically drop `bdr-enterprise` extension when dropping the `bdr` extension (RM13703)
  This improves usability when trying to drop the bdr extension without cascade, which is useful for example when user wants to keep the pglogical node associated with BDR.

- Improve error reporting when joining node with same name as existing active node (RM13447, RT66940)
  The previous error message was confusing as it made it seem like BDR does not allow node name reuse at all (which it does).
- Set application_name in `bdr_init_physical`
  Helps when diagnosing issues with this tool.
- Improve documentation of `ALTER TABLE` limitations (RM13512, RM13244, RT66940)
  Including new workaround for changing length of varchar columns.

Resolved Issues

- Fix pg_dump for BDR galloc sequences (RM13462, RT67051)
  Galloc sequences internally store extra data in sequence heap; BDR now hides the extra data from SELECTs so that queries on the sequence (which can be normal user query or a query from pg_dump for example) only show the usual sequence information.
- Fix enforcement of REPLICA IDENTITY FULL for CLCD
  Advanced conflict-handling approaches (CLCD, CRDT) require the table to have REPLICA IDENTITY FULL. However due to how the features initially evolved independently, this was not enforced (and documented) properly and consistently. We now correctly enforce the REPLICA IDENTITY FULL for CLCD for every table.
- Fix node name reuse of nodes which were used as join sources for other existing nodes in a BDR group (RM12178, RM13447)
  The source nodes have special handling so we need to make sure that newly joining node is not confused with node of same name that has been parted.
- Apply local states for existing nodes on newly joining node (RT66940)
  Otherwise decision making in during the join process might use wrong state information and miss some tasks like slot creation or subscription creation.
- Correctly clean node-level log filters and conflict resolver configuration (RM13704)
  This solves issues when trying to drop BDR node without dropping associated pglogical node and later recreating the BDR node again.
- Prevent a segfault in Raft on the parted BDR node (RM13705)

BDR 3.6.15

BDR 3.6.15 is the fifteenth minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified previously.

Improvements

- Keep a permanent log of all resolved CAMO decisions (RM12712)
  Record every decision taken by the CAMO partner when queried by `bdr.logical_transaction_status`, i.e. in the failover case.
- Add functions for enabling/disabling row version tracking (RM12930)
  Easier to use and less error prone interface than manually adding column and trigger.
- Add `currval()` and `lastval()` support for `timeshard` and `galloc` sequences (RM12059)
- Add `pglogical.min_worker_backoff_delay` setting to rate limit background worker relaunches, and `pglogical.worker_tasks` diagnostic view for background worker activity. See pglogical 3.6.15 release notes and documentation for details.

Resolved Issues

- Prevent buffer overrun when copying a TOAST column value inside the walsender output plugin (RT66839)
  This fixes issue that resulted in walsender crashes with certain types of workloads which touch TOASTed columns.
- Fix “type bdr.column_timestamps not found” error when bdr-enterprise extension is not installed when bdr enterprise library is in shared_preload_libraries (RT66758, RM13110)

BDR 3.6.14

BDR 3.6.14 is a critical maintenance release of the BDR 3.6 series. This release includes major fixes for CAMO and other features as well as minor new features.

Improvements

- Add `bdr.camo_local_mode_delay` to allow throttling in CAMO Local mode (RM12402)
  Provides a simple throttle on transactional throughput in CAMO Local mode, so as to prevent the origin node from processing more transactions than the pair would be able to handle with CAMO enabled.
- Add `bdr.camo_enable_client_warnings` to control warnings in CAMO mode (RM12558)
  Warnings are emitted if an activity is carried out in the database for which CAMO properties cannot be guaranteed. Well-informed users can choose to disable this if they want to avoid such warnings filling up their logs.
- Warn on unrecognized configuration settings
- Move ‘loading BDR’ message earlier in startup messages

- Significantly enhance docs for Row Version Conflict Detection (RT66493)
- Clarify docs that NOTIFY is not possible with CAMO/Eager
- Add `global_lock_request_time`, `local_lock_request_time` and `last_state_change_time` columns to `bdr.global_locks` view for lock monitoring and diagnostic use.
- Add SQL functions for export/import of consensus snapshot (RM11433)
  These functions allow for manual synchronization of BDR system catalogs in case of corruption or user mistake.

Resolved Issues

- UPDATEs skipped on the partner node because remote_commit_ts set incorrectly (RM12476)
  Commit timestamps were unset in some CAMO messages, leading to losing last-update-wins comparisons that they should have won, which meant some UPDATEs were skipped when an UPDATE happened concurrently from another master. This doesn't occur normally in an AlwaysOn cluster, though could occur if writes happen via a passive master node.
- Only resolve those prepared transactions for which controlling backend is gone (RM12388)
  This fixes a race condition between the pglogical manager process and the user backend running a CAMO transaction. A premature attempt by the manager process to resolve a prepared transaction could lead to the transaction getting marked as aborted on the partner node, whereas the origin ends up committing the transaction. This results in data divergence. This fix ensures that the manager process only attempts to resolve prepared transactions for which the controlling user backend has either exited or is no longer actively managing the CAMO transaction. The revised code also avoids taking ProcArrayLock, reducing contention and thus improving performance and throughput.
- Prevent premature cleanup of commit decisions on a CAMO partner. (RM12540)
  Ensure to keep commit or abort decisions on CAMO or Eager All Node transactions in bdr.node_pre_commit for longer than 15 minutes if there is at least one node that has not learned the decision and may still query it. This eliminates a potential for inconsistency between the CAMO origin and partner nodes.
- 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.
- Fix potential divergence by concurrent updates on toasted data from multiple nodes (RM11058)
  This can occur when an UPDATE changes one or more toasted columns, while a concurrent, but later UPDATE commits on a different node. This occurs because PostgreSQL does not WAL log TOAST data if it wasn't changed by an UPDATE command. As a result the logically decoded rows have these columns marked as unchanged TOAST and don't contain the actual value. Fix is handled automatically on BDR-EE, but on BDR-SE additional triggers need to be created on

tables that publish updates and that have toastable data (this is also done automatically). The additional check has a small but measurable performance overhead. Logged data will increase in affected cases only. We recommend tuning `toast_tuple_target` to optimize storage. Tables with `REPLICA IDENTITY FULL` are not affected by this issue or fix.

- Properly close connections after querying camo partner to avoid leak. (RM12572)
- Correct `bdr.wait_for_apply_queue` to respect the given LSN (RM12552)
  In former releases, the `target_lsn` argument was overridden and the function acted the same as if no `target_lsn` had been given.
- Ignore progress messages from unknown nodes (RT66461)
  Avoids problems during node parting.
- Make bdr.xact_replication work with ALTER TABLE and parallel query (RM12489)

BDR 3.6.12

BDR 3.6.12 is the twelfth minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified previously.

Improvements

- Apply `check_full_row` on `DELETE` operations (RT66493)
  This allows detection of `delete_recently_updated` conflict even if the `DELETE` operation happened later in wall-clock time on tables with full row checking enabled.
- Improve Global DML lock tracing
  Add more information to the Global DML Lock trace to help debugging global locking issues more effectively.
- Validate replication sets at join time. (RM12020, RT66310)
  Raise an ERROR from `bdr.join_node_group()` if the joining node was configured to subscribe to non-default replication sets by using `bdr.alter_node_replication_sets()` before join but some of the subscribed-to replication sets are missing.

On prior releases the joining node might fail later in the join process and have to be force-parted. Or it might appear to succeed but join with empty tables.

Resolved Issues

- Fix crash in `bdr.run_on_all_nodes` (RM12114, RT66515)
  Due to incorrect initialization the `bdr.run_on_all_nodes` could have previously crashed with segmentation fault in presence of `PARTED` nodes.

- Don't broadcast the epoch consumed WAL messages (RM12042)
  Broadcasting the message to all nodes could result in some nodes moving the Global DDL Lock Epoch forward in situations where it wasn't safe to do so yet, resulting in lowered protection against concurrent DML statements when running a statement that requires a Global DML Lock.
- Fix global locking on installations with multiple BDR nodes on single PostgreSQL instance
  The global locking could get spurious timeouts because the lock messages contained wrong node id if there were more than one BDR node on a single PostgreSQL instance.
- Fix typos in some example SQL in docs

BDR 3.6.11

BDR 3.6.11 is the eleventh minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified previously.

Improvements

- Support APIs for PostgreSQL 11.6
- Allow the use of "-"(hyphen) character in the node name (RM11567, RT65945)
  If a pglogical3 node would have been created with a hyphen in the node name BDR couldn't create the node on that database.
- Don't generate `update_origin_change` conflict if we know the updating node has seen the latest local change (RM11556, RT66145)
  Reduces conflict resolution overhead and logging of `update_origin_change` when the conflict can be shown to be false-positive. This does not completely remove false-positives from `update_origin_change` but reduces their occurrence in presence of UPDATES on older rows. This reduces conflict log spam when origin changes for older rows. Also, conflict triggers will be called significantly fewer times.
- Extend `bdr.wait_for_apply_queue` to wait for a specific LSN (RM11059, RT65827)
- Add new parameter `bdr.last_committed_lsn` (RM11059, RT65827)
  Value will be reported back to client after each COMMIT, allowing applications to perform causal reads across multiple nodes.
- Add status query functions for apply and received LSN (RM11059, RM11664)
  New functions `bdr.get_node_sub_receive_lsn` and `bdr.get_node_sub_apply_lsn` simplify fetching the internal information required for HAproxy health check scripts.
- Add sum() and avg() aggregates for CRDT types (RM11592, RT66168)
- Speed up initial synchronization of replication slots on physical standby (RM6747)
- Add `bdr.pg_xact_origin` function to request origin for an xid (RM11971)

- Add `bdr.truncate_locking` configuration option which sets the `TRUNCATE` command's locking behavior (RT66326)
  This configuration option determines whether (when `true`) `TRUNCATE` obeys the `bdr.ddl_locking` setting which is the new, safe behavior or if (when `false`, the default) never does any locking, which is the old, potentially unsafe behavior.
- Allow conflict triggers to see commit timestamp of `update_recently_deleted` target rows (RM11808, RT66182)

Resolved Issues

- Add hash/equality opclass for the column_timestamps data type (RT66207)
  REPLICA IDENTITY FULL requires comparison of all columns of a tuple, hence column_timestamps data type must support equality comparisons.
- Correct conflict docs for BDR-EE (RT66239, RM9670)
  Changes made in BDR3.5 were not correctly reflected in conflict docs
- Don't check protocol version for galloc sequences during initial sync (RM11576, RT65660)
  If galloc sequences already exist, bdr_init_physical doesn't need to recheck protocol versions.
- Fix galloc sequence chunk tracking corruption on lagging nodes (RM11933, RT66294)
  In presence of node with lagging consensus the chunk tracking table would diverge on different nodes which then resulted in wrong chunks being assigned on consensus leader change. As a result node might start generating already used sequence numbers. This fix ensures that the table never diverges.
- Fix galloc sequence local chunk information corruption (RM11932, RT66294)
  Make sure we correctly error out when in all cases request of new chunk has failed, otherwise we might assign bogus chunks to the sequence locally which would result in potentially duplicate sequence numbers generated on different nodes.
- Fix a case where the consensus worker event loop could stall in the message broker when trying to reconnect to an unreachable or unresponsive peer node by being more defensive about socket readability/writeability checks during the libpq async connection establishment phase. (RM11914)

This issue is most likely to arise when a peer node's underlying host fails hard and ceases replying to all TCP requests, or where the peer's network blackholes traffic to the peer instead of reporting a timely ICMP Destination Unreachable message.

Effect of the issue on affected nodes would result in operations which require consensus to either stall or not work at all - those include: DDL lock acquisition, Eager transaction commit, calling `bdr.get_consensus_status()` function, galloc sequence chunk allocation, leader election and BDR group slot advancing. This could have been visible to users as spurious lock timeout errors or increased lag for the BDR group slot.

- Fix a race condition with global locks and DML (RM12042)
  Prevent mismatching ordering of lock operations against DML with three or more concurrently writing nodes. This allows to properly protect a TRUNCATE against concurrent DML from multiple writer nodes.
- Repeat reporting of `local_node_id` to support transparent proxies (EE) (RM12025, RM12033)
  With CAMO enabled, BDR reports a `bdr.local_node_id` GUC to the client. To fully support transparent proxies like HAproxy, BDR now reports this value once per transaction in combination with `transaction_id`, to ensure a client doesn't ever return incorrect results from PQparameterStatus() because of a stale cache caused by missing a transparent connection switch.
- Fix global DDL and DML lock recovery after instance restart or crash (RM12042)
  Previous versions of BDR might not correctly block the writes against global lock if the node or apply worker restarted after the lock was acquired. This could lead to divergent data changes in case the protected command(s) were changing data concurrently.
- Fix global DDL and DML lock blocking of replication changes (RM12042)
  Previous versions of BDR would continue replication of changes to a locked table from other nodes. This could result in temporary replication errors or permanent divergent data changes if the transaction which acquired the global lock would be applied on some nodes with delay.
- Fix hang in cleanup/recovery of acquired global lock in the apply worker
  The apply worker which acquired global lock for another node could on exit leak the hanging lock which could then get "stolen" by different backend. This could cause the apply worker to wait for lock acquisition of same lock forever after restart.
- Don't hold back freezeLimit forever (EE) (RM11783)
  The Enterprise Edition of BDR holds back freeze point to ensure enough info is available for conflict resolution at all times. Make sure that we don't hold the freeze past xid wraparound warning limit to avoid loss of availability. Allow the limit to move forward gracefully to avoid risk of vacuum freeze storms.
- Properly close connections in `bdr.run_on_all_nodes`
  Removes log spam about connection reset by peer when `bdr.run_on_all_nodes` is used.
- Clarify docs that CREATE MATERIALIZED VIEW is not supported yet. (RT66363)

BDR 3.6.10

BDR 3.6.10 is the tenth minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified previously.

Improvements

- Add new optional performance mode for CAMO - remote_write (EE) (RM6749)
  This release enables a CAMO remote_write mode offering quicker feedback at time of reception

of a pre-commit message from the CAMO partner, rather than only after the application of the transaction. Significantly better performance in exchange for small loss of robustness.

- Defer switching to CAMO mode until the partner has caught up (EE) (RM9605/RT65000/RT65827)
  In async mode for improved availability, CAMO allows to switch to a local mode in case the CAMO partner is not reachable. When switching back, it may have to catchup before it can reasonably confirm new transactions from its origin. The origin now uses an estimate of the catchup time to defer the switch back to CAMO mode to eliminate transactions timing out due to the CAMO partner still catching up.

- Add functions wait_for_apply_queue and wait_for_camo_partner_queue (EE)
  Allows to wait for transactions already received but currently queued for application. These can be used to prevent stale reads on a BDR node replicated to in `remote_write` mode.

- Improve monitoring of replication, especially around catchup estimates for peer nodes (EE) (RM9798)
  Introduce two new views `bdr.node_replication_rates` and `bdr.node_estimates` to get a reasonable estimate of how far behind a peer node is in terms of applying WAL from this local node. The `bdr.node_replication_rates` view gives an overall picture of the outgoing replication activity in terms of the average apply rate whereas the `bdr.node_estimates` focuses on the catchup estimates for peer nodes.

- Support Column-Level Conflict Resolution for partitioned tables (EE) (RM10098, RM11310)
  Make sure that newly created or attached partitions are setup for CLCD if their parent table has CLCD enabled.

- Take global DML lock in fewer cases (RM9609).
  Don't globally lock relations created in current transaction, and also relations that are not tables (for example views) as those don't get data via replication.

Resolved Issues

- Disallow setting `external` storage parameter on columns that are part of a primary key (RM11336).
  With such a setting, any `UPDATE` could not be replicated as the primary key would not get decoded by PostgreSQL.

- Prevent ABA issue with `check_full_tuple = true`. (RM10940, RM11233)
  We only do the full row check if `bdr.inc_row_version()` trigger exists on a table from now on to prevent ABA issue when detecting conflict on UPDATEs that didn't change any data when `check_full_tuple` is set to `true`.

BDR 3.6.9

BDR 3.6.9 is the ninth minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified previously.

Improvements

- Parameters to help BDR assessment by tracking certain application events (EE) bdr.assess_update_replica_identity = IGNORE (default) | LOG | WARNING | ERROR Updates of the Replica Identity (typically the Primary Key) bdr.assess_lock_statement = IGNORE (default) | LOG | WARNING | ERROR Two types of locks that can be tracked are:
  - explicit table-level locking (LOCK TABLE . . .) by user sessions
  - explicit row-level locking (SELECT . . . FOR UPDATE/FOR SHARE) by user sessions (RM10812,RM10813)

Resolved Issues

- Fix crash MIN/MAX for gsum and pnsum CRDT types (RM11049)
- Disallow parted nodes from requesting bdr.part_node() on other nodes. (RM10566, RT65591)

BDR 3.6.8

BDR 3.6.8 is the eighth minor release of the BDR 3.6 series. This release includes a fix for a critical data loss issue as well as fixes for other issues identified with previous releases.

Improvements

- Create the `bdr.triggers` view (EE) (RT65773) (RM10874) More information on the triggers related to the table name, the function that is using it, on what event is triggered and what's the trigger type.

Resolved Issues

- Loss of TOAST data on remote nodes replicating UPDATEs (EE) (RM10820, RT65733)
  A bug in the transform trigger code path has been identified to potentially set toasted columns (very long values for particular columns) to NULL when applying UPDATEs on remote nodes, even when transform triggers have never been used. Only BDR-EE is affected and only when tables have a toast table defined and are not using REPLICA IDENTITY FULL. BDR3 SE is not affected by this issue. LiveCompare has been enhanced with damage assessment and data recovery features, with details provided in a separate Tech Alert to known affected users. This release prevents further data loss due to this issue.
- CAMO: Eliminate a race leading to inadvertent timeouts (EE) (RM10721)
  A race condition led to pre-commit confirmations from a CAMO partner being ignored. This in turn caused inadvertent timeouts for CAMO-protected transactions and poor performance in combination with `synchronous_replication_availability` set to `async`. This fixes an issue introduced with release 3.6.7.

- CAMO: Properly handle transaction cancellation at COMMIT time (EE) (RM10741)
  Allow the COMMIT of a CAMO-protected transaction to be aborted (more gracefully than via node restart or PANIC). Enable run-time reconciliation with the CAMO partner to make the CAMO pair eventually consistent.
- CAMO: Ensure the status query function keeps CAMO enabled. (EE) (RM10803)
  The use of the `logical_transaction_status` function disabled CAMO for the entire session, rather than just for the query. Depending on how a CAMO client (or a proxy in between) used the session, this could lead to CAMO being inadvertently disabled. This has been fixed and CAMO remains enabled independent of calls of this function.
- Eager: cleanup stale transactions. (EE) (RM10595)
  Ensures transactions aborted during their COMMIT phase are cleaned up eventually on all nodes.
- Correct TransactionId comparison when setting VACUUM freeze limit.
  This could lead to ERROR: cannot freeze committed xmax for a short period at xid wrap, causing VACUUMs to fail. (EE) (RT65814, RT66211)

BDR 3.6.7.1

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

Resolved Issues

- Prevent bogus forwarding of transactions from a removed origin. (RT65671, RM10605)
  After the removal of an origin, filter transactions from that origin in the output plugin, rather than trying to forward them without origin information.

BDR 3.6.7

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

Improvements

- CAMO and Eager switched to use two-phase commit (2PC) internally.
  This is an internal change that made it possible to resolve a deadlock and two data divergence issues (see below). This is a node-local change affecting the transaction's origin node exclusively and has no effect on the network protocol between BDR nodes. BDR nodes running CAMO now require a configuration change to allow for enough `max_prepared_transactions`; see Upgrading for more details. Note that there is no restriction on the use of temporary tables, as exists in explicit 2PC in PostgreSQL.

- Add globally-allocated range sequences (RM2125)
  New sequence kind which uses consensus between nodes to assign ranges of sequence numbers to individual nodes for each sequence as needed. Supports all of smallint, integer and bigint sequences (that includes serial column type).
- Implement Multi-Origin PITR Recovery (EE) (RM5826)
  BDR will now allow PITR of all or some replication origins to a specific point in time, providing a fully consistent viewpoint across all subsets of nodes. For multi-origins, we view the WAL stream as containing multiple streams all mixed up into one larger stream. There is still just one PIT, but that will be reached as different points for each origin separately. Thus we use physical WAL recovery using multiple separate logical stopping points for each origin. We end up with one LSN "stopping point" in WAL, but we also have one single timestamp applied consistently, just as we do with "single origin PITR".
- Add `bdr.xact_replication` option for transaction replication control
  Allows for skipping replication of whole transaction in a similar way to what `bdr.ddl_replication` does for DDL statements but it affects all changes including `INSERT/UPDATE/DELETE`. Can only be set via `SET LOCAL`. Use with care!
- Prevent accidental manual drop of replication slots created and managed by BDR
- Add `bdr.permit_unsafe_commands` option to override otherwise disallowed commands (RM10148)
  Currently overrides the check for manual drop of BDR replication slot in the Enterprise Edition.
- Allow setting `bdr.ddl_replication` and `bdr.ddl_locking` as `bdr_superuser` using the `SET` command
  This was previously possible only via the wrapper functions `bdr.set_ddl_replication()` and `bdr.set_ddl_locking()` which are still available.
- Improve performance of consensus messaging layer (RM10319, RT65396)

Resolved Issues

- Delete additional metadata in `bdr.drop_node` (RT65393, RM10346)
  We used to keep some of the local node info behind which could prevent reuse of the node name.
- Correctly synchronize node-dependent metadata when using `bdr_init_physical` (RT65221, RM10409)
  Synchronize additional replication sets and table membership in those as well as stream triggers and sequence information in `bdr_init_physical`, in a similar way to logical synchronization.
- Fix potential data divergence with CAMO due to network glitch (RM#10147)
  This fixes an data inconsistency that could arise between the nodes of a CAMO pair in case of an unreachable or unresponsive (but operational) CAMO partner and a concurrent crash of the CAMO origin node. An in-flight COMMIT of a transaction protected by CAMO may have ended up getting committed on one node, but aborted on the other, after both nodes are operational and connected.

- Fix a potential deadlock between a cross-CAMO pair (RM#7907)
  With two nodes configured as a symmetric CAMO pair, it was possible for the pair to deadlock, if both nodes were down and restarting, but both having CAMO transactions in-flight.
- Fix potential data divergence for Eager transaction in face of a crash (RM#9907)
  In case of a crash of the origin node of an Eager transaction just before the final local commit, such a transaction could have ended up aborted on the origin but committed on all other nodes. This is fixed by using 2PC on the origin node as well and properly resolving in-flight Eager transaction after a restart of the origin node.
- Correct handling of fast shutdown with CAMO transactions in-flight (RM#9556)
  A fast shutdown of Postgres on a BDR node that's in the process of committing a CAMO-protected transaction previously led to a PANIC. This is now handled gracefully, with the in-flight CAMO transaction still being properly recovered after a restart of that node.

BDR 3.6.6

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

Improvements

- Add `bdr.drop_node()` (RM9938)
  For removing node metadata from local database, allowing reuse of the node name in the cluster.
- Include `bdr_init_physical` in BDR-SE (RM9892)
  Improves performance during large node joins - BDR-EE has included this tool for some time.
- Enhance bdr_init_physical utility in BDR-EE (RM9988) Modify bdr_init_physical to optionally use selective pg_basebackup of only the target database as opposed to the earlier behavior of backup of the entire database cluster. Should make this activity complete faster and also allow it to use less space due to the exclusion of unwanted databases.
- TRUNCATE is now allowed during eager replicated transactions (RM9812)
- New `bdr.global_lock_table()` function (RM9735).
  Allows explicit acquire of global DML lock on a relation. Especially useful for avoidance of conflicts when using TRUNCATE with concurrent write transactions.
- New conflict type `update_pkey_exists` (RM9976)
  Allows conflict resolution when a `PRIMARY KEY` was updated to one which already exists on the node which is applying the change.
- Reword DDL locking skip messages and reduce log level
  The previous behavior was too intrusive.

- Add `bdr.apply_log_summary` (RM6596)
  View over `bdr.apply_log` which shows the human-readable conflict type and resolver string instead of internal id.
- Add `bdr.maximum_clock_skew` and `bdr.maximum_clock_skew_action` configuration options (RM9379)
  For checking clock skew between nodes and either warning or delaying apply in case the clock skew is too high.

Resolved Issues

- Move CRDT type operators from public schema to `pg_catalog` (RT65280, RM10027)
  Previously BDR operators were installed in public schema, preventing their use by servers implementing stricter security policies. No action required.
- Remember if unsupported Eager Replication command was run in current transaction.
  This allows us to prevent situations where an unsupported command was run while Eager Replication was turned off and later in the transaction the Eager Replication is turned on.
- Fix the "`!`" operator for `crdt_pnsum` data type (RM10156)
  It's the operator for resetting the value of the column, but in previous versions the reset operation didn't work on this type.

BDR 3.6.5

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

Improvements

- Allow late enabling of CAMO (RM8886)
  The setting `pg2q.enable_camo` may now be turned on at any point in time before a commit, even if the transaction already has an id assigned.
- Add version-2 KSUUIDs which can be compared using simple comparison operators (RM9662)
- New `delete_recently_updated` conflict type (RM9673/RT65063) Triggered by `DELETE` operation arriving out of order - the `DELETE` has an older commit timestamp than the most recent local `UPDATE` of the row. Can be used to override the default policy of `DELETE` always winning.
- Make bdr admin function replication obey DDL replication filters (RT65174)
  So that commands like `bdr.replication_set_add_table` don't get replicated to a node which didn't replicate `CREATE TABLE` in the first place.

- Don't require group replication set to be always subscribed by every node (RT65161)
  Since we now apply DDL replication filters to admin functions, it's no longer necessary to force group replication set to be subscribed by every node as other replication sets can be configured to replicate the admin function calls.
- Allow a few more DDL operations to skip the global DML lock
  The following DDL operations have been optimized to acquire only a global DDL lock, but not the DML one:
  - ALTER TABLE .. ALTER COLUMN .. SET STATISTICS
  - ALTER TABLE .. VALIDATE CONSTRAINT
  - ALTER TABLE .. CLUSTER ON
  - ALTER TABLE .. RESET
  - CREATE TRIGGER
- Add new BDR trigger that resolves Foreign Key anomalies on DELETE (RM9580)
- Add new function bdr.run_on_all_nodes() to assist monitoring and diagnostics (RM9945)
- Extend the CAMO reference client in C
  Allow setting a `bdr.commit_scope` for test transactions.
- Prevent replication of CLUSTER command to avoid operational impact
- To assist with security and general diagnostics, any DDL that skips replication or global DDL locking at user request will be logged. For regular users of non-replicated and/or non-logged DDL this may increase log volumes. Some log messages have changed in format. This change comes into effect when `bdr.ddl_locking = off` and/or `bdr.ddl_replication = off`.
- Greatly enhance descriptions of BDR admin functions with regard to (RM8345)
  their operational impact, replication, locking and transactional nature
- Detailed docs to explain concurrent Primary Key UPDATE scenarios (RM9873/RT65156)
- Document examples of using bdr.replication_set_add_ddl_filter() (RM9691)

Resolved Issues

- Rework replication of replication set definition (RT64451)
  Solves the issue with the replication set disappearing from some nodes that could happen in certain situations.
- Acquire a Global DML lock for these DDL commands for correctness (RM9650)
  - CREATE UNIQUE INDEX CONCURRENTLY
  - DROP INDEX CONCURRENTLY
  - bdr.drop_trigger() admin function since adding or removing any constraint could allow replication-halting DML

- Correctly ignore nodes that are parting or parted in the Raft code (RM9666/RT64891)
  Removes the excessive logging of node membership changes.
- Don't try to do CRDT/CLCD merge on `update_recently_deleted` (RM9674)
  It's strictly row-level conflict; doing a merge would produce the wrong results.
- Allow BDR apps that require standard_conforming_strings = off (RM9573/RT64949)
- Use replication slot metadata to postpone freezing of rows (RM9670) (EE-only)
  Otherwise an update_origin_change conflict might get undetected after a period of node downtime or disconnect. The SE version can only avoid this using parameters.
- Correct bdr_wait_slot_confirm_lsn() to wait for the LSN of last commit, rather than the LSN of the current write position. In some cases that could have released the wait earlier than appropriate, and in other cases it might have been delayed.

BDR 3.6.4

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

The Highlights of BDR 3.6.4

- Apply statistics tracking (RM9063)
  We now track statistics about replication and resource use for individual subscriptions and relations and make them available in the `pglogical.stat_subscription` and `pglogical.stat_relation` views. The tracking can be configured via the `pglogical.stat_track_subscripti` and `pglogical.stat_track_relation` configuration parameters.
- Support CAMO client protocol with Eager All Node Replication
  Extend `bdr.logical_transaction_status` to be able to query the status of transactions replicated in global commit scope (Eager All Node Replication). Add support for Eager All Node Replication in the Java CAMO Reference client.

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 being copied twice if both the root partition and individual partitions were published via the replication set. This is now solved; we only do the initial copy on the root partition if it's published.
- 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 merging of `crdt_pnsum` data type (RT64975)
  The internal index was handled wrongly, potentially causing a segmentation fault; this is now resolved.
- Fix cache lookup failed on database without BDR extension installed (RM9217)
  This could previously lead to errors when dropping tables on a PostgreSQL instance which has the BDR library loaded but does not have the extension installed.
- Fix permission issues on `bdr.subscription_summary` (RT64945)
  No need to have permissions on `pglogical.get_sub_progress_timestamp()` to use this view anymore.
- Cleanup prepared Eager All Node transactions after failures (RM8996)
  Prevents inconsistencies and hangs due to unfinished transactions after node or network failures. Uses Raft to ensure consistency between the nodes for the cleanup of such dangling prepared transactions.

Other Improvements

- The `replicate_inserts` option now affects initial COPY
  We now do initial copy of data only if the table replicates inserts.
- Lower log level for internal node management inside Raft worker (RT64891)
  This was needlessly spamming logs during node join or parting.
- Warn when executing DDL without DDL replication or without DDL locking
  The DDL commands executed without DDL replication or locking can lead to divergent databases and cause replication errors so it's prudent to warn about them.
- Allow create statistics without dml lock (RM9507)
- Change documentation to reflect the correct default settings for the update_missing conflict type.

BDR 3.6.3

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

The Highlights of BDR 3.6.3

- Add btree/hash operator classes for CRDT types (EE, RT64319) This allows the building of indexes on CRDT columns (using the scalar value) and the querying of them them using simple equality/inequality clauses, using the in GROUP BY clauses etc.
- Add implicit casts from int4/int8 for CRDT sum types (EE, RT64600)
  To allow input using expressions with integer and CRDT sum types together. For example: `CREATE TABLE t (c bdr.crdt_gsum NOT NULL DEFAULT 0);`

- New `update_recently_deleted` conflict type (RM8574)
  Conflicts are handled differently for the special case of `update_missing` when BDR detects that the row being updated is missing because it was just recently deleted. See UPDATE/DELETE Conflicts in the documentation for details.
- Allow DDL operations in CAMO protected transactions, making automatic disabling of CAMO obsolete (EE, RT64769)
- Add the connection status checking function `bdr.is_camo_partner_connected` for CAMO (EE). See the Commit At Most Once documentation for details.
- Persist the last_xact_replay_timestamp (RT63881)
  So that it's visible even if the subscription connection is down (or remote node is down).
- Major documentation improvements
  Copy-edit sentences to make more sense, add extra clarifying info where the original wording was confusing.

Resolved Issues

- Use group locking for global DML lock (RT64404)
  This allows better cooperation between the global DML locker and the writers which are doing catch up of the remaining changes.

Other Improvements

- Support mixed use of legacy CRDT types and new CRDT types which are in bdr schema
  Implicitly cast between the two so their mixed usage and potential migration is transparent.
- Improve static code scanning
  Every build is scanned both by Coverity and Clang scan-build.
- Log changes of `bdr.ddl_replication` and `bdr.ddl_locking`
  Helps with troubleshooting when divergent DDL was run.
- 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 a single source of chapter list and ordering for both PDF and HTML docs.

BDR 3.6.2

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

The Highlights of BDR 3.6.2

- All the SQL visible interfaces are now moved to the `bdr` schema (EE)
  The CRDT types and per column conflict resolution interfaces are now in the `bdr` schema instead of `bdr_crdt` and `bdr_conflicts`. The types and public interfaces still exist in those schemas for compatibility with existing installations, however their use is not recommended as they are

now deprecated and may be removed in a future release. Please use the ones in `bdr` schema. Documentation only contains references to the `bdr` schema now as well.
- Add `bdr.node_conflict_resolvers` view (RT64388)
  Shows current conflict resolver setting for each conflict type on the local node including the defaults.
- Add a CAMO Reference Client implementation in C and Java to the documentation.
- Support DEFERRED UNIQUE indexes
  They used to work only in limited cases before this release.

Resolved Issues

- Fix consensus request timeout during leader election (RT64569)
  The timeout wasn't applied when the leader was unknown leading to immediate failures of any action requiring consensus (for example global DDL locking). This is now resolved.
- Improve cleanup on failure during a DDL locked operation, This speeds up DDL locking subsystem recovery after error so that errors don't create a cascading effect.
- Unify the replication of admin function commands (RT64544)
  This makes the replication and locking behavior of administration function commands more in-line with DDL in all situations, including logical standby.
- Support covering UNIQUE indexes (RT64650)
  Previously, the covering UNIQUE indexes could result in ambiguous error messages in some cases.
- Switch to monotonic time source for Raft timing (RM6390)
  This improves reliability of Raft internal timing in presence of time jumps caused by NTPd and similar. As a result Raft reliability is improved in general.
- Improve locking in the internal connection pooler
  For more reliable messaging between nodes.
- Restore consensus protocol version on restart (RT64526)
  This removes the need for renegotiation every time a consensus worker or a node is restarted, making the features depending on newer protocol version consistently available across restarts.
- Correct automatic disabling and re-enabling of pg2q.enable_camo when using DDL in a transaction. Ensure it cannot be manually re-enabled within the same transaction.
- Fix handling of CAMO confirmations arriving early, before the origin starts to wait. This prevents timeouts due to such a confirmation being ignored.

BDR 3.6.1

BDR 3.6.1 is the first minor release of the BDR 3.6 series. This release includes minor new features and fixes including all the fixes from 3.6.0.1 and 3.6.0.2.

The highlights of 3.6.1

- Add `bdr.role_replication` configuration option (RT64330)
  The new option controls the replication of role management statements (`CREATE/ALTER/DROP/GRANT ROLE`).

This option is dependent on `bdr.ddl_replication` as the role management statements still obey the standard rules of the DDL replication. By default this is set to `on`, meaning that these statements are replicated if executed in a BDR-enabled database.

- Add `--standby` option to `bdr_init_physical` (RM8543, EE)
  Allows the creation of a logical standby using `bdr_init_physical`; previously only a full blown send/receive node could be created this way.
- Add `last_xact_replay_timestamp` to `bdr.subscription_summary` (RT63881)
  Shows the commit timestamp of the last replayed transaction by the subscription.
- Stop join on unrecoverable error (RT64463)
  Join might fail during the structure synchronization, which currently is an unrecoverable error. Instead of retrying like for other (transient) errors, just part the joining node and inform the user that there was an error.

Resolved Issues

- Improve the trigger security checking (RT64412)
  Allow triggers to have a different owner than the table if the trigger uses bdr or pglogical trigger functions, security definer functions (as those redefine security anyway) and also always allow replication set membership changes during initial replication set synchronization during the node join.
- Make BDR replicated commands obey `bdr.ddl_replication` (RT64479)
  Some of the BDR function calls (like `bdr_conflicts.column_timestamps_enable`) are replicated in a similar way as normal DDL commands including the DDL locking as appropriate. These commands in previous versions of BDR however ignored the `bdr.ddl_replication` setting and were always replicated. This is now fixed. In addition just like normal DDL, these commands are now never replicated from the logical standby.
- Don't try to replicate generic commands on global objects
  Several commands on global objects would be replicated even in situations where they shouldn't be because of how they are represented internally. Handling of the following commands has been fixed:
  - `ALTER ROLE/DATABASE/TABLESPACE ... RENAME TO`
  - `ALTER DATABASE/TABLESPACE ... OWNER TO`
  - `COMMENT ON ROLE/DATABASE/TABLESPACE`
  - `SECURITY LABEL ON ROLE/DATABASE/TABLESPACE`
- Properly timeout on CAMO partner and switch to Local mode (RT64390, EE)
  Disregard the connection status of other BDR nodes and switch to Local mode as soon as the designated CAMO partner node fails. Makes the switch to Local mode work in a four or more node cluster.

BDR 3.6.0.2

The BDR 3.6.0.2 release is the second bug-fix release in the BDR 3.6 series.

Resolved Issues

- Dynamic disabling of CAMO upon the first DDL (EE, RT64403)
- Fix hang in node join caused by timing issues when restoring Raft snapshot (RT64433)
- Fix the trigger function ownership checks (RT64412)
- Improve behavior of `promote_node` and `join_node_group` with `wait_for_completion := false`

BDR 3.6.0.1

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

Resolved Issues

- Support `target_table_missing` conflict for transparent partitioning (EE) (RT64389)
- Fix message broker sometimes discarding messages (common side-effect are DDL locking time-outs)
- Raft protocol negotiations improvements
- Fixed memory leak in tracing code
- Improve synchronous `remote_write` replication performance (RT64397)
- Fixed commit timestamp variant handling of CLCD (EE)
- Re-add support for binary protocol
- Correct Local mode for CAMO with `synchronous_replication_availability = 'async'` (EE)
- Disallow and provide a hint for unsupported operations in combination with CAMO (EE).
- Fix deadlock in `logical_transaction_status` (EE)

BDR 3.6.0

The version 3.6 of BDR3 is a major update which brings improved CAMO, performance improvements, better conflict handling and bug fixes.

The highlights of BDR 3.6

- Differentiate BDR RemoteWrite mode and set `write_lsn`
- Significant replication performance improvement
- Cache table synchronization state
- Only send keepalives when necessary
- Only do flush when necessary
- Serialize transactions in fewer cases in wal sender (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 per node and conflict type
- Improve conflict detection for updates
- Simplification of CAMO configuration (EE)
- Performance improvements for CAMO (EE)

Resolved issues

- Fix reporting of replay lag (RT63866)
- Fix CRDTs and conflict triggers for repeated UPDATEs of same row in transaction (RT64297)
- Don't try to replicate REINDEX of temporary indexes

Other improvements

- Improved vacuum handling of Raft tables
- Improve and clarify CAMO documentation (EE)

Appendix B: Conflict Details

This section documents in detail the behavior of BDR3 when **conflicts** occur.

For every isolation test, the *expected output* is displayed, with additional annotations commenting the context and the interpretation of the outcomes.

Each test is defined by a sequence of specific DML actions; the following table provides links to each relevant combination:

	INSERT	UPDATE	DELETE	TRUNCATE
INSERT	ii	iu	id	it
UPDATE	iu	uu	ud	ut
DELETE	id	ud	dd	dt
TRUNCATE	it	ut	dt	tt
INSERT-INSERT	iii	iiu	iid	iit
UPDATE-UPDATE	-	uuu	uud	uut
UPDATE-DELETE	-	-	-	udt
DELETE-UPDATE	-	duu	-	-
DELETE-DELETE	-	-	ddd	ddt
TRUNCATE-UPDATE	-	tuu	-	-
TRUNCATE-TRUNCATE	-	-	-	ttt

Test `two_node_dmlconflict_ii`

```
Parsed test spec with 2 sessions

starting permutation: s1i s2i s1w s2w s1s s2s
```

We insert a row into `node1`:

Node 1 (step `i`):

```
INSERT INTO test_dmlconflict VALUES('x', 1, 'foo');
```

We insert a row with the same primary key into `node2`:

Node 2 (step `i`):

```
INSERT INTO test_dmlconflict VALUES('y', 1, 'bar');
```

We wait until `INSERT` on `node1` is replicated to all other nodes:

Node 1 (step `w`):

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

We wait until INSERT on node2 is replicated to all other nodes:

Node 2 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

y               1               bar
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

y               1               bar

starting permutation: s1a s2a s1i s2i s1w s2w s1s s2s s1teardown s2teardown
```

Node 1 (step a):

```
SELECT bdr.alter_node_set_conflict_resolver('node1', 'insert_exists', 'skip');
alter_node_set_conflict_resolver

t
```

Node 2 (step a):

```
SELECT bdr.alter_node_set_conflict_resolver('node2', 'insert_exists', 'skip');
alter_node_set_conflict_resolver

t
```

We insert a row into node1:

Node 1 (step i):

```
INSERT INTO test_dmlconflict VALUES('x', 1, 'foo');
```

We insert a row with the same primary key into node2:

Node 2 (step i):

```
INSERT INTO test_dmlconflict VALUES('y', 1, 'bar');
```

We wait until INSERT on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until INSERT on node2 is replicated to all other nodes:

Node 2 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

x               1               foo
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

y               1               bar
```

Node 1 (step teardown):

```
SELECT bdr.alter_node_set_conflict_resolver('node1', 'insert_exists', 'update_if_newer');
alter_node_set_conflict_resolver

t
```

Node 2 (step teardown):

```
SELECT bdr.alter_node_set_conflict_resolver('node2', 'insert_exists', 'update_if_newer');
alter_node_set_conflict_resolver

t
```

Test two_node_dmlconflict_iu

```
Parsed test spec with 3 sessions

starting permutation: s3setup s1i s2w1 s2s s2u s2w s1w s1s s2s s3s s3teardown
```

We artificially introduce a 10 second replication delay between Node 1 and Node 2, to force conflicts due to a different replay order.

Node 3 (step `setup`):

```
SELECT pglogical.alter_subscription_disable
('bdr_postgres_bdrgroup_node1_node3');
UPDATE pglogical.subscription
SET sub_apply_delay = '10s'
WHERE sub_name = 'bdr_postgres_bdrgroup_node1_node3';
SELECT pglogical.alter_subscription_enable
('bdr_postgres_bdrgroup_node1_node3');

alter_subscription_disable

t
alter_subscription_enable

t
```

We insert a row into `node1`:

Node 1 (step `i`):

```
INSERT INTO test_dmlconflict VALUES('x', 1, 'foo');
```

We wait until the `INSERT` on `node1` is replicated to `node2`:

Node 2 (step `w1`):

```
SELECT * from pg_sleep(1);
pg_sleep
```

State of `node2`:

Node 2 (step `s`):

```
SELECT * FROM test_dmlconflict;
a               b               c

x               1               foo
```

On `node2` we update the row replicated from `node1`:

Node 2 (step `u`):

```
UPDATE test_dmlconflict set a='z' where b=1;
```

We wait until the UPDATE is replicated to all other nodes:

Node 2 (step w):

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

Then we wait until the insert from node1 is replicated to node3:

Node 1 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

z               1               foo
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

z               1               foo
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

x               1               foo
```

Node 3 (step teardown):

```
SELECT pglogical.alter_subscription_disable
('bdr_postgres_bdrgroup_node1_node3');
UPDATE pglogical.subscription
SET sub_apply_delay = '1s'
WHERE sub_name = 'bdr_postgres_bdrgroup_node1_node3';
SELECT pglogical.alter_subscription_enable
```

```
('bdr_postgres_bdrgroup_node1_node3');
SELECT bdr.alter_node_set_conflict_resolver('node3', 'update_missing', 'insert_or_skip');

alter_subscription_disable

t
alter_subscription_enable

t
alter_node_set_conflict_resolver

t

starting permutation: s3setup s3a s1s s1i s2w1 s2s s2u s2w s1w s1s s2s s3s s3teardown
```

We artificially introduce a 10 second replication delay between Node 1 and Node 2, to force conflicts due to a different replay order.

Node 3 (step setup):

```
SELECT pglogical.alter_subscription_disable
('bdr_postgres_bdrgroup_node1_node3');
UPDATE pglogical.subscription
SET sub_apply_delay = '10s'
WHERE sub_name = 'bdr_postgres_bdrgroup_node1_node3';
SELECT pglogical.alter_subscription_enable
('bdr_postgres_bdrgroup_node1_node3');

alter_subscription_disable

t
alter_subscription_enable

t
```

Node 3 (step a):

```
SELECT bdr.alter_node_set_conflict_resolver('node3', 'update_missing', 'insert_or_error');
alter_node_set_conflict_resolver

t
```

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

We insert a row into node1:

Node 1 (step i):

```
INSERT INTO test_dmlconflict VALUES('x', 1, 'foo');
```

We wait until the INSERT on node1 is replicated to node2:

Node 2 (step w1):

```
SELECT * from pg_sleep(1);
pg_sleep
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

x               1               foo
```

On node2 we update the row replicated from node1:

Node 2 (step u):

```
UPDATE test_dmlconflict set a='z' where b=1;
```

We wait until the UPDATE is replicated to all other nodes:

Node 2 (step w):

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

Then we wait until the insert from node1 is replicated to node3:

Node 1 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

z               1               foo
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

z               1               foo
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

z               1               foo
```

Node 3 (step teardown):

```
SELECT pglogical.alter_subscription_disable
('bdr_postgres_bdrgroup_node1_node3');
UPDATE pglogical.subscription
SET sub_apply_delay = '1s'
WHERE sub_name = 'bdr_postgres_bdrgroup_node1_node3';
SELECT pglogical.alter_subscription_enable
('bdr_postgres_bdrgroup_node1_node3');
SELECT bdr.alter_node_set_conflict_resolver('node3', 'update_missing', 'insert_or_skip');

alter_subscription_disable

t
alter_subscription_enable

t
alter_node_set_conflict_resolver

t
```

Test two_node_dmlconflict_id

```
Parsed test spec with 3 sessions

starting permutation: s1i s2w1 s2s s2d s2w s1w s1s s2s s3s s3teardown
alter_subscription_enable

t
```

We insert a row into a table on node1:

Node 1 (step i):

```
INSERT INTO test_dmlconflict VALUES('x', 1, 'foo');
```

We wait until the INSERT on node1 is replicated to node2:

Node 2 (step w1):

```
SELECT * from pg_sleep(1);
pg_sleep
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

x               1               foo
```

On node2 we delete the row replicated from node1:

Node 2 (step d):

```
DELETE from test_dmlconflict where b=1;
```

We wait until the DELETE is replicated to all other nodes:

Node 2 (step w):

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

Now we wait until the insert from node1 is replicated to node3 as well:

Node 1 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

x               1               foo
```

Node 3 (step `teardown`):

```
BEGIN;
SELECT pglogical.alter_subscription_disable('bdr_postgres_bdrgroup_node1_node3');
END;
UPDATE pglogical.subscription set sub_apply_delay = '1s' where sub_name = 'bdr_postgres_bdrgr
SELECT pglogical.alter_subscription_enable('bdr_postgres_bdrgroup_node1_node3');

alter_subscription_disable

t
alter_subscription_enable

t
```

Test `two_node_dmlconflict_it`

```
Parsed test spec with 3 sessions

starting permutation: s1i s2w1 s2s s2t s2w s1w s1s s2s s3s s3teardown
alter_subscription_enable

t
```

We `INSERT` a row into a table on `node1`:

Node 1 (step `i`):

```
INSERT INTO test_dmlconflict VALUES('y', 2, 'baz');
```

We wait until the `INSERT` on `node1` is replicated to `node2`:

Node 2 (step `w1`):

```
SELECT * from pg_sleep(1);
pg_sleep
```

State of `node2`:

Node 2 (step `s`):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

```
x               1               foo
y               2               baz
```

On node2 we truncate the test table after the INSERT from node1 is replicated:

Node 2 (step t):

```
TRUNCATE test_dmlconflict;
```

We wait until the TRUNCATE is replicated to all other nodes:

Node 2 (step w):

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

Now we wait until the INSERT from node1 is replicated to node3 as well:

Node 1 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

y               2               baz
```

Node 3 (step teardown):

```
BEGIN;
SELECT pglogical.alter_subscription_disable('bdr_postgres_bdrgroup_node1_node3');
END;
UPDATE pglogical.subscription set sub_apply_delay = '1s' where sub_name = 'bdr_postgres_bdrgr
```

```
SELECT pglogical.alter_subscription_enable('bdr_postgres_bdrgroup_node1_node3');

alter_subscription_disable

t
alter_subscription_enable

t
```

Test two_node_dmlconflict_uu

```
Parsed test spec with 2 sessions

starting permutation: s1u s2u s1w s2w s1s s2s
```

We UPDATE a row from node1:

Node 1 (step u):

```
UPDATE test_dmlconflict SET a = 'x' where b = 1;
```

We UPDATE a row from node2 concurrently:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y' where b = 1;
```

We wait until the UPDATE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the UPDATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

y               1               foo
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c

y              1              foo

starting permutation: s1a s2a s1u s2u s1w s2w s1s s2s s1teardown s2teardown
WARNING:  setting update_origin_change to skip may result in loss of UPDATE
DETAIL:  This results in loss of update in case conflict is falsely detected
HINT:  Use inc_row_version(), check_full_tuple and REPLICA IDENTITY FULL
```

Node 1 (step a):

```
SELECT bdr.alter_node_set_conflict_resolver('node1', 'update_origin_change', 'skip');
alter_node_set_conflict_resolver

t
WARNING:  setting update_origin_change to skip may result in loss of UPDATE
DETAIL:  This results in loss of update in case conflict is falsely detected
HINT:  Use inc_row_version(), check_full_tuple and REPLICA IDENTITY FULL
```

Node 2 (step a):

```
SELECT bdr.alter_node_set_conflict_resolver('node2', 'update_origin_change', 'skip');
alter_node_set_conflict_resolver

t
```

We UPDATE a row from node1:

Node 1 (step u):

```
UPDATE test_dmlconflict SET a = 'x' where b = 1;
```

We UPDATE a row from node2 concurrently:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y' where b = 1;
```

We wait until the UPDATE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the UPDATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

x               1               foo
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

y               1               foo
```

Node 1 (step teardown):

```
SELECT bdr.alter_node_set_conflict_resolver('node1', 'update_origin_change', 'update_if_newer
alter_node_set_conflict_resolver

t
```

Node 2 (step teardown):

```
SELECT bdr.alter_node_set_conflict_resolver('node2', 'update_origin_change', 'update_if_newer
alter_node_set_conflict_resolver

t
```

Test two_node_dmlconflict_uu_replayorder

```
Parsed test spec with 3 sessions

starting permutation: s1u s2w1 s2u s2w s1w s1s s2s s3s s3teardown
alter_subscription_enable

t
```

Node 1 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'bar';
```

We wait until the UPDATE on node1 is replicated to node2:

Node 2 (step w1):

```
SELECT * from pg_sleep(1);
pg_sleep
```

On node2 we update the same row updated by node1:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'z', b = '1', c = 'baz';
```

We wait until the UPDATE is replicated to all other nodes:

Node 2 (step w):

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

Now we wait until the UPDATE from node1 is replicated to node3 as well:

Node 1 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

z               1               baz
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

z               1               baz
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

z               1               baz
```

Node 3 (step teardown):

```
BEGIN;
SELECT pglogical.alter_subscription_disable('bdr_postgres_bdrgroup_node1_node3');
END;
```

```
UPDATE pglogical.subscription set sub_apply_delay = '1s' where sub_name = 'bdr_postgres_bdrgr
SELECT pglogical.alter_subscription_enable('bdr_postgres_bdrgroup_node1_node3');

alter_subscription_disable

t
alter_subscription_enable

t
```

Test two_node_dmlconflict_ud

```
Parsed test spec with 2 sessions

starting permutation: s1alc s1d s2u s1w s2w s1s s2s s1slc s1rlc
```

call bdr.alter_node_add_log_config():

Node 1 (step alc):

```
SELECT bdr.alter_node_add_log_config('node1','test_config','t','log_table');;
alter_node_add_log_config

t
```

We delete the only row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'bar';
```

Now we wait until the commit on node1 is updated on all nodes:

Node 1 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

Now we wait until the commit on node2 is replicated on all nodes:

Node 2 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

Read entries of the log_table at node1:

Node 1 (step slc):

```
SELECT nspname, relname, conflict_type, conflict_resolution, conflict_index FROM log_table;
nspname        relname        conflict_type  conflict_resolutionconflict_index

public         test_dmlconflict13             2
```

Remove log_config from node1:

Node 1 (step rlc):

```
SELECT bdr.alter_node_remove_log_config('node1','test_config');
alter_node_remove_log_config

t

starting permutation: s1urd s2urd s1d s2u s1w s2w s1s s2s s1urdteardown s2urdteardown
```

Node 1 (step urd):

```
SELECT bdr.alter_node_set_conflict_resolver('node1', 'update_recently_deleted', 'insert_or_sk
alter_node_set_conflict_resolver

t
```

Node 2 (step urd):

```
SELECT bdr.alter_node_set_conflict_resolver('node2', 'update_recently_deleted', 'insert_or_sk
alter_node_set_conflict_resolver

t
```

We delete the only row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'bar';
```

Now we wait until the commit on node1 is updated on all nodes:

Node 1 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

Now we wait until the commit on node2 is replicated on all nodes:

Node 2 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c

y              1              bar
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

Node 1 (step urdteardown):

```
SELECT bdr.alter_node_set_conflict_resolver('node1', 'update_recently_deleted', 'skip');
alter_node_set_conflict_resolver

t
```

Node 2 (step urdteardown):

```
SELECT bdr.alter_node_set_conflict_resolver('node2', 'update_recently_deleted', 'skip');
alter_node_set_conflict_resolver

t

starting permutation: s2u s1d s2w s1w s1s s2s
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'bar';
```

We delete the only row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

Now we wait until the commit on node2 is replicated on all nodes:

Node 2 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

Now we wait until the commit on node1 is updated on all nodes:

Node 1 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c


starting permutation: s1dru s2dru s1d s2u s1w s2w s1s s2s s1druteardown s2druteardown
```

Node 1 (step dru):

```
SELECT bdr.alter_node_set_conflict_resolver('node1', 'delete_recently_updated', 'skip');
alter_node_set_conflict_resolver

t
```

Node 2 (step dru):

```
SELECT bdr.alter_node_set_conflict_resolver('node2', 'delete_recently_updated', 'skip');
alter_node_set_conflict_resolver

t
```

We delete the only row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'bar';
```

Now we wait until the commit on node1 is updated on all nodes:

Node 1 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

Now we wait until the commit on node2 is replicated on all nodes:

Node 2 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

y               1               bar
```

Node 1 (step druteardown):

```
SELECT bdr.alter_node_set_conflict_resolver('node1', 'delete_recently_updated', 'update');
alter_node_set_conflict_resolver

t
```

Node 2 (step druteardown):

```
SELECT bdr.alter_node_set_conflict_resolver('node2', 'delete_recently_updated', 'update');
alter_node_set_conflict_resolver

t
```

Test two_node_dmlconflict_ud_replayorder

```
Parsed test spec with 3 sessions

starting permutation: s1u s2d s2w s1w s1s s2s s3s s3teardown
alter_subscription_enable

t
```

Node 1 (step u):

```
UPDATE test_dmlconflict SET  a = 'y', b = 1, c = 'bar';
```

On node2 we delete the same row updated by node1:

Node 2 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We wait until the DELETE is replicated to all other nodes:

Node 2 (step w):

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

Now we wait until the UPDATE from node1 is replicated to node3 as well:

Node 1 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

Node 3 (step teardown):

```
BEGIN;
SELECT pglogical.alter_subscription_disable('bdr_postgres_bdrgroup_node1_node3');
END;
UPDATE pglogical.subscription set sub_apply_delay = '1s' where sub_name = 'bdr_postgres_bdrgr
SELECT pglogical.alter_subscription_enable('bdr_postgres_bdrgroup_node1_node3');

alter_subscription_disable

t
alter_subscription_enable

t
```

Test two_node_dmlconflict_ut

```
Parsed test spec with 2 sessions

starting permutation: s1t s2u s1w s2w s1s s2s
```

We truncate the table on node1:

Node 1 (step t):

```
TRUNCATE test_dmlconflict;
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'bar';
```

We wait until the TRUNCATE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the UPDATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

```
starting permutation: s2u s1t s2w s1w s1s s2s
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'bar';
```

We truncate the table on node1:

Node 1 (step t):

```
TRUNCATE test_dmlconflict;
```

We wait until the UPDATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE on node1 is replicated to all other nodes:

Node 1 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

Test two_node_dmlconflict_dd

```
Parsed test spec with 2 sessions

starting permutation: s1d s2d s1w s2w s1s s2s
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where  b = 1;
```

We delete same row from node2 concurrently:

Node 2 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the DELETE on node2 is replicated to all other nodes:

Node 2 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

Test two_node_dmlconflict_dt

```
Parsed test spec with 2 sessions

starting permutation: s1d s2t s1w s2w s1s s2s
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We truncate test_dmlconflict on node2:

Node 2 (step t):

```
TRUNCATE test_dmlconflict;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the TRUNCATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c


starting permutation: s2t s1d s1w s2w s1s s2s
```

We truncate test_dmlconflict on node2:

Node 2 (step t):

```
TRUNCATE test_dmlconflict;
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the TRUNCATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

Test two_node_dmlconflict_tt

```
Parsed test spec with 2 sessions

starting permutation: s1t s2t s1w s2w s1s s2s
```

We truncate the table on node1:

Node 1 (step t):

```
TRUNCATE test_dmlconflict;
```

Node 2 (step t):

```
TRUNCATE test_dmlconflict;
```

We wait until TRUNCATE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until TRUNCATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

Test three_node_dmlconflict_iii

```
Parsed test spec with 3 sessions

starting permutation: s1i s2i s3i s1w s2w s3w s1s s2s s3s
```

We insert a row into node1:

Node 1 (step i):

```
INSERT INTO test_dmlconflict VALUES('x', 1, 'foo');
```

We insert a row with the same primary key into node2:

Node 2 (step i):

```
INSERT INTO test_dmlconflict VALUES('y', 1, 'bar');
```

We insert a row with the same primary key into node3:

Node 3 (step i):

```
INSERT INTO test_dmlconflict VALUES('z', 1, 'baz');
```

We wait until INSERT on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until INSERT on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the INSERT on node3 is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

z               1               baz
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

z               1               baz
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

z               1               baz
```

Test three_node_dmlconflict_iiu

```
Parsed test spec with 3 sessions

starting permutation: s1i s2w1 s2s s2u s3i s2w s1w s3w s1s s2s s3s s3teardown
alter_subscription_enable

t
```

We insert a row into a table on node1:

Node 1 (step i):

```
INSERT INTO test_dmlconflict VALUES('x', 1, 'foo');
```

We wait until the INSERT on node1 is replicated to node2:

Node 2 (step w1):

```
SELECT * from pg_sleep(1);
pg_sleep
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

x               1               foo
```

On node2 we update the same row inserted by node1:

Node 2 (step u):

```
UPDATE test_dmlconflict set a='z' where b=1;
```

We insert a row with same primary key on node3 before INSERT from node1‘ is replicated:

Node 3 (step i):

```
INSERT INTO test_dmlconflict VALUES('y', 1, 'baz');
```

We wait until the UPDATE is replicated to all other nodes:

Node 2 (step w):

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

Now we wait until the insert from node1 is replicated to node3 as well:

Node 1 (step w):

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

We wait until the INSERT on node3 is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

y               1               baz
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

y               1               baz
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

y               1               baz
```

Node 3 (step teardown):

```
BEGIN;
SELECT pglogical.alter_subscription_disable('bdr_postgres_bdrgroup_node1_node3');
END;
UPDATE pglogical.subscription set sub_apply_delay = '1s' where sub_name = 'bdr_postgres_bdrgr
SELECT pglogical.alter_subscription_enable('bdr_postgres_bdrgroup_node1_node3');

alter_subscription_disable

t
alter_subscription_enable

t
```

Test three_node_dmlconflict_iid

```
Parsed test spec with 3 sessions

starting permutation: s1i s2w1 s2s s2d s3i s2w s1w s3w s1s s2s s3s s3teardown
alter_subscription_enable

t
```

We insert a row into a table on node1:

Node 1 (step i):

```
INSERT INTO test_dmlconflict VALUES('x', 1, 'foo');
```

We wait until the INSERT on node1 is replicated to node2:

Node 2 (step w1):

```
SELECT * from pg_sleep(1);
pg_sleep
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

x               1               foo
```

On node2 we delete the row replicated from node1:

Node 2 (step d):

```
DELETE from test_dmlconflict where b=1;
```

We insert a row with same primary key on node3 before INSERT from node1 is replicated:

Node 3 (step i):

```
INSERT INTO test_dmlconflict VALUES('y', 1, 'baz');
```

We wait until the DELETE is replicated to all other nodes:

Node 2 (step w):

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

Now we wait until the insert from node1 is replicated to node3 as well:

Node 1 (step w):

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

We wait until the INSERT on node3 is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

y               1               baz
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

y               1               baz
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

x               1               foo
```

Node 3 (step teardown):

```
BEGIN;
SELECT pglogical.alter_subscription_disable('bdr_postgres_bdrgroup_node1_node3');
END;
UPDATE pglogical.subscription set sub_apply_delay = '1s' where sub_name = 'bdr_postgres_bdrgr
SELECT pglogical.alter_subscription_enable('bdr_postgres_bdrgroup_node1_node3');

alter_subscription_disable

t
alter_subscription_enable

t
```

Test three_node_dmlconflict_iit

```
Parsed test spec with 3 sessions

starting permutation: s1i s2w1 s2s s2t s3i s2w s1w s3w s1s s2s s3s s3teardown
alter_subscription_enable

t
```

We insert a row into a table on node1:

Node 1 (step i):

```
INSERT INTO test_dmlconflict VALUES('x', 1, 'foo');
```

We wait until the INSERT on node1 is replicated to node2:

Node 2 (step w1):

```
SELECT * from pg_sleep(1);
pg_sleep
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

z               2               bar
x               1               foo
```

On node2 we truncate the test table after INSERT from node1 is replicated:

Node 2 (step t):

```
TRUNCATE test_dmlconflict;
```

We insert a row with the same primary key on node3 before the INSERT from node1 is replicated:

Node 3 (step i):

```
INSERT INTO test_dmlconflict VALUES('y', 1, 'baz');
```

We wait until the TRUNCATE is replicated to all other nodes:

Node 2 (step w):

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

Now we wait until the insert from node1 is replicated to node3 as well:

Node 1 (step w):

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

We wait until the INSERT on node3 is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

y               1               baz
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

y               1               baz
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c

x               1               foo
```

Node 3 (step teardown):

```
BEGIN;
SELECT pglogical.alter_subscription_disable('bdr_postgres_bdrgroup_node1_node3');
END;
UPDATE pglogical.subscription set sub_apply_delay = '1s' where sub_name = 'bdr_postgres_bdrgr
SELECT pglogical.alter_subscription_enable('bdr_postgres_bdrgroup_node1_node3');

alter_subscription_disable

t
alter_subscription_enable

t
```

Test three_node_dmlconflict_uuu

```
Parsed test spec with 3 sessions

starting permutation: s1u s2u s3u s1w s2w s3w s1s s2s s3s
```

We UPDATE a row from node1:

Node 1 (step u):

```
UPDATE test_dmlconflict SET a = 'x', b = '1', c = 'foo';
```

We UPDATE a row from node2 concurrently:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'bar';
```

We UPDATE a row from node3 concurrently:

Node 3 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'baz';
```

We wait until the UPDATE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the UPDATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the UPDATE on node3 is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c

y              1              baz
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c

y              1              baz
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c

y              1              baz
```

Test three_node_dmlconflict_uud

```
Parsed test spec with 3 sessions

starting permutation: s1u s2w1 s2u s3d s1w s2w s3w s1s s2s s3s s3teardown
alter_subscription_enable

t
```

Node 1 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'baz';
```

We wait until the UPDATE on node1 is replicated to node2:

Node 2 (step w1):

```
SELECT * from pg_sleep(1);
pg_sleep
```

On node2 we update the same row updated by node1:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'z', b = '1', c = 'bar';
```

We DELETE the row on node3 before update from node1 arrives:

Node 3 (step d):

```
DELETE FROM test_dmlconflict;
```

Now we wait until the UPDATE from node1 is replicated to node3 as well:

Node 1 (step w):

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

We wait until the UPDATE is replicated to all other nodes:

Node 2 (step w):

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

We wait until the DELETE on node3 is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

Node 3 (step teardown):

```
BEGIN;
SELECT pglogical.alter_subscription_disable('bdr_postgres_bdrgroup_node1_node3');
END;
UPDATE pglogical.subscription set sub_apply_delay = '1s' where sub_name = 'bdr_postgres_bdrgr
SELECT pglogical.alter_subscription_enable('bdr_postgres_bdrgroup_node1_node3');

alter_subscription_disable

t
alter_subscription_enable

t
```

Test three_node_dmlconflict_uut

```
Parsed test spec with 3 sessions

starting permutation: s1u s2u s3t s1w s2w s3w s1s s2s s3s
```

We update a row on node1:

Node 1 (step u):

```
UPDATE test_dmlconflict SET a = 'z', b = '1', c = 'baz';
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'bar';
```

TRUNCATE the table from node3:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We wait until the UPDATE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the UPDATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE`` onnode3‘ is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c


starting permutation: s1u s3t s2u s1w s2w s3w s1s s2s s3s
```

We update a row on node1:

Node 1 (step u):

```
UPDATE test_dmlconflict SET a = 'z', b = '1', c = 'baz';
```

TRUNCATE the table from node3:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'bar';
```

We wait until the UPDATE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the UPDATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE`` onnode3‘ is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

```
starting permutation: s3t s2u s1u s1w s2w s3w s1s s2s s3s
```

TRUNCATE the table from node3:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'bar';
```

We update a row on node1:

Node 1 (step u):

```
UPDATE test_dmlconflict SET a = 'z', b = '1', c = 'baz';
```

We wait until the UPDATE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the UPDATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE`` onnode3‘ is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

Test three_node_dmlconflict_udt

```
Parsed test spec with 3 sessions

starting permutation: s1d s2u s3t s1w s2w s3w s1s s2s s3s
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = 1, c = 'bar' where b = 1;
```

We truncate the table on node3:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the UPDATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE on node3 is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

```
starting permutation: s2u s1d s3t s1w s2w s3w s1s s2s s3s
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = 1, c = 'bar' where b = 1;
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We truncate the table on node3:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the UPDATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE on node3 is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

```
starting permutation: s2u s3t s1d s1w s2w s3w s1s s2s s3s
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = 1, c = 'bar' where b = 1;
```

We truncate the table on node3:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the UPDATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE on node3 is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

```
starting permutation: s1d s3t s2u s1w s2w s3w s1s s2s s3s
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We truncate the table on node3:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = 1, c = 'bar' where b = 1;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the UPDATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE on node3 is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

```
starting permutation: s3t s2u s1d s1w s2w s3w s1s s2s s3s
```

We truncate the table on node3:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = 1, c = 'bar' where b = 1;
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the UPDATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE on node3 is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c


starting permutation: s3t s1d s2u s1w s2w s3w s1s s2s s3s
```

We truncate the table on node3:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = 1, c = 'bar' where b = 1;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the UPDATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE on node3 is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

Test three_node_dmlconflict_duu

```
Parsed test spec with 3 sessions

starting permutation: s1d s2u s3u s1w s2w s3w s1s s2s s3s
```

We delete the only row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict;
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'bar';
```

We update the same row on node3:

Node 3 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'baz';
```

now we wait until the commit on node1 is updated on all nodes:

Node 1 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

now we wait until the commit on node2 is replicated on all nodes:

Node 2 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

now we wait until the commit on node3 is replicated on all nodes:

Node 3 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

```
starting permutation: s2u s1d s3u s2w s1w s3w s1s s2s s3s
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'bar';
```

We delete the only row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict;
```

We update the same row on node3:

Node 3 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'baz';
```

now we wait until the commit on node2 is replicated on all nodes:

Node 2 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

now we wait until the commit on node1 is updated on all nodes:

Node 1 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

now we wait until the commit on node3 is replicated on all nodes:

Node 3 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

```
starting permutation: s2u s3u s1d s2w s3w s1w s1s s2s s3s
```

We update the same row on node2:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'bar';
```

We update the same row on node3:

Node 3 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'baz';
```

We delete the only row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict;
```

now we wait until the commit on node2 is replicated on all nodes:

Node 2 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

now we wait until the commit on node3 is replicated on all nodes:

Node 3 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

now we wait until the commit on node1 is updated on all nodes:

Node 1 (step w):

```
select bdr.wait_slot_confirm_lsn(null,null);
wait_slot_confirm_lsn
```

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

Test three_node_dmlconflict_ddd

```
Parsed test spec with 3 sessions

starting permutation: s1d s2d s3d s1w s2w s3w s1s s2s s3s
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict;
```

We delete a row from node2 concurrently:

Node 2 (step d):

```
DELETE FROM test_dmlconflict;
```

We delete a row from node3 concurrently:

Node 3 (step d):

```
DELETE FROM test_dmlconflict;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the DELETE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the DELETE on node3 is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

Test three_node_dmlconflict_ddt

```
Parsed test spec with 3 sessions

starting permutation: s1d s2d s3t s1w s2w s3w s1s s2s s3s
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We delete a row from node2:

Node 2 (step d):

```
DELETE FROM test_dmlconflict where b = 2;
```

We truncate test_dmlconclict on node3:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the DELETE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE on node3 is replicated to all other nodes:

Node 3 (step w):

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

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c


starting permutation: s2d s1d s3t s1w s2w s3w s1s s2s s3s
```

We delete a row from node2:

Node 2 (step d):

```
DELETE FROM test_dmlconflict where b = 2;
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We truncate test_dmlconclict on node3:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the DELETE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE on node3 is replicated to all other nodes:

Node 3 (step w):

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

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

```
starting permutation: s2d s3t s1d s1w s2w s3w s1s s2s s3s
```

We delete a row from node2:

Node 2 (step d):

```
DELETE FROM test_dmlconflict where b = 2;
```

We truncate test_dmlconclict on node3:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the DELETE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE on node3 is replicated to all other nodes:

Node 3 (step w):

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

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a               b               c
```

```
starting permutation: s1d s3t s2d s1w s2w s3w s1s s2s s3s
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We truncate test_dmlconclict on node3:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We delete a row from node2:

Node 2 (step d):

```
DELETE FROM test_dmlconflict where b = 2;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the DELETE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE on node3 is replicated to all other nodes:

Node 3 (step w):

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

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

```
starting permutation: s3t s2d s1d s1w s2w s3w s1s s2s s3s
```

We truncate test_dmlconclict on node3:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We delete a row from node2:

Node 2 (step d):

```
DELETE FROM test_dmlconflict where b = 2;
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the DELETE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE on node3 is replicated to all other nodes:

Node 3 (step w):

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

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

```
starting permutation: s3t s1d s2d s1w s2w s3w s1s s2s s3s
```

We truncate test_dmlconclict on node3:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We delete a row from node1:

Node 1 (step d):

```
DELETE FROM test_dmlconflict where b = 1;
```

We delete a row from node2:

Node 2 (step d):

```
DELETE FROM test_dmlconflict where b = 2;
```

We wait until the DELETE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the DELETE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE on node3 is replicated to all other nodes:

Node 3 (step w):

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

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

Test three_node_dmlconflict_tuu

```
Parsed test spec with 3 sessions

starting permutation: s1u s2w1 s2u s3t s1w s2w s3w s1s s2s s3s s3teardown
alter_subscription_enable

t
```

Node 1 (step u):

```
UPDATE test_dmlconflict SET a = 'z', b = '1', c = 'baz';
```

We wait until the UPDATE on node1 is replicated to node2:

Node 2 (step w1):

```
SELECT * from pg_sleep(1);
pg_sleep
```

On node2 we update the same row updated by node1:

Node 2 (step u):

```
UPDATE test_dmlconflict SET a = 'y', b = '1', c = 'bar';
```

We TRUNCATE the table on node3 before update from node1 arrives:

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

Now we wait until the UPDATE from node1 is replicated to node3 as well:

Node 1 (step w):

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

We wait until the UPDATE is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE on node3 is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

Node 3 (step teardown):

```
BEGIN;
SELECT pglogical.alter_subscription_disable('bdr_postgres_bdrgroup_node1_node3');
END;
UPDATE pglogical.subscription set sub_apply_delay = '1s' where sub_name = 'bdr_postgres_bdrgr
SELECT pglogical.alter_subscription_enable('bdr_postgres_bdrgroup_node1_node3');

alter_subscription_disable

t
alter_subscription_enable

t
```

Test three_node_dmlconflict_ttt

```
Parsed test spec with 3 sessions

starting permutation: s1t s2t s3t s1w s2w s3w s1s s2s s3s
```

We truncate the table on node1:

Node 1 (step t):

```
TRUNCATE test_dmlconflict;
```

Node 2 (step t):

```
TRUNCATE test_dmlconflict;
```

Node 3 (step t):

```
TRUNCATE test_dmlconflict;
```

We wait until the TRUNCATE on node1 is replicated to all other nodes:

Node 1 (step w):

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

We wait until the TRUNCATE on node2 is replicated to all other nodes:

Node 2 (step w):

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

We wait until the TRUNCATE on node3 is replicated to all other nodes:

Node 3 (step w):

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

State of node1:

Node 1 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node2:

Node 2 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

State of node3:

Node 3 (step s):

```
SELECT * FROM test_dmlconflict;
a              b              c
```

Appendix C: Known Issues

This section discusses currently known issues in BDR3.

Data Consistency

Please remember to read about Conflicts to understand the implications of the asynchronous operation mode in terms of data consistency.

Concurrent Join and Part

As noted in Creating and Joining a BDR Group, if a new node is being joined concurrently while there is another join or part operation in progress, the new node will sometimes not have consistent data after the join has finished.

List of Issues

In the remaining part of this section we list a number of known issues that are tracked in BDR3's ticketing system, each marked with an unique identifier.

- (RM11693) If the resolver for the `update_origin_change` conflict is set to `skip`, `synchronous_commit=remote_apply` is used, and concurrent updates of the same row are repeatedly applied on two different nodes, then one of the update statements might hang due to a deadlock with the pglogical writer. As mentioned in the Conflicts chapter, `skip` is not the default resolver for the `update_origin_change` conflict, and this combination is not intended to be used in production: it discards one of the two conflicting updates based on the order of arrival on that node, which is likely to cause a divergent cluster.
  In the rare situation that you do choose to use the `skip` conflict resolver, please note the issue with the use of the `remote_apply` mode.
- (RM12052, RM14453) When changing the `apply_delay` value with `alter_node_group_config()` the change does not apply to nodes that are already members of the group. This feature is not intended for use in production and exists to assist with testing BDR. This has been noted in the function documentation.
- (RM14528) An ERROR message "unexpected HTSU_Result after locking" might be logged sporadically due to a unhandled race condition in conflict detection code. The operation that throws this will be retried. The issue occurs rarely, and the cluster will recover automatically from it. This is correctly handled in BDR 3.7.
- (RM16008) A `galloc` sequence might skip some chunks if the sequence is created in a rolled back transaction and then created again with the same name, or if it is created and dropped when DDL

replication is not active and then it is created again when DDL replication is active. The impact of the problem is mild, because the sequence guarantees are not violated; the sequence will only skip some initial chunks. Also, as a workaround the user can specify the starting value for the sequence as an argument to the `bdr.alter_sequence_set_kind()` function.

Appendix D: Libraries

In this section we list the libraries used by BDR3, with the corresponding licenses.

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

LLVM

Copyright © 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 © 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:

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-coreopenssl.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 (eaycryptsoft.com). This product includes software written by Tim Hudson (tjhcryptsoft.com).

Original SSLeay Licence

Copyright © 1995-1998 Eric Young (eaycryptsoft.com) All rights reserved.

This package is an SSL implementation written by Eric Young (eaycryptsoft.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 (tjhcryptsoft.com).

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 (eaycryptsoft.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 (tjhcryptsoft.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.

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.

Appendix E: Table Rewrite Example

This chapter provides a full example of how to change the type of a column for a large, replicated table. While trying to reduce the effect on concurrently running transactions.

As a generic example, we use the table `pgbench_accounts` from a generic pgbench run and demonstrate how to alter the type of the column `bid` from `INT` to `BIGINT` with minimal impact on transactional throughput and fully using BDR replication.

Motivation

The naive way to alter the table would simply be a direct `ALTER TABLE` statement. Given BDR cannot currently replicate this, it could also be applied on each node individually with DDL replication turned off:

```
SELECT bdr.set_ddl_replication('off');
ALTER TABLE pgbench_tellers ALTER COLUMN bid TYPE BIGINT;
SELECT bdr.set_ddl_replication('on');
```

(Note that `bdr.run_on_all_nodes` cannot be used in this case, because `ALTER TABLE...ALTER COLUMN TYPE` cannot run in a transaction block.)

While this may work for smaller tables, it blocks replication and is therefore not a feasible approach for large tables, unless entire BDR cluster is taken offline for maintenance.

Preparation

To perform the bulk of the table rewrite in the background, a temporary column is needed to hold the data of the new type, in our example a `BIGINT` value:

```
ALTER TABLE pgbench_accounts ADD COLUMN bid_new BIGINT;
```

Do not add any constraints just yet. Concurrent transactions may insert new rows or update existing ones. The simplest way to cover inserts is with a `DEFAULT`, for example `now()`, if applicable. However, sometimes a default is not feasible, but data needs to be derived from the old column and assigned to the new one. This can be achieved with a row trigger as follows:

```
-- Trigger function to copy value from the old column to the new one.  This
-- may need to perform further transformation between types.
CREATE FUNCTION pgbench_accounts_copy_bid_to_bid_new()
  RETURNS TRIGGER LANGUAGE plpgsql AS
$$
BEGIN
  NEW.bid_new = NEW.bid;
  RETURN NEW;
END;
```

```
$$;

-- Row trigger for INSERTs and UPDATEs
CREATE TRIGGER pgbench_accounts_insert_update_trigger
  BEFORE INSERT OR UPDATE ON pgbench_accounts
  FOR EACH ROW EXECUTE FUNCTION pgbench_accounts_copy_bid_to_bid_new();
```

None of these operations modify any data in the table, yet. These DDL operations will properly replicate to all BDR nodes and should complete in less than a few seconds.

Actual Table Rewrite

At this point, all rows touched by concurrent transactions should properly populate the new column. To rewrite existing rows, each one needs to be updated. While it's possible to perform a single `UPDATE`, there's again a trade-off between holding locks and therefore blocking other transactions versus overhead and overall time it takes to rewrite the table.

For large tables, it is recommended to update in batches so as to space out the load and minimize locking time and conflicts with concurrent transactions. This can be achieved with a `PROCEDURE` updating the entire table in smaller transactional batches. How exactly rows are batched together is not very relevant and does not necessarily need to be on the primary key, because the downstream replica nodes will use the replica identity to lookup the rows to update, anyway.

Due to the async nature of BDR, the application of `UPDATE`s may be deferred on the peer nodes. Such a storm of updates - no matter how many transactions these are distributed over - may lead to quite a lag, which is not desirable. To spread out the load and reduce the impact on concurrent transactions, we recommend to add waits for peer nodes to catch up.

For our example, we batch by using a modulus of the primary key and wait after every transaction. While this clearly slows down the table rewrite, it has little impact on the throughput of a concurrent `pgbench` run.

```
CREATE PROCEDURE update_pgbench_accounts_for_alter_column()
LANGUAGE plpgsql
AS $$
DECLARE
  -- Batch size in id range, may be adjusted to the table size and
  -- concurrent workload.  Bigger batches may have a larger impact on
  -- latency of concurrent transactions, while smaller batches increase
  -- the overall overhead.
  batch_size CONSTANT INT := 1000;
  aid_min INT;
  aid_max INT;
  i INT;
BEGIN
  SELECT min(aid) INTO aid_min FROM pgbench_accounts;
  SELECT max(aid) INTO aid_max FROM pgbench_accounts;
```

```
  i := aid_min;
  LOOP
    -- Update a reasonably small chunk of the entire table at once.
    UPDATE pgbench_accounts SET bid_new = bid
      WHERE aid BETWEEN i AND (i + batch_size)
        AND bid_new IS NULL;
    COMMIT;

    -- After commit, wait for peer nodes to catch up before
    -- proceeding with the next batch.  This prevents building up a lot of
    -- lag and spreads out the load a bit.
    PERFORM bdr.wait_slot_confirm_lsn(NULL, NULL);
    ROLLBACK;

    RAISE NOTICE 'completed % of % transactions',
      ((i - aid_min) / batch_size),
      ((aid_max - aid_min + batch_size - 1) / batch_size);

    -- Advance to the next batch.
    i := i + batch_size;

    -- Termination condition.
    IF i > aid_max THEN EXIT; END IF;
  END LOOP;
END;
$$;
```

With this procedure in place, the rewriting process can be triggered with:

```
CALL update_pgbench_accounts_for_alter_column();
```

The way the procedure above is written, the process may be interrupted and restarted. Most of the work performed will be retained in subsequent runs. This process may take a long time, as it's runtime is proportional to the number of rows in the table.

Completing the Alteration of the Column Type

After the table is fully rewritten, a few more DDL operations are needed. Note that these will need to acquire global locks. It is therefore *mandatory* to wait for peer nodes to catch up. Therefore, please execute `SELECT bdr.wait_slot_confirm_lsn(NULL, NULL);` at least once after the table rewrite is completed.

If the column had any indexes, these should be re-created on the new column at this point. It is safe to use `CREATE INDEX CONCURRENTLY` individually on each node without DDL replication to reduce lock durations, if the table is not partitioned. For example:

```
SELECT bdr.set_ddl_replication('off');
CREATE INDEX CONCURRENTLY pgbench_accounts_bid_new_idx
    ON pgbench_accounts(bid_new);
SELECT bdr.set_ddl_replication('on');
```

All required constraints, including `NOT NULL` should be duplicated as well now, if required. Using DDL replication by BDR works well for this, again. For example:

```
ALTER TABLE pgbench_accounts ALTER COLUMN bid_new SET NOT NULL;
```

Finally, to swap the new column into place of the old one, drop the trigger and the old column and rename the new one into place. Which should all be done in a single atomic transaction (again replicated by BDR, with a runtime of only few seconds, independent of the size of the table):

```
BEGIN;
DROP TRIGGER pgbench_accounts_insert_update_trigger ON pgbench_accounts;
ALTER TABLE pgbench_accounts DROP COLUMN bid;
ALTER TABLE pgbench_accounts RENAME COLUMN bid_new TO bid;
COMMIT;
```

> **Warning**
> Dropping the column may be prevented by other objects that depend on it, like views, procedures, etc. It's possible to use `CASCADE` to drop the column, but everything that referred it will need to be recreated.

Cleaning up

Indexes and contstraints may need to be renamed to match the original names. The helper functions and procedures created may be dropped again:

```
DROP PROCEDURE update_pgbench_accounts_for_alter_column;
DROP FUNCTION pgbench_accounts_copy_bid_to_bid_new();
```

Appendix F: CAMO Reference Client Implementations

To benefit from using CAMO, the client application program needs to handle and respond to failure cases using the recommended steps. To assist with correct and successful implementation, 2ndQuadrant provides reference implementations in these languages:

- C
- java
- Python

These reference implementations are usable without royalties or licences.

These reference clients are based on a state machine that goes through multiple steps of applying and possibly retrying a given transaction. The application only needs to provide a change application callback, while the provided transaction handler takes care of starting a transaction initially and finally committing, possibly checking with the help of a CAMO partner. Note that in case a retry is required, this callback may be called repeatedly.

Also note that so far, the reference implementations only support a single connection string. Therefore, they are intended to connect to a proxy that arbitrates between two nodes of a symmetric node pair.

Source Code in C

```
#include <unistd.h>

/* Postgres server specific includes */
#include <postgres_fe.h>
#include <libpq-fe.h>

#include "txn_handler.h"
#include <assert.h>

static TransactionStatus isTransactionCommitted(PGconn *conn,
                                    uint32_t node_id, uint32_t xid);
static TransactionStatus tryCommit(PGconn *conn,
                                    uint32_t *node_id, uint32_t *xid);

/* state machine task */
typedef enum TxnApplicationTask
{
    START_TXN,
    SET_COMMIT_SCOPE,
    APPLY_CHANGES,
    QUERY_STATUS,
```

```
    COMMIT_TXN,
    ROLLBACK_TXN,
} TxnApplicationTask;

const char *
txnStatusAsString(TransactionStatus s)
{
    switch (s)
    {
        case CLEAN: return "CLEAN";
        case IN_PROGRESS: return "IN_PROGRESS";
        case COMMITTED: return "COMMITTED";
        case ABORTED: return "ABORTED";
        case UNCERTAIN: return "UNCERTAIN";
        case APPERR_APPLICATION_FAILURE: return "APPERR_APPLICATION_FAILURE";
        case APPERR_COMMIT_FAILURE: return "APPERR_COMMIT_FAILURE";
        case CFGERR_TRANSACTION_ID: return "CFGERR_TRANSACTION_ID";
        case INTERR_STILL_IN_PROGRESS: return "INTERR_STILL_IN_PROGRESS";
        case INTERR_TRANSACTION_STATUS: return "INTERR_TRANSACTION_STATUS";
        case INTERR_NODE_ID: return "INTERR_NODE_ID";
        case INTERR_TRANSACTION_ID: return "INTERR_TRANSACTION_ID";
        default: return "<UNKNOWN STATUS>";
    }
}

static TransactionStatus
isTransactionCommitted(PGconn *conn, uint32_t node_id, uint32_t xid)
{
    PGresult   *res;
    char        query[128];
    const char *status;
    TransactionStatus retval;

    if (PQstatus(conn) == CONNECTION_BAD)
        return UNCERTAIN;

    snprintf(query, sizeof(query) - 1,
             "SELECT bdr.logical_transaction_status(%d, %d);",
             node_id, xid);
    res = PQexec(conn, query);
    if (PQresultStatus(res) == PGRES_TUPLES_OK)
    {
        if (PQntuples(res) == 0)
            return INTERR_TRANSACTION_STATUS;
        else if (PQntuples(res) >= 1)
```

```
        {
            status = PQgetvalue(res, 0, 0);
            if (strcmp(status, "committed") == 0)
                retval = COMMITTED;
            else if (strcmp(status, "aborted") == 0)
                retval = ABORTED;
            else if (strcmp(status, "in progress") == 0)
                /*
                 * This can only happen on the origin, where the original
                 * backend is still waiting for the commit confirmation from
                 * the CAMO partner.  Wait a bit and reconnect, as we prefer
                 * to get an answer from the CAMO partner, instead.
                 *
                 * Note that for the client, the transaction must remain in
                 * the UNCERTAIN state in this case, rather than revert back
                 * to an IN_PROGRESS state.  As the transaction has already
                 * been requested to commit and the CAMO partner may
                 * actually have committed or aborted.  It's just the origin
                 * that doesn't know, yet.
                 */
                retval = UNCERTAIN;
            else
                retval = UNCERTAIN;
        }
        else
            retval = INTERR_TRANSACTION_STATUS;
    }
    else if (PQstatus(conn) == CONNECTION_OK)
    {
        /* No tuples received, but the connection is okay. Abort. */
        retval = INTERR_TRANSACTION_STATUS;
    }
    else
    {
        Assert(PQstatus(conn) == CONNECTION_BAD);
        retval = UNCERTAIN;
    }

    if (res)
        PQclear(res);

    return retval;
}

static TransactionStatus
```

```
tryCommit(PGconn *conn, uint32_t *node_id, uint32_t *xid)
{
    PGresult           *res;
    const char         *node_id_str, *xid_str;
    TransactionStatus   retval;

    node_id_str = PQparameterStatus(conn, "bdr.local_node_id");
    xid_str = PQparameterStatus(conn, "transaction_id");

    if (!xid_str || strcmp(xid_str, "(null)") == 0)
        return CFGERR_TRANSACTION_ID;

    if (!node_id_str)   /* unable to get the local node id */
        return INTERR_NODE_ID;
    *node_id = atoi(node_id_str);
    if (node_id == NULL || *node_id == 0)   /* invalid local node id */
        return INTERR_NODE_ID;

    if (!xid_str)       /* unable to get the current xid */
        return INTERR_TRANSACTION_ID;
    *xid = atoi(xid_str);
    if (*xid < 3)       /* got an invalid xid */
        return INTERR_TRANSACTION_ID;

    res = PQexec(conn, "COMMIT");
    if (PQresultStatus(res) == PGRES_COMMAND_OK)
        retval = COMMITTED;
    else if (PQstatus(conn) == CONNECTION_BAD)
        retval = UNCERTAIN;
    else
        retval = APPERR_COMMIT_FAILURE;

    PQclear(res);
    return retval;
}

void
applyCamoTransaction(PGconn *conn, const char *commit_scope,
                     change_fn change_cb,
                     status_fn status_cb, void *context)
{
    PGresult           *res;
    ChangeResult        cr;
    TransactionStatus   status = CLEAN;
    TxnApplicationTask  task = START_TXN;
```

```
uint32_t            node_id = 0, xid = 0;
char                tmp_sql[256];

while (status != COMMITTED && !isPermanentError(status))
{
    if (task == START_TXN)
    {
        /* Start the transaction */
        res = PQexec(conn, "BEGIN");
        status = IN_PROGRESS;
        if (PQresultStatus(res) == PGRES_COMMAND_OK)
        {
            if (commit_scope)
                task = SET_COMMIT_SCOPE;
            else
                task = APPLY_CHANGES;
        }
        else
            task = ROLLBACK_TXN;
        PQclear(res);
    }
    else if (task == ROLLBACK_TXN)
    {
        /* Rollback and retry on error */
        res = PQexec(conn, "ROLLBACK");
        if (PQresultStatus(res) != PGRES_COMMAND_OK)
            status_cb(context, status, "ROLLBACK failed, ignoring...");
        PQclear(res);
        task = START_TXN;
        status = CLEAN;
    }
    else if (task == SET_COMMIT_SCOPE)
    {
        /* Set the requested commit scope */
        snprintf(tmp_sql, 256, "SET LOCAL bdr.commit_scope = '%s';",
                 commit_scope);
        res = PQexec(conn, tmp_sql);
        if (PQresultStatus(res) == PGRES_COMMAND_OK)
            task = APPLY_CHANGES;
        else
        {
            status_cb(context, status, "SET LOCAL failed, retrying...");
            task = ROLLBACK_TXN;
        }
    }
```

```
else if (task == APPLY_CHANGES)
{
    /* Apply changes by invoking the callback function. */
    cr = change_cb(conn, context);

    /*
     * Depending on the change application status, either commit
     * the changes, retry upon temporary errors or abort in the
     * face of permanent failures.
     */
    if (cr == CR_SUCCESS)
        task = COMMIT_TXN;
    else if (cr == CR_TEMPORARY_FAILURE)
    {
        status_cb(context, status, "temp failure on change application, retrying...")
        task = ROLLBACK_TXN;
    }
    else
    {
        Assert(cr == CR_PERMANENT_FAILURE);
        status = APPERR_APPLICATION_FAILURE;
        status_cb(context, status, "change application failed");
        res = PQexec(conn, "ROLLBACK");
        if (PQresultStatus(res) != PGRES_COMMAND_OK)
            status_cb(context, status, "ROLLBACK failed, ignoring...");
        PQclear(res);
    }
}
else if (task == COMMIT_TXN)
{
    /* Attempt to commit the transaction. */
    assert(status == IN_PROGRESS);
    status = tryCommit(conn, &node_id, &xid);
    if (status == UNCERTAIN)
    {
        /*
         * In case we lost the connection to the server, try to
         * reconnect by resetting it and start over with beginning of
         * a transaction.  Note that there's no need to rollback in
         * this case.
         */
        task = QUERY_STATUS;
        PQreset(conn);
    }
}
```

```
else if (task == QUERY_STATUS)
{
    /*
     * Check status of any previous attempt to apply the transaction,
     * which led to an uncertain result.
     */
    assert(status == UNCERTAIN);
    status = isTransactionCommitted(conn, node_id, xid);
    if (status == COMMITTED)
        status_cb(context, status, "commit of xid %d confirmed", xid);
    else if (status == ABORTED)
    {
        status_cb(context, status,
                  "standby aborted xid %d, retrying...", xid);
        task = START_TXN;   /* transaction got aborted, retry */
        status = CLEAN;     /* reconnected, no need to rollback */
    }
    else if (status == UNCERTAIN)
    {
        /*
         * Still UNCERTAIN.  Attempt to reconnect (possibly
         * switching between nodes) and retry the status
         * query.
         */
        status_cb(context, status,
                  "status query for xid %u failed, reconnecting...",
                  xid);
        sleep(1);
        PQreset(conn);
    }
    else if (isPermanentError(status))
        status_cb(context, status,
                  "permanent error getting status of xid %d", xid);
}
else
    assert(false);

/*
 * Handle loss of connection for multiple states of the state
 * machine, but only before attempting to commit.
 */
if (PQstatus(conn) == CONNECTION_BAD && status == IN_PROGRESS)
{
    PQreset(conn);
    task = START_TXN;
```

```
        }
    }
}
```

Source Code in Java

```
package com.secondquadrant.bdr.camo_client;

import java.math.BigInteger;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

import org.postgresql.PGConnection;
import org.postgresql.util.PSQLException;
import com.secondquadrant.bdr.camo_client.TxnHandler.ITxnChangeApplicator.ChangeResult;

/**
 * Runs the client side CAMO protocol for a given transaction of the
 * application, possibly retrying the transaction on temporary failures.
 */
public class TxnHandler {
    static final int INITIAL_BACK_OFF_DELAY = 100;

    /**
     * Status of a transaction attempted.
     */
    public enum TransactionStatus {
        /** Clean starting state, transaction is known unapplied. */
        CLEAN,

        /**
         * The transaction is in progress and changes are being
         * applied. Nothing committed, yet.
         */
        IN_PROGRESS,

        /** Transaction has successfully committed. */
        COMMITTED,

        /**
         * Transaction is known to have aborted and been rolled
         * back.
```

```
 */
ABORTED,

/**
 * A commit has been attempted, but the client is currently
 * uncertain about the transaction's status. This should only
 * ever happen in case of a server-side node failure,
 * resulting in a bad connection state. The client cannot know
 * whether the server committed the transaction before the
 * connection dropped (or timed out).
 */
UNCERTAIN,

/**
 * The changes attempted to perform raised a permanent error
 * that's not worth retrying. The {link TxnHandler} aborted
 * any attempts to commit this transaction.
 */
APPERR_APPLICATION_FAILURE,

/**
 * The client is well connected, but the server returned a
 * permanent, unrecoverable error for the COMMIT of the
 * transaction. The {link TxnHandler} aborted any attempts to
 * commit this transaction.
 */
APPERR_COMMIT_FAILURE,

/**
 * Failed to get the transactions id.
 */
CFGERR_TRANSACTION_ID,

/**
 * Failed getting the status of the uncertain transaction.
 */
INTERR_TRANSACTION_STATUS,

/**
 * Server passed an invalid transaction id.
 */
INTERR_TRANSACTION_ID,

/**
 * Server passed an invalid node id.
```

```
     */
    INTERR_NODE_ID,
}

/**
 * The interface providing callbacks for the {link TxnHandler},
 * defining the transactional changes to apply.
 */
public interface ITxnChangeApplicator {
    public enum ChangeResult {
        /**
         * Changes have been applied successfully and are ready to
         * be committed.
         */
        CR_SUCCESS,

        /**
         * A transient error occurred during application of
         * changes. The application expects the {link TxnHandler}
         * to rollback the current transaction, but retry it by
         * invoking the {link ITxnChangeApplicator#applyChanges
         * applyChanges method}.
         */
        CR_TEMPORARY_FAILURE,

        /**
         * A permanent failure occurred during application of
         * changes. The application expects the {link TxnHandler}
         * to rollback the current transaction and return the
         * error to the caller of
         *
         * link #applyCamoTransaction(TxnHandler)}.
         */
        CR_PERMANENT_FAILURE
    }

    /**
     * A callback required to perform the actual transaction
     * changes (DML) and only those. Transaction control (begin as
     * well as commit) must be left to the {link TxnHandler}. May
     * be invoked several times, in case retries are required.
     */
    public ChangeResult applyChanges(Connection conn);

    /**
```

```
     * A callback informing the user of the TxnHandler about the
     * current status of the transaction. May be called multiple
     * times per transaction even for good runs. Should only be
     * used for informational and logging purposes.
     */
    public void reportStatus(TransactionStatus status, String msg);
}

/**
 * States of the internal state machine for {link TxnHandler}.
 */
private enum TxnApplicationTask {
    /** Transaction is to be started. */
    START_TXN,
    /** Pending switch to a non-local commit scope. */
    SET_COMMIT_SCOPE,
    /** Transactional changes are about to be applied. */
    APPLY_CHANGES,
    /** Status of the transaction needs to be queried. */
    QUERY_STATUS,
    /** Pending changes need to be committed. */
    COMMIT_TXN,
    /** Transaction failed and needs to be rolled back. */
    ROLLBACK_TXN
}

private String dsn;
private String dbuser;
private String dbpassword;
private Connection conn = null;
private String commit_scope = "local";

private BigInteger xid = BigInteger.ZERO;
private BigInteger nodeId = BigInteger.ZERO;

private int backOffDelay = 0;

public TxnHandler(String dsn, String dbuser, String dbpassword) {
    this.dsn = dsn;
    this.dbuser = dbuser;
    this.dbpassword = dbpassword;
}

public void setCommitScope(String scope) {
    this.commit_scope = scope;
```

```
}

private static boolean isConnectionOkay(Connection conn) {
    try {
        return conn != null && !conn.isClosed();
    } catch (SQLException e) {
        return false;
    }
}

/**
 * A best-effort sleep routine that just returns on interrupts.
 *
 * param msecs maximum sleep time
 */
private static void sleep(int msecs) {
    try {
        Thread.sleep(msecs);
    } catch (InterruptedException ex) {
        Thread.currentThread().interrupt();
    }
}

/**
 * Check if the given error is worth retrying.
 *
 * param ex the exception thrown by some JDBC method
 * return true if the error may be temporary
 */
public static boolean isTemporaryError(PSQLException ex) {
    String sqlstate = ex.getSQLState();
    return sqlstate.startsWith("080")    /* connection exceptions */ ||
        sqlstate.startsWith("53") /* insufficient resources */;
}

/**
 * Attempt to establish a connection. Tries only exactly once and
 * invalidates the {link conn} field upon failure.
 */
private void tryConnect(TransactionStatus status, ITxnChangeApplicator ca) {
    Properties props = new Properties();
    props.setProperty("user", dbuser);
    if (dbpassword != null)
        props.setProperty("password", dbpassword);
    props.setProperty("ssl", "true");
```

```
        props.setProperty("sslfactory",
                          "org.postgresql.ssl.NonValidatingFactory");

        try {
            conn = DriverManager.getConnection(dsn, props);
        } catch (PSQLException ex) {
            if (!isTemporaryError(ex)) {
                ca.reportStatus(status, "Connection failed with: " +
                                ex.getMessage());
            }
            conn = null;
        } catch (SQLException ex) {
            ca.reportStatus(status, "Connection failed with: " +
                            ex.getMessage());
        }
    }

    /**
     * Check the commit status of a transaction attempted to commit.
     *
     * param currentConn the connection to use, defining the node to query
     * param nodeId      identifier of the origin node that processed the
     *                    transaction
     * param xId         identifier of the transaction on the origin node
     * return status of the transaction
     */
    public static TransactionStatus isTransactionCommitted(
        Connection currentConn,
        BigInteger nodeId,
        BigInteger xId,
        boolean was_camo_used
    ) {
        ResultSet res = null;
        String status = null;
        String query = null;
        TransactionStatus retVal = TransactionStatus.COMMITTED;

        assert (TxnHandler.isConnectionOkay(currentConn));

        try {
            currentConn.setAutoCommit(true);
            query = "SELECT bdr.logical_transaction_status(" + nodeId +
                ", " + xId + ", " + (was_camo_used ? "true" : "false") + ");";
            Statement stmt = currentConn.createStatement();
            res = stmt.executeQuery(query);
```

```
    if (!res.isBeforeFirst()) {
        retVal = TransactionStatus.INTERR_TRANSACTION_STATUS;
    } else if (res.next()) {
        status = res.getString(1);

        if (status.toLowerCase().contentEquals("committed"))
            retVal = TransactionStatus.COMMITTED;
        else if (status.toLowerCase().contentEquals("aborted"))
            retVal = TransactionStatus.ABORTED;
        else if (status.toLowerCase().contentEquals("in progress"))
            // This can only happen on the origin, where the
            // original backend is still waiting for the
            // commit confirmation from the CAMO partner. Wait
            // a bit and reconnect, as we prefer to get an
            // answer from the CAMO partner, instead.
            //
            // Note that for the client, the transaction must
            // remain in the UNCERTAIN state in this case,
            // rather than revert back to an IN_PROGRESS
            // state. As the transaction has already been
            // requested to commit and the CAMO partner may
            // actually have committed (or aborted). It's just
            // the origin that doesn't know, yet.
            retVal = TransactionStatus.UNCERTAIN;
        else
            retVal = TransactionStatus.UNCERTAIN;
    } else {
        retVal = TransactionStatus.INTERR_TRANSACTION_STATUS;
    }
} catch (SQLException ex) {
    System.err.println("Status query failed with: " + ex.getMessage());
    ex.printStackTrace();
    retVal = TransactionStatus.UNCERTAIN;
} finally {
    try {
        if (res != null)
            res.close();
    } catch (SQLException ex) {
        System.err.println("Cleanup query result failed with: " +
                           ex.getMessage());
        ex.printStackTrace();
    }
}
return retVal;
```

```
}

/**
 * Attempt to commit the transaction currently open on the given
 * connection.
 *
 * param ca for status reporting
 */
public TransactionStatus tryCommit(ITxnChangeApplicator ca) {
    String nodeIdStr, xidStr = null;
    PGConnection cn = (PGConnection) conn;
    nodeIdStr = cn.getParameterStatus("bdr.local_node_id");
    xidStr = cn.getParameterStatus("transaction_id");

    if (xidStr == null || xidStr.contains("null"))
        return TransactionStatus.CFGERR_TRANSACTION_ID;
    if (nodeIdStr == null)
        return TransactionStatus.INTERR_NODE_ID;

    try {
        assert (nodeIdStr != null);
        nodeId = new BigInteger(nodeIdStr);
    } catch (NumberFormatException ex) {
        return TransactionStatus.INTERR_NODE_ID;
    }

    try {
        assert (xidStr != null);
        xid = new BigInteger(xidStr);
    } catch (NumberFormatException ex) {
        return TransactionStatus.INTERR_TRANSACTION_ID;
    }
    if (xid.intValue() < 3)
        return TransactionStatus.INTERR_TRANSACTION_ID;

    try {
        conn.commit();
    } catch (PSQLException ex) {
        if (TxnHandler.isTemporaryError(ex)) {
            ca.reportStatus(TransactionStatus.UNCERTAIN,
                            "failed to commit xid " + xid + ", retrying");
            return TransactionStatus.UNCERTAIN;
        } else {
            ca.reportStatus(TransactionStatus.APPERR_COMMIT_FAILURE,
                            "failed to commit xid " + xid + ": " + ex);
```

```
            return TransactionStatus.APPERR_COMMIT_FAILURE;
        }
    } catch (SQLException ex) {
        ca.reportStatus(TransactionStatus.APPERR_COMMIT_FAILURE,
                        "failed to commit xid " + xid + ": " + ex);
        return TransactionStatus.APPERR_COMMIT_FAILURE;
    }

    return TransactionStatus.COMMITTED;
}

public boolean isBeforeCommit(TransactionStatus status) {
    return status == TransactionStatus.CLEAN ||
        status == TransactionStatus.IN_PROGRESS;
}

public boolean isPermanentError(TransactionStatus status) {
    return status != TransactionStatus.CLEAN &&
        status != TransactionStatus.IN_PROGRESS &&
        status != TransactionStatus.COMMITTED &&
        status != TransactionStatus.ABORTED &&
        status != TransactionStatus.UNCERTAIN;
}

/**
 * Close the connection and invalidate it. Usually triggers a new
 * connection attempt by the state machine.
 */
private void disconnect() {
    try {
        conn.close();
    } catch (Exception ex) {
        // ignore
    }
    conn = null;
}

private void rollBackAndCleanUp(TransactionStatus status) {
    try {
        conn.rollback();
    } catch (SQLException ex) {
        // ignore all errors
    }
}
```

```
/**
 * Apply a given transaction with CAMO protections, possibly
 * retrying it until it succeeds or yields a permanent error.
 *
 * param ca a change applicator object providing callbacks that
 *           are capable of applying the transactional changes to
 *           be committed.
 */
public void applyCamoTransaction(ITxnChangeApplicator ca) {
    TxnApplicationTask task = TxnApplicationTask.START_TXN;
    TransactionStatus status = TransactionStatus.CLEAN;
    boolean use_camo = commit_scope != "global";

    // Reset backOffDelay before attempting the first time.
    backOffDelay = 0;

    while (status != TransactionStatus.COMMITTED &&
           !isPermanentError(status)) {
        // Attempt to connect if we don't already have a valid
        // connection.
        if (!TxnHandler.isConnectionOkay(conn)) {
            // Delay re-connect and/or retry attempts.
            if (backOffDelay > 0) {
                sleep(backOffDelay);
                backOffDelay *= 1.3;
            }

            tryConnect(status, ca);

            // It's safe to reset to START_TXN if the former
            // connection was interrupted prior to attempting to
            // commit.
            if (isBeforeCommit(status)) {
                task = TxnApplicationTask.START_TXN;
            }
        }

        // If the connection failed, start the back-off delay.
        if (!TxnHandler.isConnectionOkay(conn) && backOffDelay == 0) {
            ca.reportStatus(status,
                "starting back-off delay due to connection failure.");
            backOffDelay = INITIAL_BACK_OFF_DELAY;
        }

        if (conn != null) {
```

```
if (task == TxnApplicationTask.START_TXN) {
    status = TransactionStatus.IN_PROGRESS;
    try {
        // Start a new transaction
        conn.setAutoCommit(false);
        if (commit_scope == "local") {
            task = TxnApplicationTask.APPLY_CHANGES;
        } else {
            task = TxnApplicationTask.SET_COMMIT_SCOPE;
        }
    } catch (SQLException ex) {
        task = TxnApplicationTask.ROLLBACK_TXN;
        ca.reportStatus(status,
            "setAutoCommit failed, retrying...");
    }
} else if (task == TxnApplicationTask.ROLLBACK_TXN) {
    rollBackAndCleanUp(status);
    task = TxnApplicationTask.START_TXN;
    status = TransactionStatus.CLEAN;
} else if (task == TxnApplicationTask.SET_COMMIT_SCOPE) {
    // Set a specific commit scope
    try {
        conn.createStatement().execute(
            "SET LOCAL bdr.commit_scope = '" + commit_scope +
            "';");
        task = TxnApplicationTask.APPLY_CHANGES;
    } catch (SQLException e) {
        ca.reportStatus(status,
            "SET LOCAL failed, retrying...");
        task = TxnApplicationTask.ROLLBACK_TXN;
    }
} else if (task == TxnApplicationTask.APPLY_CHANGES) {
    ChangeResult cr = ChangeResult.CR_SUCCESS;
    cr = ca.applyChanges(conn);

    if (cr == ChangeResult.CR_SUCCESS)
        task = TxnApplicationTask.COMMIT_TXN;
    else if (cr == ChangeResult.CR_TEMPORARY_FAILURE) {
        ca.reportStatus(status,
            "temporary failure on change application, retrying...");
        task = TxnApplicationTask.ROLLBACK_TXN;
        // Start a back-off delay before re-trying.
        if (backOffDelay == 0)
            backOffDelay = INITIAL_BACK_OFF_DELAY;
    } else {
```

```
            assert (cr == ChangeResult.CR_PERMANENT_FAILURE);
            status = TransactionStatus.APPERR_APPLICATION_FAILURE;
            ca.reportStatus(status, "change application failed");
            rollBackAndCleanUp(status);
        }
    } else if (task == TxnApplicationTask.COMMIT_TXN) {
        assert (status == TransactionStatus.IN_PROGRESS);
        status = tryCommit(ca);
        if (status == TransactionStatus.UNCERTAIN) {
            task = TxnApplicationTask.QUERY_STATUS;
        }
    } else if (task == TxnApplicationTask.QUERY_STATUS) {
        // Check status of any previous attempt to apply
        // the transaction, which led to an uncertain
        // result.
        assert (status == TransactionStatus.UNCERTAIN);
        status = isTransactionCommitted(conn, nodeId, xid,
            use_camo);
        if (status == TransactionStatus.COMMITTED) {
            ca.reportStatus(status, "standby committed xid " + xid);
        } else if (status == TransactionStatus.ABORTED) {
            ca.reportStatus(status, "standby aborted xid " + xid +
                            ", retrying...");
            task = TxnApplicationTask.START_TXN;
            status = TransactionStatus.CLEAN;
        } else if (status == TransactionStatus.UNCERTAIN) {
            // Still UNCERTAIN.  Attempt to reconnect
            // (possibly switching between nodes) and
            // retry the status query.
            String msg = String.format(
                "failed to determine status of xid %d, retrying...",
                xid);
            ca.reportStatus(status, msg);

            // Force a reconnection attempt, even if the
            // client is currently connected to the origin
            // node.  This does not guarantee getting a
            // connection to the CAMO partner, but chances
            // are good enough.
            disconnect();

            if (backOffDelay == 0)
                backOffDelay = INITIAL_BACK_OFF_DELAY;
        } else if (isPermanentError(status)) {
            String msg = String.format(
```

```
                            "permanent error getting status of xid %d", xid);
                        ca.reportStatus(status, msg);
                    }

                } else
                    assert (false);
            }
        }
    }
}
```