

EDB Postgres Distributed
Version 3.7

1	EDB Postgres Distributed	4
2	BDR (Bi-Directional Replication)	5
2.1	Release Notes for BDR3	5
2.2	Architectural Overview	30
2.3	Application Usage	34
2.4	PostgreSQL Configuration for BDR	43
2.5	Node Management	49
2.6	DDL Replication	69
2.7	Security and Roles	86
2.8	Conflicts	91
2.9	Sequences	109
2.10	Column-Level Conflict Detection	119
2.11	Conflict-free Replicated Data Types	125
2.12	Durability & Performance Options	134
2.13	Eager Replication	137
2.14	Commit At Most Once (CAMO)	138
2.15	AutoPartition	147
2.16	Timestamp-Based Snapshots	153
2.17	Replication Sets	154
2.18	Stream Triggers	165
2.19	Backup and Recovery	176
2.20	Upgrading	180
2.20.1	Supported BDR upgrade paths	185
2.21	Explicit Two-Phase Commit (2PC)	185
2.22	Catalogs and Views	186
2.23	Monitoring	211
2.24	BDR System Functions	222
2.25	Appendix B: Conflict Details	233
2.26	Appendix B: Known Issues	233
2.27	Appendix C: CAMO Reference Client Implementations	234
3	High Availability Routing for Postgres (HARP)	234
3.1	Release Notes	234
3.1.1	Version 2.4.1	234
3.1.2	Version 2.4.0	235
3.1.3	Version 2.3.2	235
3.1.4	Version 2.3.1	235
3.1.5	Version 2.3.0	235
3.1.6	Version 2.2.3	236
3.1.7	Version 2.2.2	236
3.1.8	Version 2.2.1	236
3.1.9	Version 2.2.0	236
3.1.10	Version 2.1.1	236
3.1.11	Version 2.1.0	237
3.1.12	Version 2.0.3	237
3.1.13	Version 2.0.2	237
3.1.14	Version 2.0.1	238
3.1.15	Version 2.0.0	238
3.2	HARP functionality overview	239

3.3	Installation	244
3.4	Configuring HARP for cluster management	246
3.5	Cluster bootstrapping	255
3.6	HARP Manager	258
3.7	HARP Proxy	260
3.8	harpctl command-line tool	262
3.9	Consensus layer considerations	269
3.10	Security and roles	271
4	pglogical 3.7	273
4.1	Release Notes for pglogical 3.7	273
4.2	Nodes	286
4.3	Replication sets	290
4.4	DDL Replication	298
4.5	Subscription Overview	302
4.5.1	pglogical writer	318
4.6	PostgreSQL settings which affect pglogical	321
4.7	Failover with pglogical3	326
4.8	Restrictions	329
4.9	Error handling in pglogical	331

1 EDB Postgres Distributed

EDB Postgres Distributed provides multi-master replication and data distribution with advanced conflict management, data-loss protection, and [throughput up to 5X faster than native logical replication](#), and enables distributed PostgreSQL clusters with high availability up to five 9s.

By default EDB Postgres Distributed uses asynchronous replication, applying changes on the peer nodes only after the local commit. Additional levels of synchronicity can be configured between different nodes, groups of nodes or all nodes.

EDB Postgres Distributed consists of several components that make the whole cluster work.

Postgres server

Three different Postgres distributions can be used:

- [PostgreSQL](#) - open source
- [EDB Postgres Extended Server](#) - Postgres compatible and optimized for replication (formerly known as 2nd Quadrant Postgres)
- [EDB Postgres Advanced Server](#) - Postgres or Oracle compatible, optimized for replication, and additional enterprise features

What Postgres distribution and version is right for you depends on the features you need. See [Postgres-specific features](#) for detailed information.

BDR extension

A Postgres server with the [BDR](#) extension installed is referred to as a BDR node. BDR nodes can be either data nodes or witness nodes.

Witness nodes don't participate in data replication and are only used as a tie-breaker for consensus.

pglogical3 extension

BDR depends on [pglogical 3.7](#) to provide the replication channel upon which BDR builds.

HARP

[HARP](#) is connection management tool for a EDB Postgres Distributed cluster.

It leverages consensus-driven quorum to determine the correct connection end-point in a semi-exclusive manner to prevent unintended multi-node writes from an application. This reduces the potential for data conflicts.

Compatibility matrix

EDB Postgres Distributed	BDR Extension	pgLogical3 Extension	HARP	Community PostgreSQL	EDB Postgres Extended Server	EDB Postgres Advanced Server	PGD CLI
--------------------------	---------------	----------------------	------	----------------------	------------------------------	------------------------------	---------

EDB Postgres Distributed	BDR Extension	pgLogical3 Extension	HARP	Community PostgreSQL	EDB Postgres Extended Server	EDB Postgres Advanced Server	PGD CLI
4	4	n/a	2	12, 13, 14	12, 13, 14	12, 13, 14	1
3.7	3.7	3.7	2	11, 12, 13	11r2, 12, 13	11, 12, 13	n/a
3.6	3.6	3.6	2	10, 11	11r1	n/a	n/a

2 BDR (Bi-Directional Replication)

BDR (short for Bi-Directional Replication) is a PostgreSQL extension which 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 row-by-row to all the other servers that are part of the same BDR group.

BDR 3 ("BDR3") is built on the [pglogical 3.7](#) extension. However, everything you need to know about BDR3 is included here and it is unnecessary, as well as potentially confusing, to refer to pglogical 3.7 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 supports different versions and distributions of Postgres. See the [Compatibility matrix](#) for more information.

Postgres-specific features

Some features are only available on particular versions of Postgres server.

- Conflict-free Replicated Data Types - additional data types which provide mathematically proven consistency in asynchronous multi-master update scenarios (EDB Postgres Advanced, EDB Postgres Extended)
- 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 (EDB Postgres Advanced, EDB Postgres Extended)
- Transform Triggers - triggers that are executed on the incoming stream of data providing ability to modify it or to do advanced programmatic filtering (EDB Postgres Advanced, EDB Postgres Extended)
- Conflict triggers - triggers which are called when conflict is detected, providing a way to use custom conflict resolution techniques (EDB Postgres Advanced, EDB Postgres Extended)
- Additional DDL support (CREATE TABLE AS) (EDB Postgres Advanced, EDB Postgres Extended)
- Advanced DDL Handling for NOT VALID constraints and ALTER TABLE (EDB Postgres Advanced, EDB Postgres Extended)
- Additional synchronization for Logical/Physical Standby servers for faster build of failoverable standbys (EDB Postgres Advanced, EDB Postgres Extended)
- Parallel Apply - allow multiple writers to apply the incoming changes (EDB Postgres Advanced, EDB Postgres Extended)
- Eager Replication - synchronizes between the nodes of the cluster before committing a transaction to provide conflict free replication (EDB Postgres Extended)
- Commit At Most Once - a consistency feature helping an application to commit each transaction only once, even in the presence of node failures (EDB Postgres Extended)
- Timestamp-based Snapshots - providing consistent reads across multiple nodes for retrieving data as they appeared or will appear at a given time (EDB Postgres Extended)
- Estimates for Replication Catch-up times (EDB Postgres Extended)
- Hold back freezing to assist resolution of UPDATE/DELETE conflicts (EDB Postgres Extended)
- Decoding Worker (EDB Postgres Extended 13 and above)

Features that are currently available only with EDB Postgres Extended are expected to be available with EDB Postgres Advanced 14.

2.1 Release Notes for BDR3

BDR 3.7.23 (2023 Nov 14)

This is a maintenance release for BDR 3.7 that includes minor improvements as well as fixes for issues identified in previous versions.

Also check the release notes for [pglogical 3.7.23](#) for resolved issues that affect BDR as well.

Resolved issues**Improvements**

- Add support for BDR 3.7.22 and above in bdr_pg_upgrade v1.2.0

Upgrades**BDR 3.7.22 (2023 Aug 31)**

This is a maintenance release for BDR 3.7 that includes minor improvements as well as fixes for issues identified in previous versions.

Also check the release notes for [pglogical 3.7.22](#) for resolved issues that affect BDR as well.

Resolved issues

- Changed `bdr.autopartition_drop_partition()` signature to use text.
- Autopartition: Drop partition if it exists It will help in recover from the cases when duplicate drop_partition workitems are created.
- Fixed memory leak in `bdr.sequence_alloc` by modifying the missing catalog signature.
- Prevented superuser check when GUC was specified on PG command line.
- Fixed check for malformed connection string tp prevent failure in `bdr.create_node()` . (RT95453)
- Backport `bdr.accept_connections` GUC.
- Fixed a memory leak in `bdr.sequence_alloc` .
- Remove txn_config entry from ReorderBuffer hash table
- Ignore global_lock check from repset_func when SDW enabled
- Added check for conflicting node names.
- Fixed an issue whereby a crash occurred when BDR extension is used with pgaudit.
- Fixed an issue by allowing a logical join of node if there are foreign key constraints violations. (RT91745)

Improvements

Upgrades

This release supports upgrading from the following versions of BDR:

- 3.7.9 and higher
- 3.6.29 and higher

BDR 3.7.21 (2023 May 16)

This is a maintenance release for BDR 3.7 that includes minor improvements as well as fixes for issues identified in previous versions.

Also check the release notes for [pglogical 3.7.21](#) for resolved issues that affect BDR as well.

Resolved issues

- Fixed memory leak in consensus process (RT91830). The memory consumed by the node is just 32 bytes, but when the consensus worker handles hundreds of requests per second, sustained for hours, the memory builds up. We saw 47% of memory consumed by consensus worker when used with HARP, which executes `bdr.consensus_kv_fetch()` at a rate of 600 times per second.
- Fixed issue where a node can be inconsistent with the group after rejoining. If a node was part of a subgroup, parted, and then rejoined to the group, it might be inconsistent with the group. The changes from some nodes of the group would be replayed from a wrong starting point, resulting in potential data loss.
- Fixed join and replication when SDW and standby_slot_names are set (RT89702, RT89536).
- Fixed upgrades for nodes with CRDTs.
- Fixed replication for subscriber-only node (RT89814).
- Fixed WARNING message in `bdr.raft_leadership_transfer()` (RT92180).
- Fixed segfault where a conflict_slot was being used (RT76439, RT92180) while using `synchronize_structures='none'` during the join. Prevent reuse of the slot after release during multi-insert (COPY).

Improvements

Upgrades

This release supports upgrading from the following versions of BDR:

- 3.7.9 and higher
- 3.6.29 and higher

BDR 3.7.20 (2023 Feb 14)

This is a maintenance release for BDR 3.7 that includes minor improvements as well as fixes for issues identified in previous versions.

Check also release notes for [pglogical 3.7.20](#) for resolved issues that affect BDR as well.

Note

This version is required for EDB Postgres Advanced Server versions 12.14.18, 13.10.14, and later.

Resolved issues

- Fix watermark handling on clusters with multiple sub-groups Watermark is used to ensure data consistency during join. Previously, this didn't work correctly in the presence of multiple data sub-groups.

Improvements

Upgrades

This release supports upgrading from the following versions of BDR:

- 3.7.9 and higher
- 3.6.29 and higher

BDR 3.7.19 (2022 Dec 13)

This is a maintenance release for BDR 3.7 that includes minor improvements as well as fixes for issues identified in previous versions.

Check also release notes for [pglogical 3.7.19](#) for resolved issues that affect BDR as well.

Resolved issues

- Fix timeout issue related to global lock handling (BDR-2836) Correctly lock Raft maintained tables when needed.

Upgrades

This release supports upgrading from the following versions of BDR:

- 3.7.9 and higher
- 3.6.29 and higher

BDR 3.7.18 (2022 Nov 16)

This is a maintenance release for BDR 3.7 that includes minor improvements as well as fixes for issues identified in previous versions.

Check also release notes for [pglogical 3.7.18](#) for resolved issues that affect BDR as well.

Resolved issues

- Don't wait for ADD CONSTRAINT progress if DDL replication is off (BDR-2645, RT86043) The constraint validation from all nodes is not needed if we don't replicate the DDL or from any node that is PARTED or STANDBY.
- Fix raft snapshot read/write routines for sequences (BDR-2666, RT86246) Adjust joining to older BDR 3.6 version nodes while using galloc sequences.
- Fix rare segfault for bdr.drop_node() Check for null values in the result from all the other nodes when trying to drop a node.
- Fix hangs in multiple concurrent joins (RT82977) Various lock corrections for functons and raft requests that reduces the probability of distributed deadlocks.

Upgrades

This release supports upgrading from the following versions of BDR:

- 3.7.9 and higher
- 3.6.29 and higher

BDR 3.7.17 (2022 Aug 23)

This is a maintenance release for BDR 3.7 that includes minor improvements as well as fixes for issues identified in previous versions.

Check also the [release notes for pglogical 3.7.17](#) for resolved issues that affect BDR as well.

Resolved issues

- Fix spurious segmentation faults when conflicts are logged to bdr.conflict_history (BDR-2403, RT83436, RT83928) When conflicts are logged to the catalog bdr.conflict_history, the pglogical writer process may crash because of a segmentation fault due to an invalid pointer being used. Fix this usage.
- Clean up the 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 presence of consensus failure.

Upgrades

This release supports upgrading from the following versions of BDR:

- 3.7.9 and higher
- 3.6.29 and higher

BDR 3.7.16 (2022 May 17)

This is a maintenance release for BDR 3.7 that includes minor improvements as well as fixes for issues identified in previous versions.

Check also release notes for pglogical 3.7.16 for resolved issues that affect BDR as well.

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.
- Show a proper wait event for CAMO / Eager confirmation waits (BDR-1899, RT75900) Show correct "BDR Prepare Phase"/"BDR Commit Phase" in `bdr.stat_activity` instead of the default "unknown wait event".
- Correct `bdr.monitor_local_replslots` for down nodes (BDR-2080) This function mistakenly returned an okay result for down nodes before.
- Reduce log for `bdr.run_on_nodes` (BDR-2153, RT80973) Don't log when setting `bdr.ddl_replication` to off if it's done with the "run_on_nodes" variants of function. This eliminates the flood of logs for monitoring functions.
- Correct an SDW decoder restart edge case (BDR-2109) Internal testing revealed a possible error during WAL decoder recovery about mismatch between `confirmed_flush` LSN of WAL decoder slot also stating: "some LCR segments might be missing". This could happen before in case the WAL decoder exited immediately after processing a "Standby" WAL record other than "RUNNING_XACTS" and would lead to a halt of replication with the decoder processes continuing to restart.

Improvements

- Use 64 bits for calculating lag size in bytes (BDR-2215)

Upgrades

This release supports upgrading from the following versions of BDR:

- 3.7.9 and higher
- 3.6.29 and higher

BDR 3.7.15 (2022 Feb 15)

This is a maintenance release for BDR 3.7 that includes minor improvements as well as fixes for issues identified in previous versions.

Check also release notes for pglogical 3.7.15 for resolved issues that affect BDR as well.

Improvements

- Performance of COPY replication including the initial COPY during join has been greatly improved for partitioned tables (BDR-1479) For large tables this can improve the load times by order of magnitude or more.

- Back-port `bdr.run_on_nodes()` and `bdr.run_on_group()` from BDR 4.0 (BDR-1433) These functions behave same as `bdr.run_on_all_nodes()` but allow running SQL on specific group or set of nodes rather than all nodes.
- Add `execute_locally` option to `bdr.replicate_ddl_command` (RT73533) This allows optional queueing of ddl commands for replication to other groups without executing it locally.
- Don't ERROR on consensus issue during JOIN The reporting of these transient errors was confusing as they are shown in `bdr.worker_errors`. These are now changed to WARNINGS.

Resolved issues

- WAL decoder confirms end LSN of the running transactions record (BDR-1264) Confirm end LSN of the running transactions record processed by WAL decoder so that the WAL decoder slot remains up to date and WAL senders get the candidate in timely manner.
- Improve handling of node name reuse during parallel join (RT74789) Nodes now have a generation number so that it's easier to identify the name reuse even if the node record is received as part of a snapshot.
- Fix locking and snapshot use during node management in the BDR manager process (RT74789) When processing multiple actions in the state machine, we make sure reacquire the lock on the processed node and update the snapshot to make sure any updates happening through consensus are taken into account.
- Improve cleanup of catalogs on local node drop Drop all groups, not only the primary one and drop all the node state history info as well.
- Don't wait for autopartition tasks to complete on parting nodes (BDR-1867) When a node has started parting process, it makes no sense to wait for autopartition tasks on such nodes to finish since it's not part of the group anymore.
- 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.
- Improve error checking for join request in `bdr_init_physical` Previously `bdr_init_physical` would simply wait forever when there was any issue with the consensus request, now we do same checking as the logical join does.
- Improve handling of various timeouts and sleeps in consensus This reduces amount of new consensus votes needed when processing many consensus requests or time consuming consensus requests, for example during join of a new node.

Upgrades

This release supports upgrading from the following versions of BDR:

- 3.7.9 and higher
- 3.6.29 and higher

BDR 3.7.14 (2021 Dec 15)

This is a maintenance release for BDR 3.7 that includes minor improvements as well as fixes for issues identified in previous versions.

Check also release notes for `pglogical 3.7.14` for resolved issues that affect BDR as well.

Improvements

- Reduce frequency of CAMO partner connection attempts (EE) In case of a failure to connect to a CAMO partner to verify its configuration and check the status of transactions, do not retry immediately (leading to a fully busy pglogical manager process), but throttle down repeated attempts to reconnect and checks to once per minute.
- Ensure CAMO configuration is checked again after a reconnect (EE)
- Add dummy CAMO configuration catalogs and Raft support (BDR-1676) This is just to ease rolling upgrades from BDR 3.7 to 4.0.x on CAMO enabled installations.
- Avoid unnecessary LCR segment reads (BDR-1426) We'll now only attempt to read new LCR segments when there are some available. This should reduce I/O load when decoding worker is enabled.

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 gallog sequence in high concurrency situations when the new chunk is used (RT76528) The gallog 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.

Other changes

- Add CAMO configuration infrastructure needed for upgrade to BDR4 (BDR-1676) Add dummy CAMO configuration infrastructure `bdr.camo_pairs` table and `bdr.add/remove_camo_pair()` functions to be able to upgrade a CAMO enabled cluster to BDR4

Upgrades

This release supports upgrading from following versions of BDR:

- 3.7.9 and higher
- 3.6.29 and higher

BDR 3.7.13.1 (2021 Nov 19)

This is a hotfix release for BDR 3.7.13.

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 3.7.13

This is a maintenance release for BDR 3.7 that includes minor improvements as well as fixes for issues identified in previous versions.

Check also release notes for pglogical 3.7.13 for resolved issues that affect BDR as well.

Improvements

- Use a separate replication origin for the BDR consensus process (BDR-1613) For Eager transactions that need to COMMIT PREPARED from the consensus process, use a dedicated replication origin, this way the consensus does not conflict with writer origins.
- 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 to use `drop_node` function instead.
- Add function `bdr.get_decoding_worker_stat()` (BDR-1302) If the Decoding Worker is enabled, this function shows information about the state of the Decoding Worker associated with the current database. This also provides more granular information about Decoding Worker progress than is available via `pg_replication_slots`.

Resolved issues

- Fix a subscriber-side memory leak when bulk-inserting into a partitioned table (BDR-1473) This improves memory usage during node join when there are partitioned tables present.
- 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 memory leak from consensus worker of Raft leader (RT74769) The tracing context was leaked causing growing memory usage from the consensus, on BDR groups with many nodes, this could cause memory exhaustion.
- Enable async conflict resolution for explicit 2PC (BDR-1666, RT71298) Continue applying the transaction using the async conflict resolution for explicit two phase commit.
- Fix potential crash if `bdr.receive_lcr` is "false" (BDR-1620) Adjust Single Decoding Worker feature to automatically disable itself if the `bdr.receive_lcr` is "false". This prevents crash situation when starting replication from a peer in the cluster (on restart, or new join) with `bdr.receive_lcr` disabled and `enable_wal_decoder` enabled.

Other changes

- Add deprecation hint for `bdr.group_max_connections` (BDR-1596) Allow `bdr.group_max_connections` option, but make sure it's properly marked as deprecated in favor of `bdr.raft_group_max_connections`. This GUC will be removed in BDR 4.0.

Upgrades

This release supports upgrading from following versions of BDR:

- 3.7.9 and higher
- 3.6.28

BDR 3.7.12 (2021 Sep 21)

This is a maintenance release for BDR 3.7 that includes minor improvements as well as fixes for issues identified in previous versions.

Check also release notes for pglogical 3.7.12 for resolved issues that affect BDR as well.

Improvements

- Tweak Single Decoding performance by caching and better locking (BDR-1311, BDR-1312) Add caching for BDR-internal catalog information about the Decoding Worker. Split a single global lock into multiple locks (one per WAL sender) for access to internal status information of the WAL sender. This improves performance especially with many concurrent WAL sender processes.
- Add a new view `bdr.replication_status` (BDR-1412) This is similar to the view `pglogical.replication_status` and shows information about the replication status of the local node with respect to all other BDR nodes in the cluster.
- Add function `bdr.wal_sender_stats()` This provides information about whether the WAL sender is using LCRs emitted by a Decoding Worker, and if so the name of the LCR file currently being read from.
- Prevent CAMO to be used in combination with Decoding Worker (BDR-792) These features cannot currently work in combination. This release prevents enabling them both in many cases. This is just a best-effort strategy to prevent mis-configuration.
- Allow to specify a `postgresql.auto.conf` file for `bdr_init_physical` (RT72989, BDR-1400) Add a command line argument to `bdr_init_physical` allowing to provide a custom file to be used for `postgresql.auto.conf`.

Resolved issues

- Fix a potential data loss issue with `bdr_init_physical` (RT71888) When reusing a slot name, previous state was not properly cleaned up in all cases. This has caused potential data loss during physical join as the slot is created ahead of time by `bdr_init_physical` with the same name. The transition from physical to logical replication could miss part of the replication stream, as this drops and recreates the slot. This release properly cleans slot information when dropped and thereby prevents data loss.
- Fix `bdr.camo_local_mode_delay` to really kick in (BDR-1352) 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).
- Prevent segfault in combination with third-party output plugins (BDR-1424, RT72006) Adjust handling of logical WAL messages specific to BDR's Eager All Node Replication mode for output plugins unrelated to BDR. This allows for example Debezium's decoderbufs output plugin to work alongside BDR.
- Improve compatibility with Postgres 13 (BDR-1396) Adjust to an API change in `ReplicationSlotAcquire` that may have led to unintended blocking when non-blocking was requested and vice versa. This version of PGLogical eliminates this potential problem, which has not been observed on production systems so far.
- Fix serialization of Raft snapshots including commit decisions (CAMO, BDR-1454) A possible mismatch in number of tuples could lead to serialization or deserialization errors for a Raft snapshot taken after transactions using CAMO or Eager All Node replication were used recently and stored their commit decisions.
- Fix `--recovery-conf` option in `bdr_init_physical`

Upgrades

This release supports upgrading from following versions of BDR:

- 3.7.9 and higher
- 3.6.27

BDR 3.7.11 (2021 Aug 18)

This is a maintenance release for BDR 3.7 that includes minor improvements as well as fixes for issues identified in previous versions.

Check also release notes for pglogical 3.7.11 for resolved issues that affect BDR as well.

Improvements

- Reduce debug logging of decoding worker (BDR-1236, BDR-1239)
- Allow configuration of maximum connections for consensus (BDR-1005) This allows for setting up very large clusters.

Resolved issues

- Fix snapshot handling in autopartition and executor For compatibility with latest version of PostgreSQL
- Fix deadlock handling in CAMO This solves issue with extremely slow resolution of conflicts in cross-CAMO setup.
- Get copy of slot tuple when logging conflict (BDR-734) Otherwise we could materialize the row early causing wrong update in presence of additional columns on the downstream.
- Improve LCR segment removal logic (BDR-1180, BDR-1183, BDR-993, BDR-1181) Make sure we keep LCR segments for all the LSN that is the smaller between group slot LSN and the decoding worker slot LSN.
- Fix handling of concurrent attach to the internal connection pooler while the pool owner (consensus worker) is restating (BDR-1113)

Upgrades

This release supports upgrading from following versions of BDR:

- 3.7.9 and higher
- 3.6.27

BDR 3.7.10 (2021 Jul 20)

This is a maintenance release for BDR 3.7 that includes minor improvements as well as fixes for issues identified in previous versions.

Improvements

- Check raft quorum in `bdr.monitor_group_raft()` (BDR-960) Return "CRITICAL" status in `bdr.monitor_group_raft()` if at least half of the voting nodes are unreachable.
- Allow `bdr_monitor` role to read additional informational views. (BDR-732)
 - `bdr.group_camo_details`
 - `bdr.group_versions_details`
 - `bdr.group_raft_details`
 - `bdr.group_replslots_details`
 - `bdr.group_subscription_summary`
- Add `is_decoder_slot` to `bdr.node_slots` to differentiate slots used by the Decoder Worker

Resolved issues

- Make the consensus worker always exit if postmaster dies (BDR1063, RT70024)
- Fix starting LSN of Decoding Worker after a restart When the Decoding Worker restarts, it scans the existing LCR segments to find the LSN, transactions upto which, are completely decoded. If this LSN is higher than the slot's confirmed LSN, it updates the slot before decoding any transactions. This avoids transactions being decoded and replicated multiple times. (BDR-876, RT71345)
- Do not synchronize Decoding Worker's replication slot on a physical standby When the WAL decoder starts the first time, the Decoding Worker's slot needs to be behind all the WAL sender slots so that it decodes the WAL required by the WAL senders. But the slot on primary has moved ahead of all WAL senders so synchronizing it is not useful. It is created anew after the physical standby is promoted. (BDR-738)
- Improve join performance when Decoding Worker is enabled When `fsync = on`, joining a new node to a cluster takes much longer with Decoding Worker enabled. Also WAL buildup is observed on the node used as the source of join. This was because the Decoding Worker synced the LCR segments too frequently. Fixed the issue by reducing the frequency. (BDR-1160, RT71345)
- Fix TOAST handling for UPDATE/UPDATE conflicts when Decoding Worker is used
- Fix filtering of additional origins when Decoding Worker is used This mostly affects mixing BDR with Decoding Worker and a separate pglogical replication.
- Eliminate potential hang in `bdr.raft_leadership_transfer` (BDR-1039) In combination with `wait_for_completion`, the best effort approach led to an infinite loop in case the original request was submitted properly, but the actual leadership transfer still failed.
- Do not throw an error when PGL manager can not start a worker (RT71345) If PGL manager throws an error, it is restarted. Since it's responsible for maintaining the node states and other BDR management tasks restarting it on such errors affects the EDB Postgres Distributed cluster's health. Instead log a WARNING.
- Make the reset configuration handling during join more deterministic (RT71021) The `autoadd_tables` option might not be respected in all cases before.
- Deprecate `pub_repsets` and `sub_repsets` in `bdr.node_summary` (BDR-702, RT70743) They now always show `NULL` rather than bogus info, will be removed completely in next major version.
- Show node and group info in `bdr.node_slots` when origin and target node are in different groups.
- Make sure `bdr.monitor_local_replslots()` understands standby nodes and subscriber-only group configuration and does not check for slots that are not needed in these situations (BDR-720)
- Fix internal connection pooler potentially not reusing free connect slots (BDR-1068)
- Fix reported schema name in the missing column error message (BDR-759)

BDR 3.7.9 (2021 Jun 15)

Improvements

- Add `bdr.local_group_slot_name()` function which returns the group slot name (BDR-931) Useful primarily for monitoring.
- Add `bdr.workers` view which show additional information about BDR workers (BDR-725) Helps with monitoring of BDR specific activity. Useful especially when joined with `bdr.stat_activity`.
- Allow Parallel Apply on logical standbys for forwarded transaction (BDR-852) Previously, parallel apply would could be used only for changes replicated directly from the upstream of the logical standby, but not for any changes coming from another node.
- Introduce `bdr.batch_inserts` configuration variable (RT71004, RT70727) This sets after how many `INSERT` s into same table in a row (in same transaction) BDR will switch to multi insert strategy.

This normally improves performance of replication of large data loads, be it via `INSERT` s or the `COPY` command. However BDR 3.7.8 would try to use this strategy always which would result in performance degradation in workloads that do many single row inserts only.

Resolved issues

- Destroy WAL decoder infra on node part/drop (BDR-1107) This ensures that the WAL decoder infra is removed when a node is parted from the cluster. We remove the LCR directory as well as the decoder slot. This allows the node to cleanly join the cluster again later, if need be.
- Do not start WAL decoder on subscriber-only node (BDR-821) The subscriber-only node doesn't send changes to any other nodes in the cluster. So it doesn't require WAL decoder infra and the WAL decoder process itself. Fixing this also ensures that the subscriber-only nodes do not hold back WAL because of an unused slot.
- Start WAL decoder only after reaching PROMOTE state (BDR-1051) We used to create WAL decoder infra when a node starts the join process. That's too early and can lead to WAL accumulation for logical standbys. Instead, we now create the WAL decoder infra only when the node reaches PROMOTE state. That's the state when other nodes may start connecting to the node and hence need WAL decoder.
- Fix group slot advance on subscriber-only nodes (BDR-916, BDR-925, RT71182) This solves excessive WAL log retention on subscriber-only nodes.
- Use correct slot name when joining subscriber-only node using `bdr_init_physical` (BDR-895, BDR-898, RT71124) The `bdr_init_physical` used to create wrong slot, which resulted in 2 slots existing on the join source node when subscriber-only node was joined using this method. This would result in excessive WAL retention on the join source node.
- Fix group monitoring view to allow more than one row per node (BDR-848) Group monitoring views would previously truncate the information from any node reporting more than one row of information. This would result in for example slots missing in `bdr.group_replslots_details`.
- Correct commit cancellation for CAMO (BDR-962) This again corrects CAMO behaviour when a user cancels a query.
- Restore global lock counters state after receiver restart (BDR-958) We already restored locks themselves but not the counters which could cause deadlocks during global locking when using parallel apply.
- Fix handling of `skip_transaction` conflict resolver when there are multiple changes in the transaction after the one that caused the `skip_transaction` (BDR-886)
- Fix Raft snapshot creation for autopartitioned tables (RT71178, BDR-955) Previously the Raft snapshot didn't take into account state of autopartition tasks on all nodes when writing the information. This could result in some nodes skipping partition creation after prolonged period of downtime.
- Adjust transaction and snapshot handling in autopartition (BDR-903) This ensures valid snapshot is used during autopartition processing at all times. The previous approach would cause problem in the future point release of PostgreSQL.

- Fix KSUUID column detection in autopartition
- Fix misreporting of node status by `bdr.drop_node()` function
- Ensure that correct sequence type is always set in the global galloc sequence state.
- Fix DDL replication and locking management of several commands (BDR-874) `ANALYZE`, `CHECKPOINT`, `CLUSTER`, `PREPARE / COMMIT / ABORT TRANSACTION`, `MOVE`, `RELEASE`, `ROLLBACK` were documented as replicated and some of these even tried to take DDL lock which they should not.
- Reduce logging of some unreplicated utility commands (BDR-874) `PREPARE` and `EXECTUE` don't need to spam about not being replicated as nobody expects that they would be.
- Fix global locking of `ALTER TABLE ... SET` (BDR-653) It should not take global DML lock.
- Fix documentation about how `TRUNCATE` command is replicated (BDR-874) While `TRUNCATE` can acquire global locks, it's not replicated the way other DDL commands are, it's replicated like DML, according to replication set settings.
- Document that CAMO and Eager currently don't work with Decoding Worker (BDR-584)
- Multiple typo and grammar fixes in docs.

BDR 3.7.8 (2021 May 18)

This is first stable release of the BDR 3.7. It includes both new major features and fixes for problems identified in 3.7.7.

Important notes

BDR 3.7 introduces several major new features as well as architectural changes some of which affect backward compatibility with existing applications. See [Upgrades](#) for details.

Upgrades are supported from BDR 3.6.25 and 3.7.7 in this release.

The highlights of BDR 3.7

- Support for PostgreSQL 11, 12 and 13
- Support EDB Advanced Server Both Standard Edition and Enterprise Edition are now available to use with EDB Advanced Server
- Parallel Apply Allows configuring number of parallel writers that apply the replication stream. This is feature is supported in Enterprise Edition only.
- AutoPartition Allows automatic management of partitioned tables, with automated creation, automated cleanup with configurable retention periods and more.
- Introduce option to separate BDR WAL decoding worker This allows using single decoding process on each node, regardless of number of subscriptions connected to it. The decoded information is stored in logical change record (LCR) files which are streamed to the other nodes in similar way traditional WAL is. Optional separation of decoding from walsender. This is feature is supported in Enterprise Edition only.
- Implement the concept of `subscriber-only` nodes These are wholly joined nodes, but they don't ever send replication changes to other BDR nodes in the cluster. But they do receive changes from all nodes in the cluster (except, of course the other subscriber-only nodes). They do not participate in the Raft voting protocol, and hence their presence (or absence) does not determine Raft leader election. We don't need to create any

replication slots on these nodes since they don't send replication changes. Similarly, we don't need to create any subscriptions for these nodes on other BDR nodes.

- Support `CREATE TABLE ... AS` and `SELECT INTO` statement This feature is now supported in Enterprise Edition only.
- New ability to define BDR sub-groups in order to better represent physical configuration of the EDB Postgres Distributed cluster. This also simplifies configurations where the EDB Postgres Distributed cluster is spread over multiple data centers and only part of the database is replicated across data centers as each subgroup will automatically have new default replication set assigned to it.
- Multiple new monitoring views Focused primarily on group level monitoring and in-progress monitoring on the apply side.
- Conflicts are now logged by default to `bdr.conflict_history` Logging to a partitioned table with row level security to allow easier access to conflicts for application users.
- New conflict types `multiple_unique_conflicts` and `apply_error_ddl` Allows continuing replication in more edge case situations
- Reduced lock levels for some DDL statements Also, documented workarounds that help with reducing lock levels for multiple other DDL statements.
- Use best available index when applying update and delete This can drastically improve performance for `REPLICA IDENTITY FULL` tables which don't have primary key.

Following are changes since 3.7.7.

Improvements

- Support Parallel Apply in EDB Advanced Server (EE)
- Increase progress reporting frequency when needed (BDR-436, BDR-522) This helps speed up the performance of `VALIDATE CONSTRAINT` without DML locking.
- Change all BDR configuration options that are settable from SQL session to be settable by `bdr_superuser` rather than only Postgres superuser.
- Set `bdr.ddl_replication` to off in `bdr.run_on_all_nodes()` (BDR-445) It's usually not desirable to replicate any DDL executed using the `bdr.run_on_all_nodes()` function as it already runs it on all nodes.
- Improve monitoring of transactions that are in progress on apply side (BDR-690, BDR-691) Add query to `pg_stat_activity` when applying DDL and several additional fields to `bdr.subscription_summary` view which show LSN of latest received change, LSN of latest received commit, applied commit LSN, flushed LSN and applied timestamp.

This helps monitoring of replication progress, especially when it comes to large transactions.

- Add view `bdr.stat_activity`, similar to `pg_stat_activity` but shows BDR specific wait states.
- Allow batching inserts outside of the initial data sync Improves performance of big data loads into existing BDR Group.
- Reduce the global lock level obtained by `DROP INDEX` from DML Global Lock to DDL Global Lock (BDR-652)

Resolved issues

- Fix replication settings of several DDL commands In general make sure that actual behavior and documented behavior for what's allowed, what's replicated and what locks are held during DDL replication match.

For example `TABLESPACE` related commands should not be replicated.

- Fix a race condition in concurrent join. (BDR-644, BDR-645) Always create initially enabled subscription if the local node has already crossed the PROMOTING state.
- Set group leader for already held lock (BDR-418, BDR-291) This solves "canceling statement due to global lock timeout" during some DDL operations when the writer already had open table before. This was especially problem when partitioning or parallel apply is involved.
- Progress WAL sender's slot based on WAL decoder input (BDR-567) Without this, server could eventually stop working with single decoding worker.
- Switch to TEMPORARY replication slots in `bdr_init_physical` (BDR-191) This ensures they are properly cleaned up after `bdr_init_physical` is done.
- Clean up XID progress records that are no longer required (BDR-436, BDR-532) Reduces the size of the xid progress snapshot.
- Track applied_timestamp correctly in BDR Writer (BDR-609) It was not updated in 3.7.7
- Fix creation of BDR Stream triggers on EPAS (BDR-581) They used to be created as wrong trigger type.
- Improve error handling when options stored in LCR file and passed to walsender differ (BDR-551)
- Enable WAL decoder config only for top node group (BDR-566) We only allow group configuration changes for top node group in general.
- Use "C" collation or "name" type for specific BDR catalog columns (BDR-561) This solves potential index collation issues for BDR catalogs.
- Correct commit cancellation for CAMO This fixes CAMO behavior when user cancels a query.
- Fix autopartition handling of tables with already existing partitions (BDR-668)
- Don't cache relation with no remote id in BDRWrite (BDR-620) Fixes replication breakage after some forms of TRUNCATE command.
- Craft upstream decoder slot name considering upstream dbname in wal decoder (BDR-460) Fixes slot names used by wal decoder.
- Use correct BDR output options used by WAL decoder and WAL sender using LCR (BDR-714)
- Fix crash of monitor functions on a broken cluster. (BDR-580, BDR-696)
- Don't show nonexistent slots for PARTED in bdr.node_slots view
- Drop Stream Trigger when dropping node (BDR-692) This enables use of `bdr_init_physical` with Stream Triggers.
- Ensure we don't segfault while handling a SIGUSR2 signal Signals can come at any point in process lifetime so don't make any assumptions about the current state.
- Handle concurrent drop of the table which can lead to missing autopartition rule
- Make sure we don't crash when we get ERROR during handing of different ERROR
- Don't send global xid to client if we are in background worker There is nobody to send this.

Other changes

- Allow session-level `bdr.xact_replication = off` when `bdr.permit_unsafe_commands` is on Helps when using `pg_restore` to manually populate the database.
- Various larger documentaion improvements

- Throw nicer error when removing table from replication set if the table is not in the repset already (BDR-562)
- Allow `check_constraints` option again, but make sure it's properly marked as deprecated (BDR-26) Will be removed in BDR 4.0.
- Move the management of WAL senders when WAL decoder is enabled/disabled to manager process (BDR-612) Managing them in consensus worker could negatively affect responsiveness of consensus subsystem.
- Check for interrupts in more places Should reduce chance of runaway loops

BDR 3.7.7 (2021 Apr 08)

This is a beta release of the BDR 3.7. It includes both new major features and fixes for problems identified in 3.7.6.

Important notes

BDR 3.7 introduces several major new features as well as architectural changes some of which affect backward compatibility with existing applications. See [Upgrades](#) for details.

Beta software is not supported in production - for application test only

Upgrades are supported from BDR 3.6.25 and 3.7.6 in this release.

Improvements

- Support Enterprise Edition features on EDB Advanced Server This notably excludes CAMO and Eager replication.
- Support most of the EDB Advanced Server DDL commands (EBC-45) Note that DDL related to queues is replicated, but the contents of queues are not replicated.
- Adjust DDL replication handling to follow more on command level rather than internal representation (BDR-275) This mainly makes filtering and documentation easier.
- Allow SELECT INTO statement in Enterprise Edition (BDR-306)
- Handle BDR sequences in COPY FROM (BDR-466) COPY FROM does it's own processing of column defaults which does not get caught by query planner hook as it only uses expression planner. Sadly, expression planner has no hook so we need to process the actual COPY FROM command itself.
- Improve `bdr.run_on_all_nodes`(BDR-326, BDR-303) Change return type to jsonb, always return status of each command, Improve error reporting by returning the actual error message received from remote server.
- Add more info to `conflict_history` (BDR-440) This adds couple new fields to the conflict history table for easier identification of tuples without having to look at the actual data.

First one is `origin_node_id` which points to origin of the change which can be different than origin of the subscription because in some situations we forward changes from different original nodes.

Second one is `change_nr` which represents the number of change (based on counter) in the transaction. One change represents one row, not one original command.

These are also added to the conflict history summary table.

Add `local_time` into `bdr.conflict_history_summary` `local_time` is the partition key of `bdr.conflict_history`, which we need to allow monitoring queries to execute efficiently.

- Add `--node-group-name` option to `bdr_init_physical` Same as `node_group_name` in `bdr.join_node_group` - allows joining sub-group of a node.
- Store LCRs under directory named after WAL decoder slot (BDR-60) Pglogical stores LCR in a directory named after the replication slot used to produce those.
- Various improvements in WAL decoder/sender coordination (BDR-232, BDR-335, BDR-342) We now expose the information about WALDecoder waitlsn and let WALSender use that information to wait and signal the WALDecoder when the required WAL is available. This avoids the unnecessary polling and improves coordinator between the two.
- Single Decoder Worker GUC Option Changes. (BDR-222) Changed `bdr.receive_logical_change_records` to `bdr.receive_lcr` and `bdr.logical_change_records_cleanup_interval` to `bdr.lcr_cleanup_interval`
- Move most of the CAMO/Eager code into BDR (BDR-330) Makes CAMO and Eager All Node less dependent on Postgres patches.
- Support the parallelization of initial sync. When parallel apply is enabled, the initial sync during logical join will be parallelized as well.
- Deprecate `bdr.set_ddl_replication` and `bdr.set_ddl_locking`.

Resolved issues

- Fix logic in `bdr_stop_wal_decoder_senders()` (BDR-232) Increase the period for which `bdr_stop_wal_decoder_senders()` should wait before checking status of WAL sender again.
- Disallow running ALTER TABLE..ADD FOREIGN KEY in some cases (EBC-38,BDR-155) If the current user does not have permissions to read the referenced table, disallow the ALTER TABLE ADD FOREIGN KEY to such a table
- Improve detection of queries which mix temporary and permanent objects These need to be disallowed otherwise they could break replication.
- Fix EXPLAIN statement when using INTO TABLE clause.
- Fix `bdr.run_on_all_nodes()` crash on mixed utility commands and DMLs (BDR-305)
- Fix CTAS handling on older minor versions of EPAS
- Consolidate table definition checks (BDR-24) This fixes several hidden bugs where we'd miss the check or creation of extra object
- Fix REINDEX and DROP index on an invalid index (BDR-155, EBC-41) REINDEX throws error if index is invalid. Users can drop invalid indexes using DROP index if_exists.
- Improve checks for local node group membership (BDR-271) Couple of functions, namely `bdr_wait_for_apply_queue` and `bdr_resynchronize_table_from_node` didn't do this check, potentially causing a crash.
- Corrected misleading CTAS ERROR In case of underlying un-supported or non-replicated utility, we should error out and should mention the underlying utility.
- Fixes and improvements around enabling WAL decoder (BDR-272, BDR-427)
- Fix pglogical manager's WAL decoder infrastructure removal (BDR-484)

BDR 3.7.6 (2021 Feb 23)

This is a beta release of the BDR 3.7. It includes both new major features and fixes for problems identified in 3.7.5.

Important notes

BDR 3.7 introduces several major new features as well as architectural changes some of which affect backward compatibility with existing applications. See [Upgrades](#) for details.

Beta software is not supported in production - for application test only

Upgrades are supported from BDR 3.6.25 in this release.

Improvements

- Introduce option to separate BDR WAL decoding worker (RM18868, BDR-51, BDR-58) This allows using single decoding process on each node, regardless of number of subscriptions connected to it. The decoded information is stored in logical change record (LCR) files which are streamed to the other nodes in similar way traditional WAL is.
- Enable parallel apply for CAMO and Eager (RM17858)
- Rework relation caching in BDRWriter This fixes missed invalidations that happened between our cache lookup and table opening. We also reduced the amount of hash table lookups (improving performance).
- Don't allow mixing temporary and permanent object in single DDL command (BDR-93) It's important to not try to replicate DDLs that work with temporary objects as such DDL is sure to break replication.
- Add `bdr.alter_subscription_skip_changes_upto()` (BDR-76) Allows skipping replication changes up to given LSN for a specified subscription. Similar function already exists in `pglogical`.
- Make the snapshot entry handler lookup more robust (BDR-86) This should make it harder to introduce future bugs with consensus snapshot handling.
- Add `bdr.consensus_snapshot_verify()` (BDR-124) Can be used to verify that consensus snapshot provided is correct before passing it to `bdr.consensus_snapshot_import()`.
- Add support for most DDL commands that are specific to EDB Postgres Advanced Server (EBC-39, EBC-40)
- Reduce WARNING spam on non-replicated commands that are not expected to be replicated in the first place (like VACUUM)
- Improve warnings and hints around CAMO configuration

Resolved issues

- Make sure we have xid assigned before opening relation in writer This should improve deadlock detection for parallel apply
- Check table oid in function `drop_trigger` (BDR-35) Fixes crash when invalid oid was passed to the function.
- Fix application of older consensus snapshots (BDR-231) We used to not handle missing group UUID correctly resulting in 3.7 node not being able to join 3.6 cluster.
- Readjust default truncate handling (BDR-25) Don't take lock by default. While this can cause potential out of order truncation, it presents better backwards compatibility.
- Fix crash when OPTION clause is used in CREATE FOREIGN TABLE statement (EBC-37)

- Ensure that we don't send extra data while talking to node with old consensus protocol (BDR-135)
- Read kv_data part of consensus snapshot in mixed version group (BDR-130) Both BDR 3.6. and 3.7 write this part of consensus snapshot but BDR 3.7 would only read it if the snapshot was also written by 3.7.
- Move bdr.constraint to EE script (EBC-36) It's Enterprise Edition only feature so the catalog should only be installed with Enterprise Edition.
- Don't try to replicate GRANT/REVOKE commands on TABLESPACE and Large Objects These objects are not replicated so trying to replicate GRANT and REVOKE would break replication.
- Make sure CAMO does not block replay progress (RT69493)
- Fix failed CAMO connection handling (RT69493, RM19924) Correct the state machine to properly cleanup and recover from this failure and reset to the UNUSED & IDLE state.
- Don't accept Raft request from unknown nodes Consensus leader should not accept raft request from nodes it does not know.
- Don't try to negotiate consensus protocol on unknown node progress (RT69779) When node is forcefully dropped, we might still receive progress message from it. This has to gracefully ignore such message otherwise consensus could break in such situation.

Other changes

- Remove code unsupported consensus protocols (BDR-86)

BDR 3.7.5 (2021 Jan 19)

This is a beta release of the BDR 3.7. It includes both new major features and fixes for problems identified in 3.7.4.

Important notes

BDR 3.7 introduces several major new features as well as architectural changes some of which affect backward compatibility with existing applications. See [Upgrades](#) for details.

Beta software is not supported in production - for application test only

Upgrades are supported from BDR 3.6.22 in this release.

Improvements

- Reduce "now supports consensus protocols" log spam. (RT69557)
- Extend `bdr.drop_node` with a `node_state` check. (RM19280) Adds a new argument 'force' to `bdr.drop_node`, defaulting to false, in which case the following additional check is performed: Via `bdr.run_on_all_nodes`, the current `node_state` of the node to be dropped is queried. If the node to be parted is not fully parted on all nodes, this now yields an error. The force argument allows to ignore this check. This feature also removes the "force" behavior that `cascade` had before, now we have two distinct options, one to skip sanity checks (force) and one to cascade to dependent objects (cascade).
- Deprecate `pg2q.enable_camo` (RM19942, RT69521) The parameter has been changed in 3.7 to the new `bdr.enable_camo`.

- Add new parameter `detector_args` to `bdr.alter_table_conflict_detection` (RT69677) Allow additional parameters for individual detectors. Currently just adds atttype for `row_version` which allows using `smallint` and `bigint`, not just the default integer for the column type.
- Add `bdr.raft_leadership_transfer` (RM20159) Promote a specific node as the Raft leader. Per Raft paper, transferring leadership to a specific node can be done by the following steps:
 - the current leader stops accepting new requests
 - the current leader sends all pending append entries to the designated leader
 - the current leader then forces an election timeout on the designated leader, giving it a better chance to become the next leader

The feature pretty much follows that outline. Instead of sending append entries just to the designated leader, we send it to all nodes as that also acts as a heartbeat. That should ensure that no other node times out while the current leader delegating power to the designated node. We also check status of the designated node and don't accept the request if the node is not an active node or if it doesn't have voting rights.

- Implement the concept of `subscriber-only` nodes These are wholly joined nodes, but they don't ever send replication changes to other BDR nodes in the cluster. But they do receive changes from all nodes in the cluster (except, of course the other subscriber-only nodes). They do not participate in the Raft voting protocol, and hence their presence (or absence) does not determine Raft leader election. We don't need to create any replication slots on these nodes since they don't send replication changes. Similarly, we don't need to create any subscriptions for these nodes on other BDR nodes. We implement this by defining a new type of BDR node group, called "subscriber-only" group. Any node supposed to be a subscriber-only node should join this node group instead of the top level BDR group. Of course, someone needs to create the subscriber-only BDR nodegroup first. The feature does not attempt to create it automatically.
- Improve DDL replication support for PostgreSQL 13 The `ALTER STATISTICS` and `ALTER TYPE ... SET` commands are now supported.

Resolved issues

- Relax the safety check in `bdr.drop_node` . (RT69639) If a node is already dropped on any peer node, that peer does not know the status of the node to drop. It must still be okay to drop that node.
- Do not re-insert a deleted autopartition rule. When an autopartition rule is dropped by one node and while the action is being replicated on some other node, if the other node executes one or more pending tasks for the table, we might accidentally re-insert the rule just being dropped. That leads to problems as where we fail to drop the table on the remote node because the dependency check on autopartition rules fails.
- Fix definition of `node_summary` and `local_node_summary` views (RT69564) While the underlying pglogical catalogs support multiple interfaces per node, BDR will only ever use one, the one that's named same as the node. These views didn't reflect that and shown wrong information - if the node had multiple interfaces the `node_summary` view would show multiple results and the `local_node_summary` would not necessarily pick the correct one from those either.
- Fix `bdr.node_log_config` (RM20318) Adjust the view `bdr.node_log_config` to return correctly the conflict resolution.
- Fix table access statistics reporting inside the writer This should fix PostgreSQL monitoring views that show access and I/O statistics for tables which was broken in previous betas.
- Fix the partitioning of `bdr.conflict_history` after upgrade from 3.6 Previously we'd keep the 3.6 definition, now we do the automatic partitioning same way as fresh 3.7 installs.
- Fix node name reuse for nodes that get initialized from snapshot (RM20111) These nodes previously missed initial state info which could cause catchup phase of join process to be skipped, with the new node missing concurrently written data as a result. This now works correctly.
- Fix potential crash on table rewrite (`VACUUM FULL`) on Standard Edition (EBC-34) Check for triggers on Standard Edition could cause crash on table rewrite previously.
- Don't try to drop Enterprise Edition objects when removing node in Standard Edition (RM19581)
- Improve documentation language

BDR 3.7.4 (2020 Nov 05)

This is a beta release of the BDR 3.7. It includes both new major features and fixes for problems identified in 3.7.3.

Important notes

BDR 3.7 introduces several major new features as well as architectural changes some of which affect backward compatibility with existing applications. See [Upgrades](#) for details.

Beta software is not supported in production - for application test only

Upgrades are supported from BDR 3.6.22 in this release.

Improvements

- Add support for PostgreSQL 13
- 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 below.
- 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.
- Add support for validating constraints without a global DML lock (RM12646) The DDL operation ALTER TABLE ... ADD CONSTRAINT can take quite some time due to the validation to be performed. BDR now allows deferring the validation and running the ALTER TABLE ... VALIDATE CONSTRAINT part without holding the DML lock during the lengthy validation period.

See the section "Adding a CONSTRAINT" in the "DDL Replication" chapter of the documentation for more details.

- ALTER TABLE ... VALIDATE CONSTRAINTS waits for completion Instead of expecting the user to explicitly wait for completion of this DDL operation, BDR now checks progress and waits for completion automatically.
- Add new conflict kind `apply_error_ddl` and resolver `skip_transaction` (RM19351) Can be used to skip transactions where DDL replication would cause `ERROR`. For example when same DDL was applied manually on multiple nodes.
- Add new statistics to `bdr.stat_subscription` (RM18548)
 - `nabort` - how many aborts did writer get
 - how many errors the writer seen (currently same as above)
 - `nskippedtx` - how many txes did the writer skip (using the `skip_transaction` conflict resolver)
 - `nretries` - how many times writer did retry without restart/reconnect
- Improve SystemTAP integration, especially for global locking.

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.

- 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.
- Numerous fixes for gallo sequences (RM18519, RM18512) The "nextval" code for gallo 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.
- 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.
- Fix a problem with NULL values in `bdr.ddl_epoch` catalog (RM19046, RM19072) Release 3.7 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.

- Corrections for upgrading from BDR 3.6.22 Properly migrate subscription writer and conflict handlers from PGLocal, where this information used to be with BDR 3.6. Ensure `bdr.conflict_history` is handled properly after an upgrade.
- 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.
- Validate inputs to `replication_set_remove_table` (RT69248, RM19620)
- Handle missing column gracefully for `ALTER COLUMN TYPE` (RM19389, RT69114) Throw the standard ERROR rather than crashing when this happens.
- Fix memory handling of a tuple slot during conflict lookup (RM18543) No longer crashes when the found tuple is logged into conflict log table.
- Fix local node cache invalidation handling (RM13821) Previously BDR might not notice node creation or node drop due to race conditions, and would chose wrong behavior inside user backend.

BDR 3.7.3 (2020 Aug 06)

This is a beta release of the BDR 3.7. It includes both new major features and fixes for problems identified in 3.7.2.

Important notes

BDR 3.7 introduces several major new features as well as architectural changes some of which affect backward compatibility with existing applications. See [Upgrades](#) for details.

Beta software is not supported in production - for application test only

Upgrade from 3.6 is not supported in this release, yet.

Improvements

- Parallel Apply (RM6503) Using the new infrastructure in pglogical 3.7.3, add support for parallel writers. The defaults are controlled by same pglogical configuration options (and hence this feature is currently off by default) The number of parallel writers can be changed per group using the `num_writers` parameter of the `bdr.alter_node_group_config()` administration interface.
- `resynchronize_table_from_node()` works with the generated columns (RM14876) It copies all the columns except the generated columns from remote node and computes the generated column values locally.
- `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 when the resync happens. This avoids a ton of WAL activity which could potentially happen when hint bit related I/O+WAL would come into the picture in the future on this destination node.
- Allow use of CRDTs on databases with BDR extension installed but without any node (RM17470). Earlier restoring CRDT values on a node with BDR extension, but without any node, would have failed with an ERROR as the CRDT data type queries for the node identifier. It is now fixed by storing an `InvalidOid` value when the node identifier is not available. If the node is subsequently added to a BDR cluster and when the CRDT value is updated, `InvalidOid` will be replaced by a proper node identifier as part of the UPDATE operation.
- Add consistent KV Store implementation for the use by the HARP project (RM17825) This is not meant for direct user consumption, but enables the HARP to work with BDR without additional consensus setup.

Resolved issues

- 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.
- Handle NULL arguments to `bdr.alter_node_set_log_config()` gracefully (RT68375, RM17994) The function caused segmentation fault when the first argument to this function is NULL. It is now fixed to provide an appropriate error message instead.
- Fix MAXVALUE and MINVALUE with galloc sequences (RM14596) While fetching values in advance, we could have reached the limit. Now we use only the values that we fetched before reaching the limit.
- Optionally wait for replication changes triggered by prior epoch (RM17594, RM17802) This improves handling of multiple concurrent DDL operations across the BDR Group which would previously result in global lock timeout, but now are allowed to pass as long as the replication lag between nodes is not too large.
- `resynchronize_table_from_node()` now correctly checks membership of the resynchronized table in replication sets subscribed by the target node (RM17621) This is important in order to not allow unprivileged users to copy tables that they don't have otherwise ability to access.
- Allow new group creation request to work after previous attempt has failed (RM17482) Previously, the new requests would always fail in some setups until BDR was completely removed from the node and reinstalled if the initial group creation has failed.
- Lower the CPU consumption of consensus worker when Autopartition feature is used (RM18002)
- Fix memory leak during initial data synchronization (RM17668)
- Fix `update_recently_deleted` conflict detection (RM16471) This conflict was not detected correctly in 3.7.2.
- Check the options when altering a galloc sequence (RM18301, RT68470) Galloc sequences do not accept some modifications, warn the user in case not allowed options are used.
- Make sure `bdr_wait_slot_confirm_lsn` is waiting for all slots (RM17478) This function used to skip some of the slots when checking if downstream has replicated everything.
- Improve `PART_CATCHUP` node state handling (RM17418) Resolves cases where node state would stay `PART_CATCHUP` forever due to race

condition between nodes.

- Make the consensus process more resilient when there are missing parted nodes Don't fail when trying to update a node's state to `PARTED` and the node no longer exists.
- Remove `--recovery-conf` argument from `bdr_init_physical` (RM17196) It didn't work previously anyway and PostgreSQL12 does not longer have `recovery.conf`.

Other improvements

- Enable `bdr.truncate_locking` by default This is needed for TRUNCATE operations to always produce consistent results when there is concurrent DML happening in the BDR Group. This was missed by previous beta.
- Create a virtual sequence record on other nodes RM16008 If we create a gallop 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.
- Significant improvements to the language in documentation.

BDR 3.7.2 (2020 Jun 01)

This is a beta release of the BDR 3.7.

Important notes

BDR 3.7 introduces several major new features as well as architectural changes some of which affect backward compatibility with existing applications. See [Upgrades](#) for details.

Beta software is not supported in production - for application test only

Upgrade from 3.6 is not supported in this release, yet.

The highlights of BDR 3.7

- Parallel Apply Allows configuring number of parallel writers that apply the replication stream.
- AutoPartition See [AutoPartition](#) for details.
- Support `CREATE TABLE ... AS` statement (RM9696) This feature is now supported in Enterprise Edition only.
- New ability to define BDR sub-groups in order to better represent physical configuration of the EDB Postgres Distributed cluster. This also simplifies configurations where the EDB Postgres Distributed cluster is spread over multiple datacenters and only part of the database is replicated across datacenters as each subgroup will automatically have new default replication set assigned to it.
- Conflicts are now logged by default to `bdr.conflict_history` Logging to a partitioned table with row level security to allow easier access to conflicts for application users.
- New conflict type `multiple_unique_conflicts` Allows resolution of complex conflicts involving multiple UNIQUE constraints for both INSERT and UPDATE.

- Merge views `bdr.node_replication_rates` and `bdr.node_estimate` into `bdr.node_replication_rates`. `bdr.node_estimate` has been removed (RM13523)
- Don't replicate REINDEX command, now treated as a maintenance command
- Various other changes to default settings

Other improvements

- Optional monitoring tables for describing node connections and geographical distribution
- Add `bdr.resynchronize_table_from_node` function (RM13565, RM14875) This function resynchronizes the relation from a remote node. 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.
- Add a function `bdr.trigger_get_origin_node_id` to be used in conflict triggers (RM15105, RT67601) This will enable users to define their conflict triggers such that a trusted node will always win in case of DML conflicts.
- Extend `bdr.wait_for_apply_queue` to wait for a specific LSN (RM11059, RT65827)
- Add committed LSN reporting via `bdr.last_committed_lsn` (RM11059, RT65827)
- BDR now accepts also URI in connection strings (RM14588) We can now specify also the format URI "postgresql://..." for the connection string.

Resolved issues

- 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.
- Correct parsing of BDR WAL messages (RT67662) In rare cases a DDL which is replicated across a EDB Postgres Distributed 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.

2.2 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.



By default BDR uses asynchronous replication, applying changes on the peer nodes only after the local commit. An optional [eager all node replication](#) feature allows for committing on all nodes using consensus.

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 don't have to write to all the masters, all of the time; it's 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 EDB Postgres Distributed 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 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.

Replicated data is sent in binary form, when it is safe to do so.

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 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, EDB Postgres Distributed 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 can run hundreds of nodes on good enough hardware and network, however for mesh based deployments it's generally not recommended to run more than 32 nodes in one cluster. 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. Standby nodes don't add additional connections to the mesh network so they are not included in the 32 node recommendation.

BDR places a limit that at most 10 databases in any one PostgreSQL instance can be BDR nodes across different BDR node groups. However BDR works best if only one BDR database per PostgreSQL instance is used.

The minimum recommended number of nodes in EDB Postgres Distributed cluster is 3, because with 2 nodes the consensus stops working if one of the node stops working.

Architectural Options & Performance

Characterising BDR performance

BDR can be configured in a number of different architectures, each of which has different performance and scalability characteristics.

The Group is the basic building block of a BDR Group consisting of 2+ nodes (servers). Within a Group, each node is in a different AZ, with dedicated router and backup, giving Immediate Switchover and High Availability. Each Group has a dedicated Replication Set defined on it. If the Group loses a node it is easily repaired/replaced by copying an existing node from the Group.

Adding more master nodes to a BDR Group does not result in significant write throughput increase when most tables are replicated because BDR has to replay all the writes on all nodes. Because BDR writes are in general more effective than writes coming from Postgres clients via SQL, some performance increase can be achieved. Read throughput generally scales linearly with the number of nodes.

The following architectures are available:

- Multimaster/Single Group
- BDR AlwaysOn

The simplest architecture is just to have one Group, so let's examine that first:

BDR MultiMaster within one Group

By default, BDR will keep one copy of each table on each node in the Group and any changes will be propagated to all nodes in the Group.

Since copies of data are everywhere, SELECTs need only ever access the local node. On a read-only cluster, performance on any one node will not be affected by the number of nodes. Thus adding nodes will increase linearly the total possible SELECT throughput.

INSERTs, UPDATEs and DELETEs (DML) are performed locally, then the changes will be propagated to all nodes in the Group. The overhead of DML apply is less than the original execution, so if you run a pure write workload on multiple nodes concurrently, a multi-node cluster will be able to handle more TPS than a single node.

Conflict handling has a cost that will act to reduce the throughput. The throughput is then dependent upon how much contention the application displays in practice. Applications with very low contention will perform better than a single node; applications with high contention could perform worse than a single node. These results are consistent with any multi-master technology, they are not a facet or peculiarity of BDR.

Eager replication can avoid conflicts, but is inherently more expensive.

Changes are sent concurrently to all nodes so that the replication lag is minimised. Adding more nodes means using more CPU for replication, so peak TPS will reduce slightly as each new node is added.

If the workload tries to uses all CPU resources then this will resource constrain replication, which could then affect the replication lag.

BDR AlwaysOn

The AlwaysOn architecture is built from 2 Groups, in 2 separate regions. Each Group provides HA and IS, but together they also provide Disaster Recovery (DR), so we refer to this architecture as AlwaysOn with Very High Availability.

Tables are created across both Groups, so any change goes to all nodes, not just to nodes in the local Group.

One node is the target for the main application. All other nodes are described as shadow nodes (or "read-write replica"), waiting to take over when needed. If a node loses contact we switch immediately to a shadow node to continue processing. If a Group fails, we can switch to the other Group. Scalability is not the goal of this architecture.

Since we write mainly to only one node, the possibility of contention between is reduced to almost zero and as a result performance impact is much reduced.

CAMO is eager replication within the local Group, lazy with regard to other Groups.

Secondary applications may execute against the shadow nodes, though these should be reduced or interrupted if the main application begins using that node.

Future feature: One node is elected as main replicator to other Groups, limiting CPU overhead of replication as the cluster grows and minimizing the bandwidth to other Groups.

Deployment

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

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](#).

2.3 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 supports replicating changes made on one node to other nodes.

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 an explicit REPLICIA IDENTITY defined on specific column(s). If one of those is not defined, a WARNING will be generated, and later UPDATEs or DELETEs will be explicitly blocked. If REPLICIA IDENTITY FULL is defined for a table, then a unique index is not

required; in that case, UPDATEs and DELETEs are allowed and will use the first non-unique index that is live, valid, not deferred and does not have expressions or WHERE clauses, otherwise a sequential scan will be used.

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.

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. Information stored by transactions running in SERIALIZABLE mode is not replicated to other nodes; the transaction isolation level of SERIALIZABLE is supported but transactions will not be serialized across nodes, in the presence of concurrent transactions on multiple nodes.

If DML is executed on multiple nodes concurrently then potential conflicts could occur if executing with asynchronous replication and these must be either handled or avoided. Various avoidance mechanisms are possible, discussed in the chapter on [Conflicts](#) which is also required reading.

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. Use of the PostgreSQL "Large object" facility is not supported in BDR.

Rules execute only on the origin node, so are not executed during apply, even if they are enabled for replicas.

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. DML changes that are made through updatable views are resolved through to base tables on the origin and then applied to the same base table name on the target.

BDR supports partitioned tables transparently, meaning that a partitioned table can be added to a replication set and changes that involve any of the partitions will be replicated downstream.

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.

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, REINDEX)
- 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.

DML and DDL Replication

Note that BDR does not replicate the DML statement, it replicates the changes caused by the DML statement. So for example, an UPDATE that changed two rows would replicate two changes, whereas a DELETE that did not remove any rows would not replicate anything. This means that the results of execution of volatile statements are replicated, ensuring there is no divergence between nodes as might occur with statement-based replication.

DDL replication works differently to DML. For DDL, BDR replicates the statement, which is then executed on all nodes. So a DROP TABLE IF EXISTS might not replicate anything on the local node, but the statement is still sent to other nodes for execution if DDL replication is enabled. Full details are covered in their own chapter: [DDL replication].

BDR goes to great lengths to ensure that intermixed DML and DDL statements work correctly, even within the same transaction.

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.

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, all columns are replicated. BDR replicates data columns based on the column name. 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.

BDR supports replicating between tables that have a 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.

Comparison between nodes with differences

Livecompare is a tool used for data comparison on a database, against BDR and non-BDR nodes. It needs a minimum number of two connections to compare against and reach a final result.

From Livecompare 1.3, you could configure with `all_bdr_nodes` set. This will save you from clarifying all the relevant DSNs for each separate node in the cluster. A EDB Postgres Distributed cluster has N amount of nodes with connection information, but its only the initial and output connection that livecompare 1.3+ needs in order to complete its job. Setting `logical_replication_mode` will state how all the nodes are communicating.

All the configuration is done within a .ini file, named `bdrLC.ini` for example. Templates for this configuration file can be seen within the `/etc/2ndq-livecompare/` location, where they were placed after the package install.

During the execution of LiveCompare, you will see N+1 progress bars, N being the number of processes. Once all the tables are sourced a time will display, as the transactions per second (tps) has been measured. This will continue to count the time, giving you an estimate, then a total execution time at the end.

This tool has a lot of customisation and filters. Such as tables, schemas and replication_sets. LiveCompare can use stop-start without losing context information, so it can be run at convenient times. After the comparison, a summary and a DML script are generated so the user can review it. Please apply the DML to fix the found differences, if any.

General Rules for Applications

As discussed above, BDR uses replica identity values to identify the rows to be changed. Applications can cause difficulties if they insert, delete, and then later re-use the same unique identifiers. This is known as the ABA Problem. BDR cannot know whether the rows are the current row, the last row, or much older rows. See https://en.wikipedia.org/wiki/ABA_problem.

Similarly, since BDR uses table names to identify the table against which changes will be replayed, a similar ABA problem exists with applications that CREATE, then DROP, and then later re-use the same object names.

These issues give rise to some simple rules for applications to follow:

1. Use unique identifiers for rows (INSERT)
2. Avoid modification of unique identifiers (UPDATE)
3. Avoid reuse of deleted unique identifiers
4. Avoid reuse of dropped object names

In the general case, breaking those rules can lead to data anomalies and divergence. Applications can break those rules as long as certain conditions are met, but use caution: although anomalies can be unlikely, they are not impossible. For example, a row value can be reused as long as the DELETE has been replayed on all nodes, including down nodes. This might normally occur in less than a second, but could potentially take days if a severe issue occurred on one node that prevented it from restarting correctly.

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.

The synchronous replication features of Postgres are available to BDR as well. 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 EDB to deploy reference TPA architectures, including those based on EDB Postgres Distributed.

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 `<name>` 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 that 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=myhost1 dbname=mydb port=5432'`

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 'd2=dbname=db2'
```

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 CRDT (Conflict-free Replicated Data Types) and CLCD (Column-Level Conflict Detection) 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 you're having performance problems, you're 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 running on EDB Postgres Extended, 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 EDB Postgres Distributed 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 parameters are 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 running on EDB Postgres Extended, 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
```

2.4 PostgreSQL Configuration for BDR

There are several PostgreSQL configuration parameters that affect BDR nodes. Note that these parameters could be set differently on each node, though that is not recommended, in general.

PostgreSQL Settings for BDR

BDR requires these PostgreSQL settings to run correctly:

- `wal_level` - Must be set to `logical`, since BDR relies upon logical decoding.
- `shared_preload_libraries` - This must contain `pglogical,bdr` (in that order), though may also contain other entries before or afterwards, as needed.
- `track_commit_timestamp` - Must be set to 'on' for conflict resolution to retrieve the timestamp for each conflicting row.

BDR requires these PostgreSQL settings to be set to appropriate values, which vary according to the size and scale of the cluster.

- `logical_decoding_work_mem` - memory buffer size used by logical decoding. Transactions larger than this will overflow the buffer and be stored temporarily on local disk. Default 64MB, but can be set much higher.
- `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 + four per BDR-enabled database + one per peer node in the BDR group + one for each writer enabled per peer node in the BDR group, for each database. Additional worker processes may be needed temporarily when node is being removed from a BDR group.
- `max_wal_senders` - Two needed per every peer node.
- `max_replication_slots` - Same as `max_wal_senders`.

- `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.

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

`max_replication_slots` sets the maximum number of replication origins. Some of the functionality of parallel apply uses an extra origin per writer. Hence, when parallel apply is enabled, you must set the `max_replication_slots` to `N * (number of writers)` plus the number of slots needed for peak demand as described in the previous paragraph.

When the [Decoding Worker](#) is enabled, this process will require one extra replication slot per BDR group.

The general safe recommended value on a 4 node BDR Group with a single database is just to set `max_replication_slots` and `max_worker_processes` to something like 50 and `max_wal_senders` to at least 10.

Note also that changing these parameters requires restarting the local node: `max_worker_processes`, `max_wal_senders`, `max_replication_slots`.

Applications may also wish to set these parameters. Please see chapter on [Durability & Performance Options] for further discussion.

- `synchronous_commit` - affects the durability and performance of BDR replication in a similar way to [physical replication](#).
- `synchronous_standby_names` - same as above

2ndQPostgres/EDB Postgres Extended 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.

- `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.
- `pglogical.writers_per_subscription` - Default number of writers per subscription (in BDR this can also be changed by `bdr.alter_node_group_config` for a group).
- `pglogical.max_writers_per_subscription` - Maximum number of writers per subscription (sets upper limit for the setting above).

`pglogical.min_worker_backoff_delay` and `pglogical.max_worker_backoff_delay`

Rate limit BDR background worker launches by preventing a given worker from being relaunched more often than every `pglogical.min_worker_backoff_delay` milliseconds. On repeated errors, the back-off increases exponentially with added jitter up to maximum

of `pglogical.max_worker_backoff_delay`.

Time-unit suffixes are supported.

Note

This setting currently only affects receiver worker, which means it primarily affects how fast a subscription will try to reconnect on error or connection failure.

The default for `pglogical.min_worker_backoff_delay` is 1 second, for `pglogical.max_worker_backoff_delay` it is 1 minute.

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

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

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

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

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 `bdr_superuser` or `superuser` roles.

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.

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` - False by default, this configuration option sets the TRUNCATE command's locking behavior. Determines whether (when true) TRUNCATE obeys the `bdr.ddl_locking` setting.

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, which means the transaction will not be visible to logical decoding by BDR and all other downstream targets of logical decoding. Data will not be transferred to any other node, including logical standby nodes.

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 unless `bdr.permit_unsafe_commands = on`.

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.batch_inserts` - How many consecutive inserts to one table within a single transaction turns on batch processing of inserts for that table.

This option allows replication of large data loads as COPY internally, rather than set of inserts. It also how the initial data during node join is copied.

- `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. (EDB Postgres Extended)

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.

Note

This is only available on EDB Postgres Extended.

Commit at Most Once

- `bdr.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.
- `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.

Note

This is only available on EDB Postgres Extended.

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. (EDB Postgres Extended)

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 EDB Postgres Distributed cluster may help EDB Support to diagnose issues. May only be set at Postgres server start.

Warning

Setting `bdr.debug_level` or `bdr.trace_level` to a value \geq `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.group_max_connections` - The maximum number of connections across all BDR groups for a Postgres server. These connections carry bdr consensus requests between the groups' nodes. Default value of this parameter is 100 connections. May only be set at Postgres server start. BDR 3.7.13 and later will emit a warning when setting this variable that it is deprecated and `bdr.raft_group_max_connections` should be used. Ignore this warning and continue to use `bdr.group_max_connections` (until you upgrade to BDR 4 and later).
- `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. (EDB Postgres Extended)
- `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. (EDB Postgres Extended)

2.5 Node Management

Each database that is member of a BDR group must be represented by its own node. A node is a 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.

A BDR Group may also contain zero or more sub-groups, allowing a variety of different architectures to be created.

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 of only one node at that point). At the time of creation, you must specify the connection string that other nodes will use to connect to this node.

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

Alternatively, use the command line utility `bdr_init_physical` 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 behaviour 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_create_subscriber` utility.

When a new BDR node is joined to an existing BDR group or a node is subscribed to an upstream peer, before replication can begin, the system must copy the existing data from the peer node(s) to the local node. This copy must be carefully coordinated so that the local and remote data starts out *identical*; it is 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 that is joining the cluster must not contain any schema or data that already exists on databases in the BDR group. We recommend that the newly joining database is empty except for the BDR and `pglogical` extension. Ensure that all required database users and roles are created. However, the schema synchronization can be optionally skipped using `synchronize_structure` parameter of `bdr.join_node_group()` function in which case the schema must exist on the newly joining node already.

We recommend that the source node which has the best connection (i.e. is closest) is selected 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 most existing nodes to be online and reachable.

The logical join procedure (which uses `bdr.join_node_group()` function) performs data sync doing `COPY` operations and will use multiple writers (parallel apply) if those are enabled.

Node join can execute concurrently with other node joins for the majority of the time taken to join. However, only one regular node at a time can be in either of the states `PROMOTE` or `PROMOTING`, which are typically fairly short if all other nodes are up and running, otherwise the join will get serialized at this stage. The subscriber-only nodes are an exception to this rule, and they can be concurrently in `PROMOTE` and `PROMOTING` states as well, so their join process is fully concurrent.

Note that the join process uses only one node as the source, so can be executed when nodes are down, if 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, run `LiveCompare` on the newly joined node to correct any data divergence once all nodes are available and caught up.

`pg_dump` may fail when there is concurrent DDL activity on the source node because of cache lookup failures. Since `bdr.join_node_group()` uses `pg_dump` internally, it may fail if there is concurrent DDL activity on the source node. Retrying the join should work in such a case.

Joining a Heterogeneous Cluster

BDR 3.7 node can join a EDB Postgres Distributed cluster running 3.6.x at a specific minimum maintenance release (e.g. 3.6.25) or a mix of 3.6 and 3.7 nodes. This procedure is useful when user wants to upgrade not just the BDR major version but also the underlying PostgreSQL (or 2ndQPostgres) major version. This can be achieved by joining a 3.7 node running on PostgreSQL (or 2ndQPostgres) 12 or 13 to a EDB Postgres Distributed cluster running 3.6.x on PostgreSQL (or 2ndQPostgres) 11. Of course, the new node can also be running on the same PostgreSQL major release as all of the nodes in the existing cluster.

BDR ensures that the replication works correctly in all directions even when some nodes are running 3.6 on one PostgreSQL major release and other nodes are running 3.7 on another PostgreSQL major release. But it's recommended that the user quickly bring the cluster into a homogenous state by parting the older nodes once enough new nodes has joined the cluster. Care must be taken to not run any DDLs that might not be available on the older versions and vice versa.

A node joining with a different major PostgreSQL release cannot use physical backup taken via `bdr_init_physical` and the node must join using the logical join method. This is necessary because the major PostgreSQL releases are not on-disk compatible with each other.

Note that when a 3.7 node joins the cluster using a 3.6 node as a source, certain configuration such as conflict resolution configurations are not copied over from the source node. The node must be configured after it has joined the cluster.

Connection DSNs and SSL (TLS)

The DSN of a node is simply 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 shown here:

```
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 that 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 to 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 to 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 16 MB of LSN has elapsed since the group slot was last advanced. In extreme cases, this may require a full 16 MB before slots are synced/created on the streaming replica. If 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.

On EDB Postgres Extended and EDB Postgres Advanced, this is resolved automatically. 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 EDB Postgres Distributed 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 slot's sync-up, allowing for faster failover to it, if required.

On PostgreSQL, it is important to ensure that slot's sync up has completed on the standby before promoting it. The following query can be run on the standby in the target database to monitor and ensure that the slots have synced up with the upstream. The promotion can go ahead when this query returns `true`.

```
SELECT true FROM pg_catalog.pg_replication_slots
WHERE
    slot_type = 'logical' AND confirmed_flush_lsn IS NOT
NULL;
```

It is also possible to nudge the slot sync-up process in the entire BDR cluster by manually performing WAL switches and by requesting all BDR peer nodes to replay their progress updates. This activity will cause the group slot to move ahead in a short timespan, and also hasten the slot sync-up activity on the standby. The following queries can be run on any BDR peer node in the target database for this:

```
SELECT bdr.run_on_all_nodes('SELECT
pg_catalog.pg_switch_wal()');
```

```
SELECT bdr.run_on_all_nodes('SELECT
bdr.request_replay_progress_update()');
```

Use the monitoring query from above on the standby to check that these queries indeed help in faster slot sync-up on that standby.

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 it as a standby, or ensure that divergent nodes are never promoted.

Physical Standby Nodes

BDR also enables the creation of traditional physical standby failover nodes. 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 the 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` (for PostgreSQL <12, for PostgreSQL 12+ these settings should be in `postgres.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 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.

The protection mechanism that ensures physical standby nodes are up to date and can be promoted (as configured `pglogical.standby_slot_names`) affects the overall replication latency of the BDR Group as the group replication only happens once the physical standby nodes are up to date.

For these reasons it's generally recommended to use either logical standby nodes or subscribe-only group instead of physical standby nodes because they

both have better operational characteristics in comparison.

When bdr-enterprise extension is installed, you can manually ensure the group slot is advanced on all nodes (as much as possible), which helps hasten the creation of BDR-related replication slots on a physical standby using the following SQL syntax:

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

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 EDB Postgres Distributed 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 EDB Postgres Distributed 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 different.

BDR maintains a group slot that always reflects the state of the cluster node showing the most lag for any outbound replication. 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`. CAMO and Eager All Node Replication use different synchronization mechanisms and do not work with synchronous replication. Please make sure `synchronous_standby_names` does not include the CAMO partner (if CAMO is used) or no BDR node at all (if Eager All Node Replication is used), but only non-BDR nodes, e.g. a the Physical Standby.

Sub-Groups

A Group may also contain zero or more sub-groups. Each sub-group can be allocated to a specific purpose within the top-level parent group. The `node_group_type` specifies the type when the sub-group is created.

Subscriber-Only Groups

BDR 3.7.5 and above supports a new kind of node, called `subscriber-only` node. As the name suggests, this type of node only subscribes to replication changes from other nodes in the cluster, but no other nodes receive replication changes from `subscriber-only` nodes. This is somewhat similar to Logical Standby nodes, but in contrast to Logical Standby, the `subscriber-only` nodes are fully joined node to the cluster. They can receive replication changes from all other nodes in the cluster and hence they are not impacted by unavailability or parting of any one node in the cluster.

Also unlike `pglogical` node, a `subscriber-only` node is a fully joined BDR node and hence it receives all replicated DDLs and acts on those. It also uses Raft to consistently report its status to all nodes in the cluster. The `subscriber-only` node does not have Raft voting rights and hence neither can become a Raft leader nor participate in the leader election. Also, while it receives replicated DDLs, it does not participate in DDL or DML lock acquisition. In

other words, a currently down `subscriber-only` node won't stop a DML lock being acquired.

The `subscriber-only` node forms the building block for BDR Tree topology. In this topology, there are a small number of fully active nodes, replicating changes in all directions, and there are a large number of `subscriber-only` nodes that only receive changes, but never send any changes to any other node in the cluster. This topology avoids connection explosion caused due to a large number of nodes, yet provide extremely large number of `leaf` nodes that can be used to consume the data.

In order to make use of `subscriber-only` nodes, the user must first create a BDR group of type 'subscriber-only'. It should be a subgroup of the group from which the member nodes will receive the replication changes. Once the subgroup is created, all nodes that intend to become `subscriber-only` nodes should join the subgroup. More than one subgroup of 'subscriber-only' type can be created and they can have different parent groups.

Once a node successfully joins the 'subscriber-only' subgroup, it will become a `subscriber-only` node and start receiving replication changes for the parent group. Any changes made directly on the `subscriber-only` node will not be replicated.

See `bdr.create_node_group()` to know how to create a subgroup of a specific type and belonging to a specific parent group.

Notes

Since a `subscriber-only` node doesn't replicate changes to any node in the cluster, it can't act as a source for syncing replication changes when a node is parted from the cluster. But if the `subscriber-only` node had already received and applied replication changes from the parted node that no other node in the cluster currently has, then that will cause inconsistency between the nodes.

For now, this can be solved by setting `pglogical.standby_slot_names` and `pglogical.standby_slots_min_confirmed` appropriately so that there is always a fully active BDR node that is ahead of the `subscriber-only` nodes. See pglogical documentation to show how to use these configuration parameters effectively.

This will be improved in a future release. We may either allow `subscriber-only` nodes to be ahead in the replication and then use them as replication source for sync or simply provide ways to optionally remove the inconsistent `subscriber-only` nodes from the cluster when another fully joined node is parted. RM20306 tracks the development task.

Decoding Worker

BDR3.7 provides an option to enable decoding worker process that performs decoding once, no matter how many nodes are being sent data. This introduces a new process, the wal decoder, on each BDR node. One WAL Sender process still exists for each connection, but these processes now just perform the task of sending and receiving data. Taken together these changes reduce the CPU overhead of larger BDR groups and also allow higher replication throughput since the WAL Sender process now spends more time on communication.

`enable_wal_decoder` is an option for each BDR group, which is currently disabled by default. `bdr.alter_node_group_config()` can be used to enable or disable the Decoding Worker for a BDR group.

When the Decoding Worker is enabled, BDR stores `Logical Change Record` (LCR, in short) files to allow buffering of changes between decoding and when all subscribing nodes have received data. LCR files are stored under the `pg_logical` directory within each local node's data directory. The number and size of the LCR files will vary as replication lag increases, so this will also need monitoring. The LCRs not required by any of the BDR nodes are cleaned periodically. The interval between two consecutive cleanups is controlled by `bdr.lcr_cleanup_interval`, which defaults to 3 minutes. The cleanup is disabled when `bdr.lcr_cleanup_interval` is zero.

When disabled, logical decoding is performed by the WAL Sender process for each node subscribing to each node. In this case, no LCR files are written.

Even though the Decoding Worker is enabled for a BDR group, following GUCs control the production and usage of LCR per node. By default these are `false`. For production and usage of LCRs we need the Decoding Worker to be enabled for the BDR group and these GUCs to be set to `true` on each of the nodes in BDR group.

- `pglogical.enable_wal_decoder` - when turned `false`, all WAL Senders using LCRs are restarted to use WAL directly. When `true` along with the BDR group config, a Decoding Worker process is started to produce LCR and WAL Senders use LCR.

- `bdr.receive_lcr` - when `true` on the subscribing node, it requests WAL Sender on the publisher node to use LCRs if available.

Notes

As of now, a Decoding Worker decodes changes corresponding to the node where it is running. A Logical standby is sent changes from all the nodes in BDR group through a single source. Hence a WAL sender serving a Logical standby can not use LCRs right now.

A Subscriber-only node receives changes from respective nodes directly and hence a WAL sender serving a Subscriber-only node can use LCRs.

Even though LCRs are produced, the corresponding WALs are still retained similar to the case when Decoding Worker is not enabled. In future, it may be possible to remove WAL corresponding the LCRs, if they are not required otherwise.

For reference, the first 24 characters of an LCR filename are similar to those in a WAL filename. The first 8 characters of the name are all '0' right now. In future they are expected to represent the TimeLineId similar to the first 8 characters of a WAL segment filename. The following sequence of 16 characters of the name is similar to the WAL segment number which is used to track LCR changes against the WAL stream. However, please note that logical changes are reordered according to the commit order of the transactions they belong to. Hence their placement in the LCR segments does not match the placement of corresponding WAL in the WAL segments. The set of last 16 characters represents sub-segment number within an LCR segment. Each LCR file corresponds to a sub-segment. LCR files are binary and variable sized. The maximum size of an LCR file can be controlled by `pglogical.max_lcr_segment_file_size`, which defaults to 1GB.

EDB Postgres Extended 13 and above is required for this feature to work.

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, 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 similar to this:

```
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.

On EDB Postgres Extended, offline nodes also hold back freezing of data to prevent losing conflict resolution data (see: [Origin Conflict Detection](#)).

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 created automatically:

- 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 direct logical standbys, if any.

The user **must not** drop those slots: they have been created automatically by BDR, and will be managed by BDR, which will drop them when/if necessary.

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. Ensure that slot names used by other software do **not** begin with the prefix `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`

Group Replication Slot

The group slot is an internal slot used by BDR primarily to track what's the oldest safe position that any node in the BDR group (including all logical standbys) has caught up to, for any outbound replication from this node.

The group slot name is given by the function `bdr.local_group_slot_name()` .

The group slot can:

- join new nodes to the BDR group without having all existing nodes up and running (although the majority of nodes should be up), without incurring data loss in case the node which was down during join starts replicating again
- part nodes from cluster consistently, even if some nodes have not caught up fully with the parted node
- hold back the freeze point to avoid missing some conflicts (EDB Postgres Extended)
- keep the historical snapshot for timestamp based snapshots (EDB Postgres Extended)

The group slot is usually inactive, and is only fast-forwarded periodically in response to Raft progress messages from other nodes.

WARNING: Do not drop the group slot. Although usually inactive, it is still vital to the proper operation of the EDB Postgres Distributed cluster. If it is dropped, then some or all of the above features will stop working and/or may have incorrect outcomes.

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 `db20xxxxxxxxxxxxxxxxxx` (20 bytes long) using a BDR group named `group20xxxxxxxxxxxxxx` (20 bytes long); the logical replication slot associated to node `a30xxxxxxxxxxxxxxxxxxxxxxxxxx` (30 bytes long) will be called:

```
bdr_db20xxxx3597186_group20xbe9cbd0_a30xxxxxxxxxxxxxxxx7f304a2
```

...since `3597186`, `be9cbd0` and `7f304a2` are respectively the hashes of `db20xxxxxxxxxxxxxxxxxx`, `group20xxxxxxxxxxxxxx` and `a30xxxxxxxxxxxxxxxxxxxxxxxxxx`.

Removing a Node From a BDR Group

Since BDR is designed to recover from extended node outages, you must explicitly tell the system if you are removing a node permanently. If you permanently shut down a node and do not 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 that 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 to allow them to catch up any missing data.

A parted node still is known to BDR, but won't consume resources. A node may 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

Dropping 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 EDB Postgres Distributed 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.

There is also a `bdr.drop_node()` function, but this is used only in emergencies, should there be a problem with parting.

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.pub_repsets
,
ns.sub_repsets
, g.node_group_default_repset AS
default_repset
, 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 shown 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 EDB Postgres Distributed cluster yet.
- `JOIN_START` : `bdr.join_node_group()` begins to join the local node to an existing EDB Postgres Distributed cluster.
- `JOINING` : The node join has 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 has 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 EDB Postgres Distributed 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`

This function 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 a record for the local node with the associated public connection string. There can be only one local record, so once it's created, the function will error if run again.

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

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

bdr.drop_node

Drops a node. This function is *not intended for regular use* and should only be executed under the instructions of Technical 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 Technical 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 removes metadata for a given node from the local database. The node can be either the local node, in which case all the node metadata are removed, including info about remote nodes are removed; or it can be the remote node, in which case only metadata for that specific node is removed.

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

This function creates a BDR group with the local node as the only member of the group.

Synopsis

```
bdr.create_node_group(node_group_name text,
```

```
parent_group_name text DEFAULT NULL,
join_node_group boolean DEFAULT true,
node_group_type text DEFAULT NULL)
```

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.
- `parent_group_name` - The name of the parent group for the subgroup.
- `join_node_group` - This helps a node to decide whether or not to join the group being created by it. The default value is true. This is used when a node is creating a shard group that it does not want to join. This can be false only if `parent_group_name` is specified.
- `node_group_type` - The valid values are NULL, 'subscriber-only', 'datanode', 'read coordinator' and 'write coordinator'. 'subscriber-only' type is used to create a group of nodes that only receive changes from the fully joined nodes in the cluster, but they never send replication changes to other nodes. See [Subscriber-Only Nodes] for more details. Datanode implies that the group represents a shard, whereas the other values imply that the group represents respective coordinators. Except 'subscriber-only', the rest three values are reserved for future use. NULL implies a normal general purpose node group will be created.

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`

This function changes the 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,
                            num_writers int DEFAULT
NULL,
                            enable_wal_decoder boolean DEFAULT NULL)
```

Parameters

- `node_group_name` - Name of an existing BDR group; the local node must be part of the group.

- `insert_to_update` - Reserved for backwards compatibility reasons.
- `update_to_insert` - Reserved for backwards compatibility reasons. versions of BDR. Use `bdr.alter_node_set_conflict_resolver` instead.**
- `ignore_redundant_updates` - Reserved for backwards compatibility reasons.
- `check_full_tuple` - Reserved for backwards compatibility reasons.
- `apply_delay` - Reserved for backwards compatibility reasons
- `check_constraints` - Whether the apply process will check the constraints when writing replicated data.
- `num_writers` - number of parallel writers for subscription backing this node group, -1 means the default (as specified by the pglogical GUC `pglogical.writers_per_subscription`) will be used. Valid values are either -1 or a positive integer.
- `enable_wal_decoder` - Enables/disables the Decoding Worker process.

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`

This function 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 the 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 the 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 that 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`

This function promotes a local logical standby node to a full member of the 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`

This function 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 return 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`, on consensus failure, this function will set the state of the given node only on the local node. In such a 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 (i.e. the majority of the nodes are down) or if the parting process gets stuck for whatever reason. But it is important to take into account that when the parting node that was receiving writes, the parting process may take a 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`

This function 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 the 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`

This function 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. Returns the number of subscriptions affected by this operation.

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 the 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`

This function disables either the specified subscription or all the subscriptions of the local BDR node. Optionally, it can also immediately stop all the workers associated with the disabled subscriptions. Also known as pause subscription. No error is thrown if the subscription is already disabled. Returns the number of subscriptions affected by this operation.

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 the 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 the 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 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 (EDB Postgres Extended).

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.

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.
- `--node-group-name` - Group to join, defaults to the same group as source 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). (EDB Postgres Extended)
- `-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

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 replication 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.

2.6 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, REINDEX)
- 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 replication, 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.

The `bdr.ddl_replication` parameter can only be set by the `bdr_superuser`, `superuser`, or in the config file.

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`, `superuser`, or in the config file.

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 replicated 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 may 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, such as when we add or modify a constraint such as a unique constraint, check constraint or NOT NULL constraint. 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.

Note that the global DML lock holds an EXCLUSIVE LOCK on the table on each node, so will block DML, other DDL, VACUUMs and index commands against that table while it runs. This is true even if the global DML lock is held for a command that would not normally take an EXCLUSIVE LOCK or higher.

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.

If a DDL statement is not replicated, no global locks will be acquired.

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 can 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].

DDL executed on temporary tables never need global locks.

ALTER or DROP of an object created in current transaction does not require global DML lock.

Monitoring of global DDL locks and global DML locks is shown in the [Monitoring](#) chapter.

Minimizing the Impact of DDL

Good operational advice for any database, these points become even more important with BDR:

- 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.
- For `ALTER TABLE`, please use `ADD CONSTRAINT NOT VALID`, followed by another transaction with `VALIDATE CONSTRAINT`, rather than just using `ADD CONSTRAINT`. Note that `VALIDATE CONSTRAINT` will wait until replayed on all nodes, which gives a noticeable delay to receive confirmations.
- When indexing, use `CONCURRENTLY` option whenever possible.

An alternate way of executing long running DDL is to disable DDL replication and then to execute the DDL statement separately on each node. That can still be done using a single SQL statement, as shown in the example below. Note that global locking rules still apply, so be careful not to lock yourself out with this type of usage, which should be seen as more of a workaround than normal usage.

```
SELECT
bdr.run_on_all_nodes($ddl$
    CREATE INDEX CONCURRENTLY index_a ON
table_a(i);
$ddl$);
```

We recommend using the `bdr.run_on_all_nodes()` technique above with `CREATE INDEX CONCURRENTLY`, noting that DDL replication must be disabled for whole session because `CREATE INDEX CONCURRENTLY` is a multi-transaction command. `CREATE INDEX` should be avoided on production systems since it prevents writes while it executes. `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`), which is available in PG12+ or 2QPG11+.

`REINDEX` or `REINDEX CONCURRENTLY` on an invalid index will fail to execute on a BDR node. The invalid indexes must be dropped and created again. The invalid indexes must be dropped using `DROP INDEX .. IF EXISTS`. `DROP INDEX` or `DROP INDEX CONCURRENTLY` without `IF EXISTS` clause on an invalid index will fail on a BDR node when DDL replication is enabled.

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

```
$ export PGOPTIONS="-c
bdr.ddl_replication=off"
$ pg_restore --section=post-data
```

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 `SET` command as the `bdr_superuser` to update the `bdr.ddl_locking` value.

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.

Note that any DDL that references or relies upon a temporary object cannot be replicated by BDR and will throw an ERROR, if executed with DDL replication enabled.

BDR DDL Command Handling Matrix

Following table describes which utility or DDL commands are allowed, which are replicated and what type of global lock they take when they are replicated.

For some more complex statements like **ALTER TABLE** these can differ depending on the sub-command(s) executed. Every such command has detailed explanation under the following table.

Command	Allowed	Replicated	Lock
ALTER AGGREGATE	Y	Y	DDL
ALTER CAST	Y	Y	DDL
ALTER COLLATION	Y	Y	DDL
ALTER CONVERSION	Y	Y	DDL
ALTER DATABASE	Y	N	N
ALTER DATABASE LINK	Y	Y	DDL
ALTER DEFAULT PRIVILEGES	Y	Y	DDL
ALTER DIRECTORY	Y	Y	DDL
ALTER DOMAIN	Y	Y	DDL
ALTER EVENT TRIGGER	Y	Y	DDL
ALTER EXTENSION	Y	Y	DDL
ALTER FOREIGN DATA WRAPPER	Y	Y	DDL
ALTER FOREIGN TABLE	Y	Y	DDL
ALTER FUNCTION	Y	Y	DDL
ALTER INDEX	Y	Y	DDL
ALTER LANGUAGE	Y	Y	DDL
ALTER LARGE OBJECT	N	N	N
ALTER MATERIALIZED VIEW	Y	N	N
ALTER OPERATOR	Y	Y	DDL
ALTER OPERATOR CLASS	Y	Y	DDL
ALTER OPERATOR FAMILY	Y	Y	DDL
ALTER PACKAGE	Y	Y	DDL
ALTER POLICY	Y	Y	DDL
ALTER PROCEDURE	Y	Y	DDL
ALTER PROFILE	Y	Y	DDL
ALTER PUBLICATION	Y	Y	DDL
ALTER QUEUE	Y	Y	DDL
ALTER QUEUE TABLE	Y	Y	DDL
ALTER REDACTION POLICY	Y	Y	DDL
ALTER RESOURCE GROUP	Y	N	N
ALTER ROLE	Y	Y	DDL
ALTER ROUTINE	Y	Y	DDL
ALTER RULE	Y	Y	DDL
ALTER SCHEMA	Y	Y	DDL
ALTER SEQUENCE	Details	Y	DML
ALTER SERVER	Y	Y	DDL
ALTER SESSION	Y	N	N
ALTER STATISTICS	Y	Y	DDL
ALTER SUBSCRIPTION	Y	Y	DDL
ALTER SYNONYM	Y	Y	DDL
ALTER SYSTEM	Y	N	N
ALTER TABLE	Details	Y	Details

Command	Allowed	Replicated	Lock
ALTER TABLESPACE	Y	N	N
ALTER TEXT SEARCH CONFIGURATION	Y	Y	DDL
ALTER TEXT SEARCH DICTIONARY	Y	Y	DDL
ALTER TEXT SEARCH PARSER	Y	Y	DDL
ALTER TEXT SEARCH TEMPLATE	Y	Y	DDL
ALTER TRIGGER	Y	Y	DDL
ALTER TYPE	Y	Y	DDL
ALTER USER MAPPING	Y	Y	DDL
ALTER VIEW	Y	Y	DDL
ANALYZE	Y	N	N
BEGIN	Y	N	N
CHECKPOINT	Y	N	N
CLOSE	Y	N	N
CLOSE CURSOR	Y	N	N
CLOSE CURSOR ALL	Y	N	N
CLUSTER	Y	N	N
COMMENT	Y	Details	DDL
COMMIT	Y	N	N
COMMIT PREPARED	Y	N	N
COPY	Y	N	N
COPY FROM	Y	N	N
CREATE ACCESS METHOD	Y	Y	DDL
CREATE AGGREGATE	Y	Y	DDL
CREATE CAST	Y	Y	DDL
CREATE COLLATION	Y	Y	DDL
CREATE CONSTRAINT	Y	Y	DDL
CREATE CONVERSION	Y	Y	DDL
CREATE DATABASE	Y	N	N
CREATE DATABASE LINK	Y	Y	DDL
CREATE DIRECTORY	Y	Y	DDL
CREATE DOMAIN	Y	Y	DDL
CREATE EVENT TRIGGER	Y	Y	DDL
CREATE EXTENSION	Y	Y	DDL
CREATE FOREIGN DATA WRAPPER	Y	Y	DDL
CREATE FOREIGN TABLE	Y	Y	DDL
CREATE FUNCTION	Y	Y	DDL
CREATE INDEX	Y	Y	DML
CREATE LANGUAGE	Y	Y	DDL
CREATE MATERIALIZED VIEW	Y	N	N
CREATE OPERATOR	Y	Y	DDL
CREATE OPERATOR CLASS	Y	Y	DDL
CREATE OPERATOR FAMILY	Y	Y	DDL
CREATE PACKAGE	Y	Y	DDL
CREATE PACKAGE BODY	Y	Y	DDL
CREATE POLICY	Y	Y	DML

Command	Allowed	Replicated	Lock
CREATE PROCEDURE	Y	Y	DDL
CREATE PROFILE	Y	Y	DDL
CREATE PUBLICATION	Y	Y	DDL
CREATE QUEUE	Y	Y	DDL
CREATE QUEUE TABLE	Y	Y	DDL
CREATE REDACTION POLICY	Y	Y	DDL
CREATE RESOURCE GROUP	Y	N	N
CREATE ROLE	Y	Y	DDL
CREATE ROUTINE	Y	Y	DDL
CREATE RULE	Y	Y	DDL
CREATE SCHEMA	Y	Y	DDL
CREATE SEQUENCE	Details	Y	DDL
CREATE SERVER	Y	Y	DDL
CREATE STATISTICS	Y	Y	DDL
CREATE SUBSCRIPTION	Y	Y	DDL
CREATE SYNONYM	Y	Y	DDL
CREATE TABLE	Details	Y	DDL
CREATE TABLE AS	Details	Y	DDL
CREATE TABLESPACE	Y	N	N
CREATE TEXT SEARCH CONFIGURATION	Y	Y	DDL
CREATE TEXT SEARCH DICTIONARY	Y	Y	DDL
CREATE TEXT SEARCH PARSER	Y	Y	DDL
CREATE TEXT SEARCH TEMPLATE	Y	Y	DDL
CREATE TRANSFORM	Y	Y	DDL
CREATE TRIGGER	Y	Y	DDL
CREATE TYPE	Y	Y	DDL
CREATE TYPE BODY	Y	Y	DDL
CREATE USER MAPPING	Y	Y	DDL
CREATE VIEW	Y	Y	DDL
DEALLOCATE	Y	N	N
DEALLOCATE ALL	Y	N	N
DECLARE CURSOR	Y	N	N
DISCARD	Y	N	N
DISCARD ALL	Y	N	N
DISCARD PLANS	Y	N	N
DISCARD SEQUENCES	Y	N	N
DISCARD TEMP	Y	N	N
DO	Y	N	N
DROP ACCESS METHOD	Y	Y	DDL
DROP AGGREGATE	Y	Y	DDL
DROP CAST	Y	Y	DDL
DROP COLLATION	Y	Y	DDL
DROP CONSTRAINT	Y	Y	DDL
DROP CONVERSION	Y	Y	DDL
DROP DATABASE	Y	N	N

Command	Allowed	Replicated	Lock
DROP DATABASE LINK	Y	Y	DDL
DROP DIRECTORY	Y	Y	DDL
DROP DOMAIN	Y	Y	DDL
DROP EVENT TRIGGER	Y	Y	DDL
DROP EXTENSION	Y	Y	DDL
DROP FOREIGN DATA WRAPPER	Y	Y	DDL
DROP FOREIGN TABLE	Y	Y	DDL
DROP FUNCTION	Y	Y	DDL
DROP INDEX	Y	Y	DDL
DROP LANGUAGE	Y	Y	DDL
DROP MATERIALIZED VIEW	Y	N	N
DROP OPERATOR	Y	Y	DDL
DROP OPERATOR CLASS	Y	Y	DDL
DROP OPERATOR FAMILY	Y	Y	DDL
DROP OWNED	Y	Y	DDL
DROP PACKAGE	Y	Y	DDL
DROP PACKAGE BODY	Y	Y	DDL
DROP POLICY	Y	Y	DDL
DROP PROCEDURE	Y	Y	DDL
DROP PROFILE	Y	Y	DDL
DROP PUBLICATION	Y	Y	DDL
DROP QUEUE	Y	Y	DDL
DROP QUEUE TABLE	Y	Y	DDL
DROP REDACTION POLICY	Y	Y	DDL
DROP RESOURCE GROUP	Y	N	N
DROP ROLE	Y	Y	DDL
DROP ROUTINE	Y	Y	DDL
DROP RULE	Y	Y	DDL
DROP SCHEMA	Y	Y	DDL
DROP SEQUENCE	Y	Y	DDL
DROP SERVER	Y	Y	DDL
DROP STATISTICS	Y	Y	DDL
DROP SUBSCRIPTION	Y	Y	DDL
DROP SYNONYM	Y	Y	DDL
DROP TABLE	Y	Y	DML
DROP TABLESPACE	Y	N	N
DROP TEXT SEARCH CONFIGURATION	Y	Y	DDL
DROP TEXT SEARCH DICTIONARY	Y	Y	DDL
DROP TEXT SEARCH PARSER	Y	Y	DDL
DROP TEXT SEARCH TEMPLATE	Y	Y	DDL
DROP TRANSFORM	Y	Y	DDL
DROP TRIGGER	Y	Y	DDL
DROP TYPE	Y	Y	DDL
DROP TYPE BODY	Y	Y	DDL

Command	Allowed	Replicated	Lock
DROP USER MAPPING	Y	Y	DDL
DROP VIEW	Y	Y	DDL
EXECUTE	Y	N	N
EXPLAIN	Y	Details	Details
FETCH	Y	N	N
GRANT	Y	Details	DDL
GRANT ROLE	Y	Y	DDL
IMPORT FOREIGN SCHEMA	Y	Y	DDL
LISTEN	Y	N	N
LOAD	Y	N	N
LOAD ROW DATA	Y	Y	DDL
LOCK TABLE	Y	N	N
MOVE	Y	N	N
NOTIFY	Y	N	N
PREPARE	Y	N	N
PREPARE TRANSACTION	Y	N	N
REASSIGN OWNED	Y	Y	DDL
REFRESH MATERIALIZED VIEW	Y	N	N
REINDEX	Y	N	N
RELEASE	Y	N	N
RESET	Y	N	N
REVOKE	Y	Details	DDL
REVOKE ROLE	Y	Y	DDL
ROLLBACK	Y	N	N
ROLLBACK PREPARED	Y	N	N
SAVEPOINT	Y	N	N
SECURITY LABEL	Y	Details	DDL
SELECT INTO	Details	Y	DDL
SET	Y	N	N
SET CONSTRAINTS	Y	N	N
SHOW	Y	N	N
START TRANSACTION	Y	N	N
TRUNCATE TABLE	Y	Details	Details
UNLISTEN	Y	N	N
VACUUM	Y	N	N

ALTER SEQUENCE

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

`ALTER SEQUENCE ... RENAME` is not supported on gallocc sequences (only). `ALTER SEQUENCE ... SET SCHEMA` is not supported on gallocc sequences (only).

ALTER TABLE

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

ALTER TABLE Disallowed Commands

Some variants of `ALTER TABLE` are currently not allowed on a BDR node:

- `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.
- `RENAME` - cannot rename an Autopartitioned table.
- `SET SCHEMA` - cannot set the schema of an Autopartitioned 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.
- `ALTER TABLE ... ADD FOREIGN KEY` - Is not supported if current user does not have permission to read the referenced table, or if the referenced table has RLS restrictions enabled which current user cannot bypass.

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

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

Starting BDR 3.7.4, certain types of constraints, such as CHECK and FOREIGN KEY constraints, can be added without taking a DML lock. But this requires a 2-step process of first creating a NOT VALID constraint and then validating the constraint in a separate transaction via `ALTER TABLE ... VALIDATE CONSTRAINT` command. See [Adding a CONSTRAINT](#) for more details.

ALTER TABLE Locking

The following variants of `ALTER TABLE` will only take DDL lock and **not** a DML lock:

- `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 (currently only available on EDB Postgres Extended and EDB Postgres Advanced)
- `ALTER TABLE ... ALTER COLUMN ... SET STATISTICS`
- `ALTER TABLE ... VALIDATE CONSTRAINT`
- `ALTER TABLE ... ATTACH PARTITION`
- `ALTER TABLE ... DETACH PARTITION`
- `ALTER TABLE ... ENABLE TRIGGER (ENABLE REPLICAS 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 = [, ...])`
- `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.

ALTER TABLE Examples

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 tables on a BDR
node
```

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 tables on a BDR
node
```

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 EDB Postgres Distributed 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 EDB Postgres Distributed 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 EDB Postgres Distributed 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 not affect
replicated tables on a BDR node
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.

Note

This currently only works on EDB Postgres Extended and EDB Postgres Advanced.

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.

COMMENT ON

All variants of COMMENT ON are allowed, but `COMMENT ON TABLESPACE/DATABASE/LARGE OBJECT` will not be replicated.

CREATE SEQUENCE

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

CREATE TABLE

Generally `CREATE TABLE` is supported but `CREATE TABLE WITH OIDS` is not allowed on a BDR node.

CREATE TABLE AS and SELECT INTO

`CREATE TABLE AS` and `SELECT INTO` are only allowed on EDB Postgres Extended and EDB Postgres Advanced and only if any sub-commands are also allowed.

You can instead achieve the same effect using, in case the `CREATE TABLE AS` is not supported on your variant of Postgres:

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

EXPLAIN

Generally `EXPLAIN` is allowed, but because `EXPLAIN ANALYZE` can have side effects on the database, there are some restrictions on it.

EXPLAIN ANALYZE Replication

EXPLAIN ANALYZE will follow replication rules of the analyzed statement.

EXPLAIN ANALYZE Locking

EXPLAIN ANALYZE will follow locking rules of the analyzed statement.

GRANT and REVOKE

Generally `GRANT` and `REVOKE` statements are supported, however `GRANT/REVOKE ON TABLESPACE/LARGE OBJECT` will not be replicated.

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()`.

SECURITY LABEL

All variants of `SECURITY LABEL` are allowed, but `SECURITY LABEL ON TABLESPACE/DATABASE/LARGE OBJECT` will not be replicated.

TRUNCATE Replication

`TRUNCATE` command is replicated as DML, not as DDL statement, so whether the `TRUNCATE` on table is replicated depends on replication set settings for each affected table.

TRUNCATE Locking

Even though `TRUNCATE` is not replicated same way as other DDL, it may acquire the global DML lock when `bdr.truncate_locking` is set to `on`.

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.

Restricted DDL Workarounds

Some of the limitations of BDR DDL operation handling can be worked around, often splitting up the operation into smaller changes can produce desired result that is either not allowed as single statement or requires excessive locking.

Adding a CONSTRAINT

Starting BDR 3.7.4, a CHECK and FOREIGN KEY constraint can be added without requiring a DML lock. This requires a 2-step process.

- `ALTER TABLE ... ADD CONSTRAINT ... NOT VALID`
- `ALTER TABLE ... VALIDATE CONSTRAINT`

These steps must be executed in two different transactions. Both these steps only take DDL lock on the table and hence can be run even when one or more nodes are down. But in order to validate a constraint, BDR must ensure that all nodes in the cluster has seen the `ADD CONSTRAINT` command and the node validating the constraint has applied replication changes from all other nodes prior to creating the NOT VALID constraint on those nodes. So even though the new mechanism does not need all nodes to be up while validating the constraint, it still requires that all nodes should have applied the `ALTER TABLE .. ADD CONSTRAINT ... NOT VALID` command and made enough progress. BDR will wait for a consistent state to be reached before validating the constraint.

Note that the new facility requires the cluster to run with Raft protocol version 24 and beyond. If the Raft protocol is not yet upgraded, the old mechanism will be used, resulting in a DML lock request.

Note

This currently only works on EDB Postgres Extended and EDB Postgres Advanced.

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 the general case for changing column type, first add a column of the desired type:

```
ALTER TABLE mytable ADD COLUMN newcolumn newtype;
```

Create a trigger defined as `BEFORE INSERT OR UPDATE ON mytable FOR EACH ROW ..`, which assigns `NEW.newcolumn` to `NEW.oldcolumn` so that new writes to the table update the new column automatically.

`UPDATE` the table in batches to copy the value of `oldcolumn` to `newcolumn` using a `PROCEDURE` with embedded transactions. Batching the work will help reduce replication lag if it is a big table. Updating by range of IDs or whatever method you prefer is fine, or the whole table in one go for smaller tables.

`CREATE INDEX ...` any required indexes on the new column. It is safe to use `CREATE INDEX ... CONCURRENTLY` run individually without DDL replication on each node, to reduce lock durations.

`ALTER` the column to add a `NOT NULL` and `CHECK` constraints, if required.

`BEGIN` a transaction, `DROP` the trigger you added, `ALTER TABLE` to add any `DEFAULT` required on the column, `DROP` the old column, and `ALTER TABLE mytable RENAME COLUMN newcolumn TO oldcolumn`, then `COMMIT`.

Because you are dropping a column, you may have to re-create views, procedures, etc. that depend on the table. Be careful if you `CASCADE` drop the column, as you will need to ensure you re-create everything that referred to it.

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`
- `bdr.column_timestamps_enable`
- `bdr.column_timestamps_disable`

Sequence Management

- `bdr.alter_sequence_set_kind`

Stream Triggers

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

2.7 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/predefined roles, named similarly to the PostgreSQL default/predefined 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 required by applications running BDR.
- `bdr_read_all_conflicts` - can view *all* conflicts in `bdr.conflict_history`.

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

Managing BDR does not require that administrators have access to user data.

Arrangements for securing conflicts are discussed here [Logging Conflicts to a Table](#).

Conflicts may be monitored using the `BDR.conflict_history_summary` view.

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.

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).

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.

Privilege Restrictions

BDR enforces additional restrictions, effectively preventing the use of DDL that relies solely on `TRIGGER` or `REFERENCES` privileges. The following sub-

sections explain these.

`GRANT ALL` will still grant both TRIGGER and REFERENCES privileges, so it is recommended that you state privileges explicitly, e.g. `GRANT SELECT, INSERT, UPDATE, DELETE, TRUNCATE` instead of `ALL`.

Foreign Key Privileges

`ALTER TABLE ... ADD FOREIGN KEY` is only supported if the user has SELECT privilege on the referenced table, or if the referenced table has RLS restrictions enabled which the current user cannot bypass.

Thus, the REFERENCES privilege is not sufficient to allow creation of a Foreign Key with BDR. Relying solely on the REFERENCES privilege is not typically useful since it makes the validation check execute using triggers rather than a table scan, so is typically too expensive to used successfully.

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. The TRIGGER privilege is seldom used and PostgreSQL Core Team has said "The separate TRIGGER permission is something we consider obsolescent."

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.

BDR Default/Predefined Roles

BDR predefined 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 privilege on

- `bdr.conflict_history_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 privilege on

- `bdr.bdr_edition`
- `bdr.bdr_version`
- `bdr.bdr_version_num`
- `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

All privileges from `bdr_read_all_stats` , plus

EXECUTE privilege on

- `bdr.monitor_group_versions`
- `bdr.monitor_group_raft`
- `bdr.monitor_local_replslots`

`bdr_application`

EXECUTE privilege 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.get_configured_camo_partner_of`
- `bdr.get_configured_camo_origin_for`
- `bdr.global_lock_table`
- `bdr.is_camo_partner_connected`
- `bdr.is_camo_partner_ready`
- `bdr.logical_transaction_status`
- `bdr.ri_fkey_trigger`
- `bdr.seq_nextval`
- `bdr.seq_currval`
- `bdr.seq_lastval`
- `bdr.trigger_get_committs`
- `bdr.trigger_get_conflict_type`
- `bdr.trigger_get_origin_node_id`
- `bdr.trigger_get_row`
- `bdr.trigger_get_type`
- `bdr.trigger_get_xid`
- `bdr.wait_for_camo_partner_queue`
- `bdr.wait_slot_confirm_lsn`

Note that many of the above functions have additional privileges required before they can be used, for example, you must be the table owner to successfully execute `bdr.alter_sequence_set_kind`. These additional rules are documented with each specific function.

`bdr_read_all_conflicts`

BDR logs conflicts into the `bdr.conflict_history` table. Conflicts are visible to table owners (only), so no extra privileges are required to read the conflict history. If it is useful to have a user that can see conflicts for *all* tables, you may optionally grant the role `bdr_read_all_conflicts` to that user.

2.8 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 that all nodes converge to the same result, and establishes commit-order-like semantics on the whole cluster.

This chapter covers row-level conflicts with standard data types in detail.

Conflict handling is configurable, as described later in this chapter. Conflicts can be detected and handled differently for each table using conflict triggers, described in the [Stream Triggers](#) chapter.

Column-level conflict detection and resolution is available with BDR, described in the [CLCD](#) chapter.

If you wish to avoid conflicts, you can use these features in BDR.

- Conflict-free data types (CRDTs) - described in the [CRDT](#) chapter.
- Eager replication - described in the [Eager Replication](#) chapter.

By default, all conflicts are logged to `bdr.conflict_history`. If conflicts are possible then table owners should monitor for them, analyze to see how they can be avoided or plans made to handle them regularly as an application task. The LiveCompare tool is also available to scan regularly for divergence.

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.

Upgrade Notes

All the SQL visible interfaces are in the `bdr` schema. All the previously deprecated interfaces in the `bdr_conflicts` or `bdr_crddt` schema were removed and will **not** work on 3.7+ nodes or in groups that contain at least one 3.7+ node. Please use the ones in `bdr` schema that are already present in all BDR versions.

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 the transactions commit, each transaction is individually valid on the node it committed on, but would not be valid if applied on another node that did 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 if no `PRIMARY KEY` exists, the same values for a single `UNIQUE` constraint).

BDR handles this by retaining the most recently inserted tuple of the two, according to the originating host's timestamps, unless overridden by a user-defined conflict handler.

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, 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 produce a `multiple_unique_conflicts` conflict.

In case of such a conflict, some rows must be removed in order for replication to continue. Depending on the resolver setting for `multiple_unique_conflicts`, the apply process will either exit with error, skip the incoming row, or delete some of the rows automatically. The automatic deletion will always try to preserve the row with the correct `PRIMARY KEY` and delete the others.

Warning

In case of multiple rows conflicting this way, if the result of conflict resolution is to proceed with the insert operation, some of the data will always be deleted!

It's also possible to define a different behaviour using a conflict trigger.

UPDATE/UPDATE Conflicts

Where two concurrent `UPDATE` s 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 the 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 (the origin is the node that 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 a 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 the 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 would 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 for PostgreSQL applications that UPDATE PKs, 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, this 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 the 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`. However in certain situations, divergence will happen even with `update_if_newer`, namely when 2 different rows both get updated concurrently to the same new primary key.

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 to avoid concurrent changes, make those changes using Eager replication.

Warning

In case the conflict resolution of `update_pkey_exists` conflict results in update, one of the rows will always be deleted!

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 will raise a `multiple_unique_conflicts` conflict.

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 above about using multiple unique constraints.

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 `DELETE`d 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 received 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.

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 a `delete_recently_updated` conflict is to `skip` the deletion. However, you can configure the resolution or a conflict trigger can be configured to handle it.

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](#)).

When this happens, the `update_missing` conflict is generated. The default conflict resolver is `insert_or_skip`, though `insert_or_error` or `skip` may be used instead. Resolvers that do insert-or-action will first try to `INSERT` a new row based on data from the `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](#)).

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 `DELETE`s 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`.

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 the 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 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 difficulties. 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 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; it is 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 concurrently on other nodes to the TRUNCATE.

Users may wish to take one of the following actions:

- 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 `bdr.truncate_locking = 'on'` to set the TRUNCATE command's locking behavior. Determines whether TRUNCATE obeys the `bdr.ddl_locking` setting. This is not the default behaviour for TRUNCATE since it requires all nodes to be up, so may not be possible or desirable in all cases.

Exclusion Constraint Conflicts

BDR does not 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 do not drop existing ones when converting a standalone database to BDR, you should expect replication to break. To get it to progress again, remove or alter 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 differ. 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 is not 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.

The 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 an UPDATE statement that did not change a value of a toasted column will produce a row without that column. As mentioned in [INSERT/UPDATE Conflicts], BDR will produce an error if an `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 the following workload on a EDB Postgres Distributed 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 are 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 a statement, the protection against these concurrency issues will not be present.

Warning

The additional WAL logging of TOAST is done using the `BEFORE UPDATE` trigger. This trigger must be sorted alphabetically last (based on trigger name) among all `BEFORE UPDATE` triggers on the table. It's prefixed with `zzzz_bdr_` to make this easier, but make sure you don't create any trigger with name that would sort after it, otherwise the protection against the concurrency issues will not be present. This trigger is not created or used when using BDR with EDB Postgres Extended.

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 value 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. For example:

- receiving parcels on one node, but delivering parcels via another node.
- 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 these 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 that 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](#).

On EDB Postgres Extended, BDR will automatically hold back the freezing of rows while a node is down to handle this situation gracefully without the need for changing parameter settings.

On other variants of Postgres, users may need to manage this situation with some care:

Freezing normally occurs when a row being vacuumed is older than `vacuum_freeze_min_age` xids from the current xid, which means that you need to configure suitably high values for these parameters:

- `vacuum_freeze_min_age`
- `vacuum_freeze_table_age`
- `autovacuum_freeze_max_age`

Values should be chosen based upon the transaction rate, giving a grace period of downtime before any conflict data is removed from the database server. For example, a node performing 1000 TPS could be down for just over 5.5 days before conflict data is removed, when `vacuum_freeze_min_age` is set to 500 million. The CommitTS datastructure will take on-disk space of 5 GB with that setting, so lower transaction rate systems may benefit from lower settings.

Initially recommended settings would be:

```
# 1 billion = 10GB
autovacuum_freeze_max_age = 1000000000
```

```
vacuum_freeze_min_age = 500000000

# 90% of autovacuum_freeze_max_age
vacuum_freeze_table_age = 900000000
```

Note that:

- `autovacuum_freeze_max_age` can only be set at server start.
- `vacuum_freeze_min_age` is user-settable, so using a low value will freeze rows early and could result in conflicts being ignored. `autovacuum_freeze_min_age` and `toast.autovacuum_freeze_min_age` can also be set for individual tables.
- running the `CLUSTER` or `VACUUM FREEZE` commands will also freeze rows early and could result in conflicts being ignored.

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 are 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 a 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, this technique allows conflict detection for both UPDATE and DELETE.

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

Note

The row-level conflict resolution is still handled based on the [Conflict Resolution](#) configuration even with row versioning. The way the row version is generated is only useful for detection of conflicts and should not be relied to as authoritative information about which version of row is newer.

To determine the current conflict resolution strategy used for a specific table, refer to the column `conflict_detection` of the view `bdr.tables`.

`bdr.alter_table_conflict_detection`

Allows the table owner to change how 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 the new conflict detection method.

- `method` - which conflict detection method to use.
- `column_name` - which column to use for storing of the column detection data; this 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](#)). This is the only method supported which does not require an 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 to 30618 or below.

Warning

Please note that when changing the conflict detection method from one that uses an extra column to store 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.
- `update_differing` - an incoming update's key row differs from a local row. This can only happen when using [Row Version Conflict Detection].
- `update_origin_change` - an incoming update is modifying a row that was last changed by a different node.
- `update_missing` - an incoming update is trying to modify a row that does not exist.
- `update_recently_deleted` - an incoming update is trying to modify a row that was recently deleted.
- `update_pkey_exists` - an incoming update has modified the [PRIMARY KEY](#) to a value that already exists on the node that is applying the change.
- `multiple_unique_conflicts` - the incoming row conflicts with multiple UNIQUE constraints/indexes in the target table.
- `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].
- `delete_missing` - an incoming delete is trying to remove a row that does not exist.
- `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` - the target table is missing.
- `apply_error_ddl` - an error was thrown by PostgreSQL when applying a replicated DDL command.

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 behaviour in certain corner case scenarios depends on the settings used for `[bdr.create_node_group]` and alternatively for `[bdr.alter_node_group]`.

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

`bdr.alter_node_set_conflict_resolver`

This function sets the behaviour 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 behaviour of conflict resolutions specified via `[bdr.create_node_group]` or `bdr.alter_node_group`.

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 coverages 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.
- `skip_transaction` - skips the whole transaction that has generated the conflict. Can be used for `apply_error_ddl` conflict.
- `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`, `multiple_unique_conflicts`, `update_recently_deleted`, `update_pkey_exists`, `delete_recently_updated` and `delete_missing` conflict types can also be resolved by user-defined logic using [Conflict Triggers](#).

Here is a matrix that will help you individuate what conflict types the conflict resolvers can handle.

	insert_exists	update_differing	update_origin_change	update_missing	update_recently_deleted	update_pkey_exists	delete_recently_updated	delete_missing	target_column_missing	source_column_missing	target_table_missing	multiple_unique_conflicts
error	X	X	X	X	X	X	X	X	X	X	X	X
skip	X	X	X	X	X	X	X	X	X	X	X	X
skip_if_recently_dropped											X	
update_if_newer	X	X	X			X						
update	X	X	X			X	X					X
insert_or_skip				X	X							
insert_or_error				X	X							
ignore									X			
ignore_if_null									X			
use_default_value										X		
conflict_trigger	X	X	X	X	X	X	X	X				X

Default Conflict Resolvers

Conflict Type	Resolver
insert_exists	update_if_newer
update_differing	update_if_newer
update_origin_change	update_if_newer
update_missing	insert_or_skip

Conflict Type	Resolver
update_recently_deleted	skip
update_pkey_exists	update_if_newer
multiple_unique_conflicts	error
delete_recently_updated	skip
delete_missing	skip
target_column_missing	ignore_if_null
source_column_missing	use_default_value
target_table_missing	skip_if_recently_dropped
apply_error_ddl	error

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

`bdr.alter_node_set_log_config`

Set the conflict logging configuration for a node.

Synopsis

```
bdr.alter_node_set_log_config(node_name text,
                             log_to_file bool DEFAULT
true,
                             log_to_table bool DEFAULT true,
NULL,
                             conflict_type text[] DEFAULT
NULL)
                             conflict_resolution text[] DEFAULT
```

Parameters

- `node_name` - name of the node that is being changed

- `log_to_file` - whether to log to the server log file
- `log_to_table` - whether to log to the `bdr.conflict_history` 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 conflict type and resolution it logs.

Logging Conflicts to a Table

Conflicts will be logged to a table if `log_to_table` is set to true. The target table for conflict logging is the `bdr.conflict_history`.

This table is range partitioned on column `local_time`. The table is managed by Autopartition. By default, a new partition is created for every day, and conflicts of the last 1 month are maintained. After that, the old partitions are dropped automatically. Autopartition pre-creates between 7 to 14 partitions in advance. `bdr_superuser` may change these defaults.

Since conflicts generated for all tables managed by BDR are logged to this table, it's important to ensure that only legitimate users can read the conflicted data. We do this by defining ROW LEVEL SECURITY policies on the `bdr.conflict_history` table. Only owners of the tables are allowed to read conflicts on the respective tables. If the underlying tables themselves have RLS policies defined, enabled and enforced, then even owners can't read the conflicts. RLS policies created with the FORCE option also apply to owners of the table. In that case, some or all rows in the underlying table may not be readable even to the owner. So we also enforce a stricter policy on the conflict log table.

The default role `bdr_read_all_conflicts` can be granted to users who need to see all conflict details logged to the `bdr.conflict_history` table, without also granting them `bdr_superuser` role.

The default role `bdr_read_all_stats` has access to a catalog view called `bdr.conflict_history_summary` which does not contain user data, allowing monitoring of any conflicts logged.

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.conflict_history
WHERE local_time > date_trunc('day',
current_timestamp)
GROUP BY 1,2,3
ORDER BY 1,2;
```

```

 nspname | relname | date |
 count
-----+-----+-----+
 my_app  | test    | 2019-04-05 |
 1
(1 row)

```

Data Verification with LiveCompare

LiveCompare is a utility program designed to compare any two databases to verify that they are identical.

LiveCompare is included as part of the BDR Stack and can be aimed at any pair of BDR nodes and, by default, it will compare all replicated tables and report differences. LiveCompare also works with non-BDR data sources such as Postgres and Oracle.

LiveCompare can also be used to continuously monitor incoming rows. It can be stopped and started without losing context information, so it can be run at convenient times.

LiveCompare allows concurrent checking of multiple tables and can be configured to allow checking of a few tables or just a section of rows within a table. Checks are performed by first comparing whole row hashes, then if different, LiveCompare will compare whole rows. LiveCompare avoids overheads by comparing rows in useful-sized batches.

If differences are found, they can be re-checked over a period, allowing for the delays of eventual consistency.

Please refer to the LiveCompare docs for further details.

2.9 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 `INSERTs`) 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 - but not necessarily all - cases.

Using BDR 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 sequence (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 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; 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, KSUIDs 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 two 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 that 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, KSUUUIDs and Other Approaches

There are other ways to generate globally unique ids without using the global sequences that can be used with BDR. For example:

- UUIDs, and their BDR variant, KSUUUIDs
- 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 does not 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 KSUUUIDs

`UUID` keys instead avoid sequences entirely and use 128-bit universal unique identifiers. These are random or pseudorandom values that are so large that it is 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 KSUUUID, which generates values that can be stored using PostgreSQL's standard `UUID` data type. A `KSUUUID` value is similar to `UUIDv1` in that it stores both timestamp and random data, following the `UUID` standard. The difference is that `KSUUUID` is K-Sortable, meaning that it's weakly sortable by timestamp. This makes it more useful as a database key as it produces more compact `btree` indexes, which improves the effectiveness of search, and allows natural time-sorting of result data. Unlike `UUIDv1`, `KSUUUID` values do not include the MAC of the computer on which they were generated, so there should be no security concerns from using `KSUUUID` s.

`KSUUUID` v2 is now recommended in all cases. Values generated are directly sortable with regular comparison operators.

There are two versions of `KSUUUID` in BDR, v1 and v2. The legacy `KSUUUID` v1 is now deprecated but is 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 `KSUUUID` 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 the designer must specify a maximum number of nodes when establishing the schema, and it 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_value)`, 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 is not 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`

Allows the owner of a sequence to set 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].

Synopsis

```
bdr.alter_sequence_set_kind(seqoid regclass, seqkind text)
```

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

Notes

When changing the sequence kind to `galloc`, the first allocated range for that sequence will use the sequence start value as the 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 with 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 that 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 to 30618 or below.

bdr.extract_timestamp_from_timeshard

This function extracts the timestamp component of the `timeshard` sequence. The return value is of type "timestampz".

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

This function extracts 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

This function extracts 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

This function converts 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`

This function 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`

This function compares the `KSUUID` v2 values.

It 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` - `KSUUUID` v2 to compare

Notes

This function is only executed on local node.

```
bdr.extract_timestamp_from_ksuuid_v2
```

This function extracts the timestamp component of `KSUUUID` v2. The return value is of type "timestampz".

Synopsis

```
bdr.extract_timestamp_from_ksuuid_v2(uuid)
```

Parameters

- `UUID` - `KSUUUID` v2 value to extract timestamp from

Notes

This function is only executed on the local node.

KSUUUID v1 Functions

Functions for working with `KSUUUID` v1 data, K-Sortable UUID data(v1).

```
bdr.gen_ksuuid
```

This function generates a new `KSUUUID` 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`

This function compares the `KSUUID` v1 values.

It 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`

This function extracts the timestamp component of `KSUUID` v1 or `UUIDv1` values. The return value is of type "timestampz".

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.

2.10 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 REPLICATA IDENTITY FULL. The bdr.alter_table_conflict_detection function does check that, and will fail with an error otherwise.

Note

This feature is currently only available on EDB Postgres Extended and EDB Postgres Advanced.

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 REPLICATA IDENTITY
FULL;
ALTER TABLE

db=# SELECT bdr.alter_table_conflict_detection(
```



```
db=# 'my_app.test_table'::regclass, 'column_modify_timestamp', 'cts');
alter_table_conflict_detection
-----
t
db=# \d my_app.test_table
```

You will see that the function adds 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 means that `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
timestamptz)
```

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.

If `column_modify_timestamp` is selected as the conflict detection method, 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 (for example, 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.

You can also use the actual commit timestamp, specified with `column_commit_timestamp` as the conflict detection method.

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
19:25:02.590677+02]}
```

(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": "2018-09-23T19:29:55.581823+02:00", "modified": [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": "2018-09-23T19:29:55.581823+02:00"}
(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_cld_` 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);
```

2.11 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). If the full state is dumped using `bdr.crdt_raw_value = on` then the value would only be able to be reloaded with `bdr.crdt_raw_value = on`.

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', 'cts');

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.

Note

This feature is currently only available on EDB Postgres Extended and EDB Postgres Advanced.

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.).

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,
    pnsun   bdr.crdt_pnsun 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 pnsun = pnsun + 1.44     WHERE id = 1;
UPDATE crdt_test SET pnsun = pnsun + 12.20    WHERE id = 2;
UPDATE crdt_test SET pnsun = pnsun + (-24.34) WHERE id = 3;

-- decrement sums
UPDATE crdt_test SET pnsun = pnsun - 7.3      WHERE id = 1;
UPDATE crdt_test SET pnsun = pnsun - 192.83   WHERE id = 2;
UPDATE crdt_test SET pnsun = pnsun - (-12.22) WHERE id = 3;

-- get current sum value
SELECT id, pnsun::numeric, pnsun FROM crdt_test;

-- show internal structure of sum
SELECT id, bdr.crdt_pnsun_to_text(pnsun) FROM crdt_test;

-- reset sum
UPDATE crdt_test SET pnsun = !pnsun WHERE id = 1;

-- get current sum value after the reset
SELECT id, pnsun::numeric, pnsun 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

2.12 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.

BDR integrates with the `synchronous_commit` option of Postgres itself, providing a variant of synchronous replication, which can be used between BDR nodes. BDR also offers two additional replication modes:

- Commit At Most Once (CAMO). This feature solves the problem with knowing whether your transaction has COMMITed (and replicated) or not in case of certain errors during COMMIT. Normally, it might be hard to know whether or not the COMMIT was processed in. With this feature, your application can find out what happened, even if your new database connection is to node than your previous connection. For more info about this feature see the [Commit At Most Once](#) chapter.
- Eager Replication. This is an optional feature to avoid replication conflicts. Every transaction is applied on *all nodes* simultaneously, and commits only if no replication conflicts are detected. This feature does reduce performance, but provides very strong consistency guarantees. For more info about this feature see the [Eager All-Node Replication](#) chapter.

Postgres itself provides [Physical Streaming Replication](#) (PSR), which is uni-directional, but offers a synchronous variant that can be used in combination with BDR.

WARNING

This only works when using a single database per node. When using multiple BDR enabled databases per node, which is not generally recommended, the LSN based confirmations may originate from any one of the databases from a node specified in `synchronous_standby_names` and thus not assure the data is really flushed to disk.

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
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
<code>synchronous_standby_names</code>	req	req	n/a	n/a
<code>synchronous_commit</code>	opt	opt	n/a	n/a
<code>synchronous_replication_availability</code>	opt	opt	opt	n/a
<code>bdr.enable_camo</code>	n/a	n/a	req	n/a
<code>bdr.camo_origin_for</code>	n/a	n/a	req	n/a
<code>bdr.camo_partner_of</code> (on partner node)	n/a	n/a	req	n/a
<code>bdr.commit_scope</code>	n/a	n/a	n/a	req
<code>bdr.global_commit_timeout</code>	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.

2.13 Eager Replication

2.14 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.

The CAMO feature in BDR offers a more general solution and does not require an `INSERT` as described above. When activated via `bdr.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 creating a pair of partner nodes that are two BDR master nodes from the same top level BDR group. In this operation mode, each node in the pair knows the outcome of any recent transaction executed on the other peer, and especially (for our need) knows the outcome of any transaction disconnected during COMMIT. We may refer as "origin" to the node that receives the transactions from the application and "partner" the node that confirms these transactions, but there is no difference in the CAMO configuration for the nodes in the CAMO pair. The pair is symmetric.

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 E](#).

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`, and both part of the same node group `mygroup` the following steps will configure the nodes to be CAMO partners for each other.

1. Create the BDR cluster where nodes `node1` and `node2` are part of `mygroup` node group.
2. Run the function `bdr.add_camo_pair()` on one node:

```
SELECT bdr.add_camo_pair('mygroup', 'node1', 'node2');
```

3. 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 bdr.enable_camo = 'remote_commit_flush';
```

...or to enable for individual transactions, issue this after starting the transaction and before committing it:

```
SET LOCAL bdr.enable_camo = 'remote_commit_flush';
```

Valid values for `bdr.enable_camo` that enable CAMO are:

- `off` (default)
- `remote_write`
- `remote_commit_async`
- `remote_commit_flush` or `on`

See the [Comparison](#) of synchronous replication modes for details about how each mode behaves. Setting `bdr.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 `bdr.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 `bdr.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 `bdr.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'`, a node (i.e. master) will detect whether its CAMO partner is ready; if not, it will temporarily switch to **Local** mode. When in Local mode, a node 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`. The current readiness status of a CAMO partner can be checked with `bdr.is_camo_partner_ready`, while `bdr.node_replication_rates` provides the current estimate of the catchup time.

The switch from CAMO protected to Local Mode is only ever triggered by an actual CAMO transaction. Either because the commit exceeds the `bdr.global_commit_timeout` or in case the CAMO partner is already known disconnected at the time of commit. This switch is independent of the estimated catchup interval. If the CAMO pair is configured to require Raft to switch to Local Mode, this switch will require a majority of nodes to be operational (see the `require_raft` flag for `bdr.add_camo_pair`). This can prevent a split brain situation due to an isolated node from switching to Local Mode. If `require_raft` is not set for the CAMO pair, the origin node will switch to Local Mode immediately.

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 a node 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 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 is the only possible configuration starting with BDR4.

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.

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 **anode 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.

Adding a CAMO pair

The function `bdr.add_camo_pair()` configures an existing pair of BDR nodes to work as a symmetric CAMO pair.

The `require_raft` option controls how and when to switch to Local Mode in case `synchronous_replication_availability` is set to `async`, allowing such a switch in general.

Synopsis

```
bdr.add_camo_pair(node_group text, left_node text, right_node text,
                  require_raft bool)
```

Note

The names `left` and `right` have no special meaning.

Note

Since BDR version 4.0, only symmetric CAMO configurations are supported, i.e. both nodes of the pair act as a CAMO partner for each other.

Changing the configuration of a CAMO pair

The function `bdr.alter_camo_pair()` allows to toggle the `require_raft` flag. Note that it is not currently possible to change the nodes of a pairing, `bdr.remove_camo_pair` followed by `bdr.add_camo_pair` must be used, instead.

Synopsis

```
bdr.alter_camo_pair(node_group text, left_node text, right_node text,
                    require_raft bool)
```

Removing a CAMO pair

The function `bdr.remove_camo_pair()` removes a CAMO pairing of two nodes and disallows future use of CAMO transactions via `bdr.enable_camo` on those two nodes.

Synopsis

```
bdr.remove_camo_pair(node_group text, left_node text, right_node text)
```

Note

The names `left` and `right` have no special meaning.

CAMO partner connection status

The function `bdr.is_camo_partner_connected` allows checking the connection status of a CAMO partner node configured in Pair mode. There currently is no equivalent for CAMO used in combination with Eager Replication.

Synopsis

```
bdr.is_camo_partner_connected()
```

Return value

A boolean value indicating whether the CAMO partner is currently connected to a WAL sender 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 configured in Pair mode. Underneath, this is what's used to trigger 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 in a timely manner (i.e. before `bdr.global_commit_timeout` expires).

Note

Note that this function queries the past or current state. A positive return value is no indication for the CAMO partner being able to confirm future transactions.

Fetch the CAMO partner

This function shows the local node's CAMO partner (configured via Pair mode).

```
bdr.get_configured_camo_partner()
```

Wait for consumption of the apply queue from the CAMO partner

The function `bdr.wait_for_camo_partner_queue` is a wrapper of `bdr.wait_for_apply_queue` defaulting to query the CAMO partner node. It yields an error if the local node is not part of a CAMO pair.

Synopsis

```
bdr.wait_for_camo_partner_queue()
```

Transaction status between CAMO nodes

This function enables a wait for CAMO transactions to be fully resolved.

```
bdr.camo_transactions_resolved()
```

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.

With CAMO used in Pair mode, this function should only ever be used on a node that's part of a CAMO pair. In combination with Eager Replication, it may be used on all nodes.

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.

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.

Also note that the client must not call this function before attempting to commit on the origin, otherwise the transaction may be forced to be rolled back.

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 `on`, `remote_write`, `remote_commit_async`, or `remote_commit_flush`. See [Commit at Most Once Settings](#))
- `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, is visible on both nodes of the CAMO pair and will eventually be replicated to all other BDR nodes. No need for the client to retry it.
- `'aborted'::TEXT` - the transaction has been aborted and will not be replicated to any other BDR node. The client 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 must 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 client.

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. A proxy may freely distribute transactions to all nodes in the commit group (i.e. to both nodes of a CAMO pair or to all BDR nodes in case of Eager All Node Replication).

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 as an error to the client applying the CAMO protocol.

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.

HARP is the only proxy that supports all of the above requirements. PgBouncer and HAProxy can work with CAMO, but do not support CAMO's `remote_write` mode.

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 TABLE DETACH PARTITION CONCURRENTLY`
- `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 origin node, so in case of disconnection, the application should be coded to immediately request the transaction 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 longer than 15 minutes.

If the application forgets the global identifier assigned, for example as a result of a restart, there is no easy way to recover that. Therefore, it is recommended applications wait for outstanding transactions to terminate before shutting down.

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.

Changing the CAMO partners in a CAMO pair is not currently possible. It's only possible to add or remove a pair. Adding or removing a pair does not need a restart of Postgres or even a reload of the configuration.

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.

Also, CAMO does not currently work together with the Decoding Worker. Installations using CAMO must keep `enable_wal_decoder` disabled for the BDR node group using CAMO.

Legacy BDR synchronous replication uses a mechanism for transaction confirmation different from CAMO. The two are not compatible and must not be used together. Therefore, a CAMO partner must not be configured in `synchronous_standby_names`. Using synchronous replication to a non-BDR node acting as a physical standby is well possible.

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, which is before the transaction becomes visible on the origin node itself.

Performance Implications

CAMO extends Postgres' replication protocol by adding an additional message round trip at commit. Applications should expect a higher commit latency than with asynchronous replication, mostly determined by the round trip time between involved nodes. Increasing the number of concurrent sessions can help to increase parallelism to still obtain reasonable transaction throughput.

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.

2.15 AutoPartition

AutoPartition allows tables to grow easily to large sizes by automatic partitioning management. This utilizes the additional features of BDR such as low-conflict locking of creating and dropping partitions.

New partitions can be created regularly and then dropped when the data retention period expires.

BDR management is primarily accomplished via SQL-callable functions. All functions in BDR are exposed in the `bdr` schema. Unless you put it into your `search_path`, you will need to schema-qualify the name of each function.

Auto Creation of Partitions

`bdr.autopartition()` is used to create or alter the definition of automatic range partitioning for a table. If no definition exists, it will be created, otherwise later executions will alter the definition.

`bdr.autopartition()` does not lock the actual table, it only changes the definition of when and how new partition maintenance actions will take place.

`bdr.autopartition()` leverages the EDB Postgres Extended features to allow a partition to be attached or detached/dropped without locking the rest of the table, the feature to set a new tablespace while allowing SELECT queries.

An ERROR is raised if the table is not RANGE partitioned or a multi-column partition key is used.

A new partition is added for every `partition_increment` range of values, with lower and upper bound `partition_increment` apart. For tables with a partition key of type `timestamp` or `date`, the `partition_increment` must be a valid constant of type `interval`. For example, specifying `1 Day` will cause a new partition to be added each day, with partition bounds that are 1 day apart.

If the partition column is connected to a `timeshard` or `ksuuid` sequence, the `partition_increment` must be specified as type `interval`. Otherwise, if the partition key is integer or numeric, then the `partition_increment` must be a valid constant of the same datatype. For example, specifying '1000000' will cause new partitions to be added every 1 million values.

If the table has no existing partition, then the specified `partition_initial_lowerbound` is used as the lower bound for the first partition. If `partition_initial_lowerbound` is not specified, then the system tries to derive its value from the partition column type and the specified `partition_increment`. For example, if `partition_increment` is specified as `1 Day`, then `partition_initial_lowerbound` will be automatically set to CURRENT DATE. If `partition_increment` is specified as `1 Hour`, then `partition_initial_lowerbound` will be set to the current hour of the current date. The bounds for the subsequent partitions will be set using the `partition_increment` value.

The system always tries to have a certain minimum number of advance partitions. In order to decide whether to create new partitions or not, it uses the specified `partition_autocreate_expression`. This can be a SQL evaluable expression, which is evaluated every time a check is performed. For example, for a partitioned table on column type `date`, if `partition_autocreate_expression` is specified as `DATE_TRUNC('day', CURRENT_DATE)`, `partition_increment` is specified as `1 Day` and `minimum_advance_partitions` is specified as 2, then new partitions will be created until the upper bound of the last partition is less than `DATE_TRUNC('day', CURRENT_DATE) + '2 Days'::interval`.

The expression is evaluated each time the system checks for new partitions.

For a partitioned table on column type `integer`, the `partition_autocreate_expression` may be specified as `SELECT max(partcol) FROM schema.partitioned_table`. The system then regularly checks if the maximum value of the partitioned column is within the distance of `minimum_advance_partitions * partition_increment` of the last partition's upper bound. It is expected that the user creates an index on the `partcol` so that the query runs efficiently. If the `partition_autocreate_expression` is not specified for a partition table on column type `integer`, `smallint` or `bigint`, then the system will automatically set it to `max(partcol)`.

If the `data_retention_period` is set, partitions will be automatically dropped after this period. Partitions will be dropped at the same time as new partitions are added, to minimize locking. If not set, partitions must be dropped manually.

The `data_retention_period` parameter is only supported for timestamp (and related) based partitions. The period is calculated by considering the upper bound of the partition and the partition is either migrated to the secondary tablespace or dropped if either of the given period expires, relative to the upper bound.

By default, AutoPartition manages partitions globally. In other words, when a partition is created on one node, the same partition is also created on all other nodes in the cluster. So all partitions are consistent and guaranteed to be available. For this, AutoPartition makes use of Raft. This behaviour can be

changed by passing `managed_locally` as `true`. In that case, all partitions are managed locally on each node. This is useful for the case when the partitioned table is not a replicated table and hence it may not be necessary or even desirable to have all partitions on all nodes. For example, the built-in `bdr.conflict_history` table is not a replicated table, and is managed by AutoPartition locally. Each node creates partitions for this table locally and drops them once they are old enough.

Tables once marked as `managed_locally` cannot be later changed to be managed globally and vice versa.

Activities are performed only when the entry is marked `enabled = on`.

The user is not expected to manually create or drop partitions for tables managed by AutoPartition. Doing so can make the AutoPartition metadata inconsistent and could cause it to fail.

Configure AutoPartition

The `bdr.autopartition` function configures automatic partitioning of a table.

Synopsis

```
bdr.autopartition(relation regclass,
                  partition_increment
text,
                  partition_initial_lowerbound text DEFAULT NULL,
                  partition_autocreate_expression text DEFAULT
NULL,
                  minimum_advance_partitions integer DEFAULT
2,
                  maximum_advance_partitions integer DEFAULT
5,
                  data_retention_period interval DEFAULT
NULL,
                  managed_locally boolean DEFAULT false,
                  enabled boolean DEFAULT on);
```

Parameters

- `relation` - name or Oid of a table.
- `partition_increment` - interval or increment to next partition creation.
- `partition_initial_lowerbound` - if the table has no partition, then the first partition with this lower bound and `partition_increment` apart upper bound will be created.
- `partition_autocreate_expression` - is used to detect if it is time to create new partitions.
- `minimum_advance_partitions` - the system will attempt to always have at least `minimum_advance_partitions` partitions.
- `maximum_advance_partitions` - number of partitions to be created in a single go once the number of advance partitions falls below `minimum_advance_partitions`.
- `data_retention_period` - interval until older partitions are dropped, if defined. This must be greater than `migrate_after_period`.
- `managed_locally` - if true then the partitions will be managed locally.
- `enabled` - allows activity to be disabled/paused and later resumed/re-enabled.

Examples

Daily partitions, keep data for one month:

```
CREATE TABLE measurement
(
```

```
logdate date not null,
peaktemp int,
unitsales int
) PARTITION BY RANGE (logdate);

bdr.autopartition('measurement', '1 day', data_retention_period := '30
days');
```

Create 5 advance partitions when there are only 2 more partitions remaining (each partition can hold 1 billion orders):

```
bdr.autopartition('Orders', '1000000000',
    partition_initial_lowerbound := '0',
    minimum_advance_partitions :=
2,
    maximum_advance_partitions :=
5
);
```

Create One AutoPartition

Use `bdr.autopartition_create_partition()` to create a standalone AutoPartition on the parent table.

Synopsis

```
bdr.autopartition_create_partition(relname regclass,
                                partname
name,                            lowerb
text,                            upperb
text,                            nodes oid[]);
```

Parameters

- `relname` - Name or Oid of the parent table to attach to
- `partname` - Name of the new AutoPartition
- `lowerb` - The lower bound of the partition
- `upperb` - The upper bound of the partition
- `nodes` - List of nodes that the new partition resides on

Stopping Auto-Creation of Partitions

Use `bdr.drop_autopartition()` to drop the auto-partitioning rule for the given relation. All pending work items for the relation are deleted and no new work items are created.

```
bdr.drop_autopartition(relation regclass);
```

Parameters

- `relation` - name or Oid of a table

Drop one AutoPartition

Use `bdr.autopartition_drop_partition` once a BDR AutoPartition table has been made, as this function can specify single partitions to drop. If the partitioned table has successfully been dropped, the function will return true.

Synopsis

```
bdr.autopartition_drop_partition(relname regclass)
```

Parameters

- `relname` - The name of the partitioned table to be dropped

Notes

This will place a DDL lock on the parent table, before using DROP TABLE on the chosen partition table.

Wait for Partition Creation

Use `bdr.autopartition_wait_for_partitions()` to wait for the creation of partitions on the local node. The function takes the partitioned table name and a partition key column value and waits until the partition that holds that value is created.

The function only waits for the partitions to be created locally. It does not guarantee that the partitions also exists on the remote nodes.

In order to wait for the partition to be created on all BDR nodes, use the `bdr.autopartition_wait_for_partitions_on_all_nodes()` function. This function internally checks local as well as all remote nodes and waits until the partition is created everywhere.

Synopsis

```
bdr.autopartition_wait_for_partitions(relation regclass, text bound);
```

Parameters

- `relation` - name or Oid of a table
- `bound` - partition key column value.

Synopsis

```
bdr.autopartition_wait_for_partitions_on_all_nodes(relation regclass, text bound);
```

Parameters

- `relation` - name or Oid of a table.

- `bound` - partition key column value.

Find Partition

Use the `bdr.autopartition_find_partition()` function to find the partition for the given partition key value. If partition to hold that value does not exist, then the function returns NULL. Otherwise OID of the partition is returned.

Synopsis

```
bdr.autopartition_find_partition(relname regclass, searchkey
text);
```

Parameters

- `relname` - name of the partitioned table.
- `searchkey` - partition key value to search.

Enable/Disable AutoPartitioning

Use `bdr.autopartition_enable()` to enable AutoPartitioning on the given table. If AutoPartitioning is already enabled, then it will be a no-op. Similarly, use `bdr.autopartition_disable()` to disable AutoPartitioning on the given table.

Synopsis

```
bdr.autopartition_enable(relname regclass);
```

Parameters

- `relname` - name of the relation to enable AutoPartitioning.

Synopsis

```
bdr.autopartition_disable(relname regclass);
```

Parameters

- `relname` - name of the relation to disable AutoPartitioning.

Synopsis

```
bdr.autopartition_get_last_completed_workitem();
```


Return the `id` of the last workitem successfully completed on all nodes in the cluster.

Check AutoPartition Workers

From using the `bdr.autopartition_work_queue_check_status` function, you can see the status of the background workers that are doing their job to maintain AutoPartitions.

The workers can be seen through these views: `autopartition_work_queue_local_status`
`autopartition_work_queue_global_status`

Synopsis

```
bdr.autopartition_work_queue_check_status(workid
bigint
                                     local boolean DEFAULT false);
```

Parameters

- `workid` - The key of the AutoPartition worker
- `local` - Check the local status only

Notes

AutoPartition workers are ALWAYS running in the background, even before the `bdr.autopartition` function is called for the first time. If an invalid worker ID is used, the function will return 'unknown'. 'In-progress' is the typical status.

2.16 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.

Note

This feature is currently only available on EDB Postgres Extended.

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.

2.17 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.

Using Replication Sets

Additional replication sets can be created using `create_replication_set()`, specifying whether to include insert, update, delete or truncate actions. An option exists to add existing tables to the set automatically, and a second option defines whether to add tables automatically when they are created.

You may also define manually which tables are added or removed from a replication set.

Tables included in the replication set will be maintained when the node joins the cluster and afterwards.

Once the node is joined, you may still remove tables from the replication set, but adding new tables must be done via a resync operation.

By default, a newly defined replication set does not replicate DDL or BDR administration function calls. Use the `replication_set_add_ddl_filter` to define which commands will be replicated.

BDR creates replication set definitions on all nodes. Each node can then be defined to publish and/or subscribe to each replication set using `alter_node_replication_sets`.

Functions exist to alter these definitions later, or to drop the replication set.

Note

Do not use the default replication set for selective replication. You should not drop or modify the default replication set on any of the BDR nodes in the cluster as it is also used by default for DDL replication and administration function calls.

Behavior of Partitioned Tables

BDR supports partitioned tables transparently, meaning 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 if the 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.

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 a member of replication set `s2`, but the table referenced by the foreign key `t2_x_fkey` is not.

Note

The `TRUNCATE CASCADE` command takes 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.

Note that, with the exception of `bdr.alter_node_replication_sets`, the 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`

This function 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 that 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`

This function modifies the 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 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.

bdr.drop_replication_set

This function 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

This function 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 is 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 is part of the BDR group, but also on a node that has not 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.

The replication sets that the node subscribes to after this call should be published by the other node/s for actually replicating the changes from those nodes to the node where this function is executed.

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) are 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`

This function 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,
```

```

NULL,
        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) then 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) then 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 that the replication is affected by the `ddl_filters` configuration.

The function will take a `DML` global lock on the relation that is 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`

This function 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 those replication sets that the current node publishes and subscribes from:

```
SELECT
  node_id,
  node_name,
  COALESCE(
    pub_repsets,
    pub_repsets
  ) AS
  pub_repsets,
  COALESCE(
    sub_repsets,
    sub_repsets
  ) AS
  sub_repsets
FROM bdr.local_node_summary;
```

This produces output like this:

```
 node_id | node_name | pub_repsets |
sub_repsets
-----+-----+-----+
 1834550102 | s01db01 | {bdrglobal,bdrs01} |
{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($$
      SELECT
node_id,
      node_name,
      COALESCE(
        pub_repsets,
pub_repsets
      ) AS
pub_repsets,
      COALESCE(
        sub_repsets,
sub_repsets
      ) AS
sub_repsets
    FROM bdr.local_node_summary;
    $$)::jsonb
  ) AS
  j
)
SELECT j->'response'->'command_tuples'->0->>'node_id' AS
node_id,
      j->'response'->'command_tuples'->0->>'node_name' AS
node_name,
      j->'response'->'command_tuples'->0->>'pub_repsets' AS
pub_repsets,
      j->'response'->'command_tuples'->0->>'sub_repsets' AS
sub_repsets
FROM node_repsets;
```

This will show, for example:

node_id	node_name	pub_repsets	sub_repsets
933864801	s02db01	{bdrglobal,bdrs02}	{bdrglobal,bdrs02}
1834550102	s01db01	{bdrglobal,bdrs01}	{bdrglobal,bdrs01}
3898940082	s01db02	{bdrglobal,bdrs01}	{bdrglobal,bdrs01}
1102086297	s02db02	{bdrglobal,bdrs02}	{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 is 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`

This function 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 then 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 the same replication mechanism as **DDL** statements. This means that the replication is affected by the `ddl_filters` configuration. Please note - 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`

This function removes the 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 then 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.

2.18 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.

Note

This feature is currently only available on EDB Postgres Extended and EDB Postgres Advanced.

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 = 'D'` 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 one 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. Technical Support 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 lead to unique constraint violations error due to a 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 behaviour 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 a matching row in a table.
- Transform triggers are called before partition table routing occurs.
- Transform triggers 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 text; 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 the `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_trigger()` 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**

This function returns the 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

This function returns the 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

This function returns the 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

This function 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

This function returns the current conflict type if called inside a conflict trigger, or `NULL` otherwise.

See [Conflict Types](conflicts.md#List of Conflict Types) for possible return values of this function.

Synopsis

```
bdr.trigger_get_conflict_type()
```

bdr.trigger_get_origin_node_id

This function 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).

bdr.ri_fkey_on_del_trigger

When called as a BEFORE trigger, this function 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`

This function 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; specifies the 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 on triggers](#).

`bdr.create_transform_trigger`

This function 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 the `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

This function 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 behaviour 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
timestampz;
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;
$$;

```

The example below shows 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;
$$;

```

2.19 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 restore would effectively reset all sequence kinds to the value of `bdr.default_sequence_kind` at time of restore.

To create a post-restore script to reset the precise sequence kind for each sequence, you might want to use an SQL script like this:

```

SELECT 'SELECT bdr.alter_sequence_set_kind('' ||
    nspname || '.' || relname || ''', '' || seqkind || ''');'
FROM bdr.sequences
WHERE seqkind != 'local';

```

Note that if `pg_dump` is run using `bdr.crdt_raw_value = on` then the dump can only be reloaded with `bdr.crdt_raw_value = on`.

Technical Support recommends the use of physical backup techniques for backup and recovery of BDR.

Physical Backup

Physical backups of a node in a EDB Postgres Distributed 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 BDR 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 EDB Postgres Distributed 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 EDB Postgres Distributed 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".

Once we have reached the defined PIT, a later one may also be set to allow the recovery to continue, as needed.

After the desired stopping point has been reached, if the recovered server will be promoted, shut it down first and move the LSN forwards using `pg_resetwal` to an LSN value higher than used on any timeline on this server. This ensures that there will be no duplicate LSNs produced by logical decoding.

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 `postgresql.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

This feature is only available on EDB Postgres Extended and Barman does not currently 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.

EDB Postgres Distributed 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 EDB Postgres Distributed 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 `postgresql.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.

pg_dump restore

The restore procedure using `pg_dump` is somehow simpler (but not as consistent as physical backups because it only makes a snapshot of a given moment in time). The dump only takes personal data and not catalog data (BDR).

Creating a new cluster from a dump needs only a new Postgresql instance where to restore normally the dump. The result is a plain postgres instance with all personal data.

No catalog changes are restored from the original BDR node. You need to manually create the EDB Postgres Distributed cluster from this instance. Create the BDR extension, create the node, create the bdr group, etc. and eventually join other nodes to this instance.

There is only one caveat to keep in mind when restoring a BDR dump. BDR creates two policies on the `bdr.conflict_history` table. PostgreSQL normal behavior does not allow to tie policies to an extension so these will be dumped together with your data. These policies, unless removed from the dump, will drag also the BDR extension and grant some permissions.

You need to manually remove those statements from the dump prior restoring it.

The statements you need to delete from the dump are the following:

```
CREATE EXTENSION IF NOT EXISTS bdr WITH SCHEMA
pg_catalog;
COMMENT ON EXTENSION bdr IS 'Bi-Directional Replication for
PostgreSQL';
CREATE POLICY conflict_filter ON bdr.conflict_history FOR SELECT USING ((( SELECT
c.relowner
FROM pg_class
c
WHERE (c.oid = conflict_history.reloid)) = ( SELECT
pg_roles.oid
FROM
pg_roles
WHERE (pg_roles.rolname = CURRENT_USER))) AND ( SELECT (NOT (c.relrowsecurity AND
c.relforcerowsecurity))
FROM pg_class
c
WHERE (c.oid =
conflict_history.reloid)))));
ALTER TABLE bdr.conflict_history ENABLE ROW LEVEL SECURITY;
CREATE POLICY conflict_read_all ON bdr.conflict_history FOR SELECT TO bdr_read_all_conflicts USING
(true);
REVOKE ALL ON TABLE bdr.conflict_history FROM postgres;
REVOKE SELECT ON TABLE bdr.conflict_history FROM PUBLIC;
GRANT ALL ON TABLE bdr.conflict_history TO
bdr_superuser;
```

```
GRANT SELECT ON TABLE bdr.conflict_history TO PUBLIC;
```

Note

In the example, `postgres` is the `initdb` user, if you create the PostgreSQL instance with a different user, then that user will be shown instead in the dump, related to the `REVOKE` on `bdr.conflict_history` table.

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_drop_replication_slot(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.

2.20 Upgrading

In this chapter we discuss upgrading software on a EDB Postgres Distributed cluster and how to minimize downtime for applications during the upgrade.

Overview

EDB Postgres Distributed cluster has two sets of software, the underlying PostgreSQL software or some flavor of it and the PGLocal/BDR software. We will discuss upgrading either or both of these softwares versions to their supported major releases.

To upgrade a EDB Postgres Distributed cluster, the following steps need to be performed on each node:

- plan the upgrade
- prepare for the upgrade
- upgrade the server software
- restart Postgres
- check and validate the upgrade

Upgrade Planning

While BDR 3.6 release supports PostgreSQL 10 and 11 major versions, BDR 3.7 supports PostgreSQL 11, 12 and 13. Please refer to the [Compatibility matrix](#) for the full list compatible software. Since BDR 3.7 supports newer PostgreSQL releases, while upgrading from BDR 3.6 to BDR 3.7, it's also possible to upgrade the newer PostgreSQL releases with minimum or no application downtime.

There are broadly two ways to upgrade the BDR version.

- Upgrading one node at a time to the newer BDR version.
- Joining a new node running a newer version of the BDR software and then optionally drop one of the old nodes.

If you are only interested in upgrading the BDR software, any of the two methods can be used. But if you also want to upgrade the PostgreSQL version, then the second method must be used.

Rolling Server Software Upgrades

A rolling upgrade is the process where the below [Server Software Upgrade](#) is performed on each node in the BDR Group one after another, while keeping the replication working.

An upgrade to 3.7 is only supported from 3.6, using a specific minimum maintenance release (e.g. 3.6.25). Please consult the Release Notes for the actual required minimum version. So if a node is running with an older 3.6 release, it must first be upgraded to the minimum and can only then be upgraded to 3.7.

Just as with a single-node database, it's possible to stop all nodes, perform the upgrade on all nodes and only then restart the entire cluster. This strategy of upgrading all nodes at the same time avoids running with mixed BDR versions and therefore is the simplest, but obviously incurs some downtime.

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 3.6.25, while nodeB and nodeC will have 3.7.8. In this state, the replication and group management will use the protocol and features from the oldest version (3.6.25 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 EDB Postgres Distributed 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.

A 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 Technical 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 Technical 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 Technical 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.

Rolling Upgrade Using Node Join

The other method of upgrading BDR software, along with or without upgrading the underlying PostgreSQL major version, is to join a new node to the cluster and later drop one of the existing nodes running the older version of the software. Even with this method, some features that are available only in the newer version of the software may remain unavailable until all nodes are finally upgraded to the newer versions.

A new node running this release of BDR 3.7.8 can join a 3.6 cluster, where each node in the cluster is running the latest 3.6.x version of BDR. The joining node may run any of the supported PostgreSQL versions 11-13, but you must not mix the Standard and Enterprise editions. If the older cluster is running a Standard Edition then it's recommended that the new joining node should also run a Standard Edition. Similarly, if the old cluster is running Enterprise Edition, the new joining node should also run the Enterprise Edition.

Care must be taken to not use features that are available only in the newer PostgreSQL versions 12-13, until all nodes are upgraded to the newer and same release of PostgreSQL. This is especially true for any new DDL syntax that may have been added to newer release of PostgreSQL.

Note that `bdr_init_physical` makes a byte-by-byte copy of the source node. So it cannot be used while upgrading from one major PostgreSQL version to another. In fact, currently `bdr_init_physical` requires that even BDR version of the source and the joining node is exactly the same. So it cannot be used for rolling upgrades via joining a new node method. In all such cases, a logical join must be used.

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 to 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.

Upgrade Preparation

BDR 3.7 contains several changes that may affect compatibility with previous releases. These may affect the Postgres configuration, deployment scripts as well as applications using BDR. We recommend to consider and possibly adjust in advance of the upgrade.

Node Management

The `bdr.create_node_group()` function has seen a number of changes:

- It is now possible to create sub-groups, resulting in a tree-of-groups structure of the whole EDB Postgres Distributed cluster. Monitoring views were updated accordingly.
- The deprecated parameters `insert_to_update`, `update_to_insert`, `ignore_redundant_updates`, `check_full_tuple` and `apply_delay` were removed. Use `bdr.alter_node_set_conflict_resolver()` instead of `insert_to_update`, `update_to_insert`. The `check_full_tuple` is no longer needed as it is handled automatically based on table conflict detection configuration.

Conflicts

The configuration of conflict resolution and logging is now copied from join source node to the newly joining node, rather than using defaults on the new node.

The default conflict resolution for some of the conflict types was changed. See (conflicts.md#default-conflict-resolvers) for the new defaults.

The conflict logging interfaces have changed from `bdr.alter_node_add_log_config` and `bdr.alter_node_remove_log_config` to `bdr.alter_node_set_log_config`.

The default conflict logging table is now named `bdr.conflict_history` and the old `bdr.apply_log` no longer exists. The new table is partitioned using the new Autopartition feature of BDR 3.7.

All conflicts are now logged by default to both log file and the conflict table.

Deprecated functions `bdr.row_version_tracking_enable()` and `bdr.row_version_tracking_disable()` were removed. Use `bdr.alter_table_conflict_detection()` instead.

Some of the configuration for conflict handling is no longer stored in `pglogical` schema. Any diagnostic queries that were using the `pglogical` tables directly will have to switch to appropriate tables in `bdr` schema. Queries using `bdr.node_group`, `bdr.local_node_summary`, `bdr.local_node_summary` or `bdr.node_local_info` will need to use the new columns `sub_repsets` and `pub_repsets` instead of `replication_sets`.

Removed Or Renamed Settings (GUCs)

The setting `report_transaction_id` has been removed and is no longer known to Postgres. It had been deprecated in the 3.6.x branch already and the underlying functionality is enabled automatically when needed, instead. So it's safe to remove `report_transaction_id` from your configuration or reset it via `ALTER SYSTEM` even on 3.6.22 (and newer). Otherwise, Postgres refuses to start after the upgrade and will report an "unrecognized configuration parameter".

The GUC to enable CAMO has moved from Postgres to BDR and got renamed from `pg2q.enable_camo` to `bdr.enable_camo`.

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.

Restart Postgres

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.

Upgrade Check and Validation

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.

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.

Similar to the upgrade of BDR itself, there are two approaches to upgrading the application schema. The simpler option is to stop all applications affected, preform the schema upgrade and restart the application upgraded to use the new schema variant. Again, this imposes some downtime.

To eliminate this downtime, BDR offers ways to perform a rolling application schema upgrade as documented in the following section.

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.

2.20.1 Supported BDR upgrade paths

Upgrading within version 3.7

BDR 3.7 does not follow semantic versioning. Instead, it uses 2 position majors similar to PostgreSQL releases prior to version 10 (for example, 9.6). BDR 3.7 also considered releases prior to 3.7.9 as beta releases and they could include breaking changes. BDR 3.7.9 was declared a Long Term Support version under the previous versioning nomenclature. All releases 3.7.9 and later maintain backward compatibility allowing upgrades to later versions.

Upgrading from version 3.6 to version 3.7

We recommend you upgrade your cluster to the latest version of 3.6 before upgrading your cluster to 3.7. You will then be able to upgrade to a version of 3.7 released on the same date or later.

2.21 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 EDB Postgres Distributed 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 bdr.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';
```

2.22 Catalogs and Views

This section contains a listing of system catalogs and views used by BDR in alphabetical order.

User-Visible Catalogs and Views

`bdr.conflict_history`

This table is the default table where conflicts are logged. The table is RANGE partitioned on column `local_time` and is managed by Autopartition. The default data retention period is 30 days.

Access to this table is possible by any table owner, who may see all conflicts for the tables they own, restricted by row-level security.

For further details see [Logging Conflicts to a Table](#).

`bdr.conflict_history` Columns

Name	Type	Description
<code>sub_id</code>	oid	which subscription produced this conflict; can be joined to <code>bdr.subscription</code> table
<code>local_xid</code>	xid	local transaction of the replication process at the time of conflict
<code>local_lsn</code>	pg_lsn	local transaction of the replication process at the time of conflict
<code>local_time</code>	timestamp with time zone	local time of the conflict
<code>remote_xid</code>	xid	transaction which produced the conflicting change on the remote node (an origin)

Name	Type	Description
remote_commit_lsn	pg_lsn	commit lsn of the transaction which produced the conflicting change on the remote node (an origin)
remote_commit_time	timestamp with time zone	commit timestamp of the transaction which produced the conflicting change on the remote node (an origin)
conflict_type	text	detected type of the conflict (see [List of Conflict Types])
conflict_resolution	text	conflict resolution chosen (see [List of Conflict Resolutions])
conflict_index	regclass	conflicting index (only valid if the index wasn't dropped since)
reloid	oid	conflicting relation (only valid if the index wasn't dropped since)
nspname	text	name of the schema for the relation on which the conflict has occurred at the time of conflict (does not follow renames)
relname	text	name of the relation on which the conflict has occurred at the time of conflict (does not follow renames)
key_tuple	json	json representation of the key used for matching the row
remote_tuple	json	json representation of an incoming conflicting row
local_tuple	json	json representation of the local conflicting row
apply_tuple	json	json representation of the resulting (the one that has been applied) row
local_tuple_xmin	xid	transaction which produced the local conflicting row (if <code>local_tuple</code> is set and the row is not frozen)
local_tuple_node_id	oid	node which produced the local conflicting row (if <code>local_tuple</code> is set and the row is not frozen)
local_tuple_commit_time	timestamp with time zone	last known change timestamp of the local conflicting row (if <code>local_tuple</code> is set and the row is not frozen)

`bdr.conflict_history_summary`

A view containing user-readable details on row conflict.

`bdr.conflict_history_summary` Columns

Name	Type	Description
schema	text	Name of the schema
table	text	Name of the table
local_time	timestamp with time zone	local time of the conflict
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.consensus_kv_data`

A persistent storage for the internal Raft based KV store used by `bdr.consensus_kv_store()` and `bdr.consensus_kv_fetch()` interfaces.

`bdr.consensus_kv_data` Columns

Name	Type	Description
kv_key	text	Unique key
kv_val	json	Arbitrary value in json format
kv_create_ts	timestamptz	Last write timestamp
kv_ttl	int	Time to live for the value in milliseconds
kv_expire_ts	timestamptz	Expiration timestamp (<code>kv_create_ts + kv_ttl</code>)

`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

Note

This catalog is only present when bdr-enterprise extension is installed.

`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

Note

This catalog is only present when bdr-enterprise extension is installed.

`bdr.ddl_replication`

This view lists DDL replication configuration as set up by current [DDL filters](#).

`bdr.ddl_replication` Columns

Name	Type	Description
<code>set_ddl_name</code>	name	Name of DDL filter
<code>set_ddl_tag</code>	text	Which command tags it applies on (regular expression)
<code>set_ddl_role</code>	text	Which roles it applies to (regular expression)
<code>set_name</code>	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
<code>log_index</code>	int8	Id of the journal entry
<code>term</code>	int8	Raft term
<code>origin</code>	oid	Id of node where the request originated
<code>req_id</code>	int8	Id for the request
<code>req_payload</code>	bytea	Payload for the request
<code>trace_context</code>	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
<code>log_index</code>	int8	Id of the journal entry
<code>term</code>	int8	Raft term
<code>request_id</code>	int8	Id of the request
<code>origin_id</code>	oid	Id of the node where the request originated
<code>req_payload</code>	bytea	Payload of the request
<code>origin_node_name</code>	name	Name of the node where the request originated

Name	Type	Description
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
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
groupid	oid	OID of the top level group (for Advisory locks)
key1	integer	First 32-bit key or lower order 32-bits of 64-bit key (for Advisory locks)
key2	integer	Second 32-bit key or higher order 32-bits of 64-bit key (for Advisory locks)
key_is_bigint	boolean	True if 64-bit integer key is used (for Advisory locks)

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
<code>origin_node_id</code>	oid	The OID of the node where the global lock has originated
<code>origin_node_name</code>	name	Name of the node where the global lock has originated
<code>lock_type</code>	text	Type of the lock (DDL or DML)
<code>relation</code>	text	Locked relation name (for DML locks) or keys (for advisory locks)
<code>pid</code>	int4	PID of the process holding the lock
<code>acquire_stage</code>	text	Internal state of the lock acquisition process
<code>waiters</code>	int4	List of backends waiting for the same global lock
<code>global_lock_request_time</code>	timestamptz	Time this global lock acquire was initiated by origin node
<code>local_lock_request_time</code>	timestamptz	Time the local node started trying to acquire the local-lock
<code>last_state_change_time</code>	timestamptz	Time <code>acquire_stage</code> last changed

Column details:

- `relation`: For DML locks, `relation` shows the relation on which the DML lock is acquired. For global advisory locks, `relation` column actually shows the two 32-bit integers or one 64-bit integer on which the lock is acquired.
- `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
<code>log_index</code>	<code>int8</code>	Id of the journal entry
<code>log_term</code>	<code>int8</code>	Raft term
<code>snapshot</code>	<code>bytea</code>	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
<code>node_id</code>	<code>oid</code>	Id of the node
<code>current_term</code>	<code>int8</code>	Raft term
<code>apply_index</code>	<code>int8</code>	Raft apply index
<code>voted_for</code>	<code>oid</code>	Vote cast by this node in this term
<code>last_known_leader</code>	<code>oid</code>	<code>node_id</code> 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.network_path_info`

A catalog view that stores user-defined information on network costs between node locations.

`bdr.network_path_info` Columns

Name	Type	Description
<code>node_group_name</code>	<code>name</code>	Name of the BDR group
<code>node_region1</code>	<code>text</code>	Node region name, from <code>bdr.node_location</code>

Name	Type	Description
node_region2	text	Node region name, from bdr.node_location
node_location1	text	Node location name, from bdr.node_location
node_location2	text	Node location name, from bdr.node_location
network_cost	numeric	Node location name, from bdr.node_location

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_ranges	int[]	Supported protocol version ranges by the node
reserved	int2	Reserved field for compatibility purposes
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 EDB Postgres Distributed cluster.

`bdr.node_conflict_resolvers`

Currently configured conflict resolution for all known conflict types.

`bdr.node_conflict_resolvers` Columns

Name	Type	Description
<code>conflict_type</code>	text	Type of the conflict
<code>conflict_resolver</code>	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
<code>node_group_id</code>	oid	ID of the node group
<code>node_group_name</code>	name	Name of the node group
<code>node_group_default_repset</code>	oid	Default replication set for this node group
<code>node_group_default_repset_ext</code>	oid	Default replication set for this node group
<code>node_group_parent_id</code>	oid	ID of parent group (0 if this is a root group)
<code>node_group_flags</code>	int	The group flags
<code>node_group_uuid</code>	uuid	The uuid of the group
<code>node_group_apply_delay</code>	interval	How long a subscriber waits before applying changes from the provider
<code>node_group_check_constraints</code>	bool	Whether the apply process should check constraints when applying data
<code>node_group_num_writers</code>	int	Number of writers to use for subscriptions backing this node group
<code>node_group_enable_wal_decoder</code>	bool	Whether the group has <code>enable_wal_decoder</code> set

`bdr.node_group_replication_sets`

A view showing default replication sets create for BDR groups. See also `bdr.replication_sets`.

`bdr.node_group_replication_sets` Columns

Name	Type	Description
<code>node_group_name</code>	name	Name of the BDR group
<code>def_repset</code>	name	Name of the default repset
<code>def_repset_ops</code>	text[]	Actions replicated by the default repset
<code>def_repset_ext</code>	name	Name of the default "external" repset (usually same as <code>def_repset</code>)
<code>def_repset_ext_ops</code>	text[]	Actions replicated by the default "external" repset (usually same as <code>def_repset_ops</code>)

`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
<code>node_id</code>	oid	The OID of the node (including the local node)
<code>applied_state</code>	oid	Internal id of the node state
<code>ddl_epoch</code>	int8	Last epoch number processed by the node
<code>pub_repsets</code>	text[]	List of replication sets published by the node (only for the local node)
<code>slot_name</code>	name	Name of the slot used to connect to that node (NULL for the local node)
<code>sub_repsets</code>	text[]	List of replication sets subscribed by the node (only for the local node)

`bdr.node_location`

A catalog view that stores user-defined information on node locations.

`bdr.node_location` Columns

Name	Type	Description
<code>node_group_name</code>	name	Name of the BDR group
<code>node_id</code>	oid	Id of the node
<code>node_region</code>	text	User supplied region name
<code>node_location</code>	text	User supplied location name

`bdr.node_log_config`

A catalog view that stores information on the conflict logging configurations.

`bdr.node_log_config` Columns

Name	Description
<code>log_name</code>	name of the logging configuration
<code>log_to_file</code>	whether it logs to the server log file
<code>log_to_table</code>	whether it logs to a table, and which table is the target
<code>log_conflict_type</code>	which conflict types it logs, if NULL means all
<code>log_conflict_res</code>	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
<code>node_id</code>	<code>oid</code>	The OID of the originating node which reported this position info
<code>peer_node_id</code>	<code>oid</code>	The OID of the node's peer (remote node) for which this position info was reported
<code>last_update_sent_time</code>	<code>timestamptz</code>	The time at which the report was sent by the originating node
<code>last_update_recv_time</code>	<code>timestamptz</code>	The time at which the report was received by the local server
<code>last_update_node_lsn</code>	<code>pg_lsn</code>	LSN on the originating node at the time of the report
<code>peer_position</code>	<code>pg_lsn</code>	Latest LSN of the node's peer seen by the originating node
<code>peer_replay_time</code>	<code>timestamptz</code>	Latest replay time of peer seen by the reporting node
<code>last_update_horizon_xid</code>	<code>oid</code>	Internal resolution horizon: all lower xids are known resolved on the reporting node
<code>last_update_horizon_lsn</code>	<code>pg_lsn</code>	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
<code>origin_node_id</code>	<code>oid</code>	OID of the node where the transaction executed
<code>origin_xid</code>	<code>oid</code>	Transaction Id on the remote origin node
<code>decision</code>	<code>char</code>	'c' for commit, 'a' for abort
<code>local_xid</code>	<code>xid</code>	Transaction Id on the local node
<code>commit_ts</code>	<code>timestamptz</code>	commit timestamp of the transaction
<code>decision_ts</code>	<code>timestamptz</code>	decision time

Note

This catalog is only present when bdr-enterprise extension is installed.

`bdr.node_replication_rates`

This view contains information about outgoing replication activity from a given node

`bdr.node_replication_rates` Columns

Column	Type	Description
peer_node_id	oid	The OID of node's peer (remote node) for which this info was reported
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_bytes	int8	Bytes difference between replay_lsn and current WAL write position on origin
replay_lag_size	text	Human-readable bytes difference between replay_lsn and current WAL write position
apply_rate	bigint	LSNs being applied per second at the peer node
catchup_interval	interval	Approximate time required for the peer node to catchup to all the changes that are 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`.

Note

This catalog is only present when bdr-enterprise extension is installed.

`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)
is_group_slot	boolean	True if the slot is the node-group crash recovery slot for this node (see ["Group Replication Slot"] (nodes.md#Group Replication Slot))
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

Name	Type	Description
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 <code>synchronous_standby_names</code>)
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	timestampz	When the connection started
state	text	State of the replication (catchup, streaming, ...) or 'disconnected' if offline
sent_lsn	pg_lsn	Latest sent position
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

Name	Type	Description
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
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
pub_repsets	text[]	Deprecated column, always NULL, will be removed in 4.0
sub_repsets	text[]	Deprecated column, always NULL, will be removed in 4.0
node_id	oid	The OID of the node
node_group_id	oid	The OID of the BDR node group
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

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	timestampz	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

Name	Type	Description
schema_changes_descr	text	The object changed
schema_changes_addrnames	text[]	Location of schema change

bdr.sequence_alloc

A view to see the allocation details for gallo sequences.

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	timestampz	Last sequence allocated

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	timestampz	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

Name	Type	Description
seq_allocated_up_to	bigint	
seq_nallocs	bigint	
seq_last_alloc	timestampz	Last sequence allocated

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', 'galloc')

bdr.stat_activity

Dynamic activity for each backend or worker process.

This contains the same information as `pg_stat_activity`, except `wait_event` is set correctly when the wait relates to BDR.

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

Column	Type	Description
blk_read_time	double precision	Total time spent reading blocks for the relation, in milliseconds (if <code>track_io_timing</code> is enabled, otherwise zero)
blk_write_time	double precision	Total time spent writing blocks for the relation, in milliseconds (if <code>track_io_timing</code> is enabled, otherwise zero)
lock_acquire_time	double precision	Total time spent acquiring locks on the relation, in milliseconds (if <code>pglogical.track_apply_lock_timing</code> 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
nabort	bigint	Number of aborts writer did for this subscription
nerror	bigint	Number of errors writer has hit for this subscription
nskippedtx	bigint	Number of transactions skipped by writer for this subscription (due to <code>skip_transaction</code> conflict resolver)
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
ndeadlocks	bigint	Number of errors that were caused by deadlocks
nretries	bigint	Number of retries the writer did (without going for full restart/reconnect)
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 <code>track_io_timing</code> is enabled, otherwise zero)
blk_write_time	double precision	Total time the subscription spent writing blocks, in milliseconds (if <code>track_io_timing</code> is enabled, otherwise zero)
connect_time	timestamp with time zone	Time when the current upstream connection was established, NULL if not connected
last_disconnect_time	timestamp with time zone	Time when the last upstream connection was dropped
start_lsn	pg_lsn	LSN from which this subscription requested to start replication from the upstream
retries_at_same_lsn	bigint	Number of attempts the subscription was restarted from the same LSN value
curr_ncommit	bigint	Number of commits this subscription did after the current connection was established

`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
<code>pgl_subscription_id</code>	oid	Subscription in pglogical
<code>nodegroup_id</code>	oid	Id of nodegroup
<code>origin_node_id</code>	oid	Id of origin node
<code>target_node_id</code>	oid	Id of target node
<code>subscription_mode</code>	char	Mode of subscription
<code>source_node_id</code>	oid	Id of source node
<code>ddl_epoch</code>	int8	DDL epoch

`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
<code>node_group_name</code>	name	Name of the BDR group the node is part of
<code>sub_name</code>	name	Name of the subscription
<code>origin_name</code>	name	Name of the origin node
<code>target_name</code>	name	Name of the target node (normally local node)
<code>sub_enabled</code>	bool	Is the subscription enabled
<code>sub_slot_name</code>	name	Slot name on the origin node used by this subscription
<code>sub_replication_sets</code>	text[]	Replication sets subscribed
<code>sub_forward_origins</code>	text[]	Does the subscription accept changes forwarded from other nodes besides the origin
<code>sub_apply_delay</code>	interval	Delay transactions by this much compared to the origin
<code>sub_origin_name</code>	name	Replication origin name used by this subscription
<code>bdr_subscription_mode</code>	char	Subscription mode
<code>subscription_status</code>	text	Status of the subscription worker
<code>node_group_id</code>	oid	The OID of the BDR group the node is part of
<code>sub_id</code>	oid	The OID of the subscription
<code>origin_id</code>	oid	The OID of the origin node
<code>target_id</code>	oid	The OID of the target node
<code>receive_lsn</code>	pg_lsn	Latest LSN of any change or message received (this can go backwards in case of restarts)
<code>receive_commit_lsn</code>	pg_lsn	Latest LSN of last COMMIT received (this can go backwards in case of restarts)
<code>last_xact_replay_lsn</code>	pg_lsn	LSN of last transaction replayed on this subscription
<code>last_xact_flush_lsn</code>	timestamptz	LSN of last transaction replayed on this subscription that's flushed durably to disk
<code>last_xact_replay_timestamp</code>	timestamptz	Timestamp of last transaction replayed on this subscription

`bdr.replication_status`

This view shows incoming replication status between the local node and all other nodes in the EDB Postgres Distributed cluster. If this is a logical standby node, then only the status for its upstream node is shown. Similarly, replication status is not shown for subscriber-only nodes since they never send replication changes to other nodes.

`bdr.replication_status` Columns

Column	Type	Description
<code>node_id</code>	oid	OID of the local node
<code>node_name</code>	name	Name of the local node
<code>origin_node_id</code>	oid	OID of the origin node
<code>origin_node_name</code>	name	Name of the origin node
<code>sub_id</code>	oid	OID of the subscription for this origin node
<code>sub_name</code>	name	Name of the subscription for this origin node
<code>connected</code>	boolean	Is this node connected to the origin node?
<code>replication_blocked</code>	boolean	Is the replication currently blocked for this origin?
<code>connect_time</code>	timestamp with time zone	Time when the current connection was established
<code>disconnect_time</code>	timestamp with time zone	Time when the last connection was dropped
<code>uptime</code>	interval	Duration since the current connection is active for this origin

`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
<code>relid</code>	oid	The OID of the relation
<code>nspname</code>	name	Name of the schema relation is in
<code>relname</code>	name	Name of the relation
<code>set_name</code>	name	Name of the replication set
<code>set_ops</code>	text[]	List of replicated operations
<code>rel_columns</code>	text[]	List of replicated columns (NULL = all columns) (*)
<code>row_filter</code>	text	Row filtering expression
<code>conflict_detection</code>	text	Conflict detection method used: <code>row_origin</code> (default), <code>row_version</code> or <code>column_level</code>

(*) 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.workers`

Information about running BDR worker processes.

This can be joined with `bdr.stat_activity` using pid to get even more insight into the state of BDR workers.

`bdr.workers` Columns

Name	Type	Description
worker_pid	int	Process Id of the worker process
worker_role	int	Numeric representation of worker role
worker_role_name	text	Name of the worker role
worker_subid	oid	Subscription Id if the worker is associated with one
worker_commit_timestamp	timestampz	Last commit timestamp processed by this worker if any
worker_local_timestamp	timestampz	Local time at which the above commit was processed if any

`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
node_group_name	name	Name of the BDR group
origin_name	name	Name of the origin node
source_name	name	
target_name	name	Name of the target node (normally local node)
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_role_name	text	Role name
worker_pid	int4	Process id of the worker causing the error
error_time	timestampz	Date and time of the error
error_age	interval	Duration since error
error_message	text	Description of the error
error_context_message	text	Context in which the error happened
remoterelid	oid	Oid of remote relation on that node
subwriter_id	oid	
subwriter_name	name	

bdr.autopartition_work_queue

Contains work items created and processed by autopartition worker. The work items are created on only one node and processed on different nodes.

bdr.autopartition_work_queue Columns

Column	Type	Description
ap_wq_workid	bigint	The Unique ID of the work item
ap_wq_ruleid	int	ID of the rule listed in autopartition_rules. Rules are specified using bdr.autopartition command
ap_wq_relname	name	Name of the relation being autopartitioned
ap_wq_relnamespace	name	Name of the tablespace specified in rule for this work item.
ap_wq_partname	name	Name of the partition created by the workitem
ap_wq_work_kind	char	The work kind can be either 'c' (Create Partition), 'm' (Migrate Partition), 'd' (Drop Partition), 'a' (Alter Partition)
ap_wq_work_sql	text	SQL query for the work item
ap_wq_work_depends	Oid[]	Oids of the nodes on which the work item depends

bdr.autopartition_workitem_status

The status of the work items which is updated locally on each node.

bdr.autopartition_workitem_status Columns

Column	Type	Description
ap_wi_workid	bigint	The ID of the work item
ap_wi_nodeid	Oid	Oid of the node on which the work item is being processed

Column	Type	Description
ap_wi_status	char	The status can be either 'q' (Queued), 'c' (Complete), 'f' (Failed), 'u' (Unknown)
ap_wi_started_at	timestamptz	The start timestamptz of work item
ap_wi_finished_at	timestamptz	The end timestamptz of work item

bdr.autopartition_local_work_queue

Contains work items created and processed by autopartition worker. This is similar to bdr.autopartition_work_queue, except that these work items are for locally managed tables. Each node creates and processes its own local work items, independent of other nodes in the cluster.

bdr.autopartition_local_work_queue Columns

Column	Type	Description
ap_wq_workid	bigint	The Unique ID of the work item
ap_wq_ruleid	int	ID of the rule listed in autopartition_rules. Rules are specified using bdr.autopartition command
ap_wq_relname	name	Name of the relation being autopartitioned
ap_wq_relnamespace	name	Name of the tablespace specified in rule for this work item.
ap_wq_partname	name	Name of the partition created by the workitem
ap_wq_work_kind	char	The work kind can be either 'c' (Create Partition), 'm' (Migrate Partition), 'd' (Drop Partition), 'a' (Alter Partition)
ap_wq_work_sql	text	SQL query for the work item
ap_wq_work_depends	Oid[]	Always NULL

bdr.autopartition_local_workitem_status

The status of the work items for locally managed tables.

bdr.autopartition_local_workitem_status Columns

Column	Type	Description
ap_wi_workid	bigint	The ID of the work item
ap_wi_nodeid	Oid	Oid of the node on which the work item is being processed
ap_wi_status	char	The status can be either 'q' (Queued), 'c' (Complete), 'f' (Failed), 'u' (Unknown)
ap_wi_started_at	timestamptz	The start timestamptz of work item
ap_wi_finished_at	timestamptz	The end timestamptz of work item

bdr.group_camo_details

Uses `bdr.run_on_all_nodes` to gather CAMO-related information from all nodes.

bdr.group_camo_details Columns

Name	Type	Description
node_id	text	Internal node id
node_name	text	Name of the node
camo_partner_of	text	Node name for whom this node is partner
camo_origin_for	text	Node name for whom this node is origin
is_camo_partner_connected	text	Connection status
is_camo_partner_ready	text	Readiness status
camo_transactions_resolved	text	Are there any pending and unresolved CAMO transactions
apply_lsn	text	Latest position reported as replayed (visible)
receive_lsn	text	Latest LSN of any change or message received (can go backwards in case of restarts)
apply_queue_size	text	Bytes difference between apply_lsn and receive_lsn

Note

This catalog is only present when bdr-enterprise extension is installed.

`bdr.group_raft_details`

Uses `bdr.run_on_all_nodes` to gather Raft Consensus status from all nodes.

`bdr.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
nodes	int	Number of nodes accessible
voting_nodes	int	Number of nodes voting
protocol_version	int	Protocol version for this node

`bdr.group_replslots_details`

Uses `bdr.run_on_all_nodes` to gather BDR/pglogical slot information from all nodes.

`bdr.group_replslots_details` Columns

Name	Type	Description
node_group_name	text	Name of the BDR group
origin_name	text	Name of the origin node

Name	Type	Description
target_name	text	Name of the target node
slot_name	text	Slot name on the origin node used by this subscription
active	text	Is the slot active (does it have a connection attached to it)
state	text	State of the replication (catchup, streaming, ...) or 'disconnected' if offline
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_byte	int8	Bytes difference between replay_lsn and current WAL write position

bdr.group_subscription_summary

Uses `bdr.run_on_all_nodes` to gather subscription status from all nodes.

bdr.group_subscription_summary Columns

Name	Type	Description
origin_node_name	text	Name of the origin of the subscription
target_node_name	text	Name of the target of the subscription
last_xact_replay_timestamp	text	Timestamp of the last replayed transaction
sub_lag_seconds	text	Lag between now and last_xact_replay_timestamp

bdr.group_versions_details

Uses `bdr.run_on_all_nodes` to gather BDR/pglogical information from all nodes.

bdr.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

Internal Catalogs and Views

`bdr.ddl_epoch`

An internal catalog table holding state per DDL epoch.

`bdr.ddl_epoch` Columns

Name	Type	Description
<code>ddl_epoch</code>	<code>int8</code>	Monotonically increasing epoch number
<code>origin_node_id</code>	<code>oid</code>	Internal node id of the node that requested creation of this epoch
<code>epoch_consume_timeout</code>	<code>timestamptz</code>	Timeout of this epoch
<code>epoch_consumed</code>	<code>boolean</code>	Switches to true as soon as the local node has fully processed the epoch

`bdr.internal_node_pre_commit`

Internal catalog table; please use the `bdr.node_pre_commit` view.

Note

This catalog is only present when bdr-enterprise extension is installed.

`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
<code>seqid</code>	<code>oid</code>	Internal OID of the sequence
<code>seqkind</code>	<code>char</code>	Internal sequence kind ('l'=local,'t'=timeshard,'g'=galloc)

`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

2.23 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.

EDB provides Postgres Enterprise Manager (PEM), which supports BDR from version 8.1. Alternatively, tools or users can make their own calls into BDR using the facilities discussed below.

Monitoring Overview

A BDR Group consists of multiple servers, often referred to as nodes. All of the nodes need to be monitored to ensure the health of the whole group.

The `bdr_monitor` role may execute the `bdr.monitor` functions to provide an assessment of BDR health using one of three levels:

- `OK` - often shown as Green
- `WARNING` - often shown as Yellow
- `CRITICAL` - often shown as Red
- as well as `UNKNOWN` - for unrecognized situations, often shown as Red

BDR also provides dynamic catalog views that show the instantaneous state of various internal metrics and also BDR metadata catalogs that store the configuration defaults and/or configuration changes requested by the user. Some of those views and tables are accessible by `bdr_monitor` or `bdr_read_all_stats`, but some contain user or internal information that has higher security requirements.

BDR allows you to monitor each of the nodes individually, or to monitor the whole group by access to a single node. If you wish to monitor each node individually, simply connect to each node and issue monitoring requests. If you wish to monitor the group from a single node then use the views starting with `bdr.group` since these requests make calls to other nodes to assemble a group-level information set.

If you have been granted access to the `bdr.run_on_all_nodes()` function by `bdr_superuser` then you may make your own calls to all nodes.

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 is an additional view used for monitoring of outgoing replication activity:

- `bdr.node_replication_rates` for monitoring outgoing replication

The `bdr.node_replication_rates` view gives an overall picture of the outgoing replication activity along with the catchup estimates for peer nodes, specifically.

```
# SELECT * FROM bdr.node_replication_rates;
-[ RECORD 1 ]-----+-----
peer_node_id   | 112898766
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 ]-----+-----
peer_node_id   | 312494765
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.

Note

This catalog is only present when bdr-enterprise extension is installed.

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 EDB Postgres Distributed 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 WAL senders using LCR

If the [Decoding Worker](#) is enabled, information about the current LCR ([Logical Change Record](#)) file for each WAL sender can be monitored via the function `bdr.wal_sender_stats`, e.g.:

```
postgres=# SELECT * FROM bdr.wal_sender_stats();
 pid | is_using_lcr | decoder_slot_name | lcr_file_name
-----+-----+-----+-----
 2059904 | f          |                   |
 2059909 | t          | bdr_postgres_bdrgroup_decoder | 0000000000000000000000000000000014000000000000000
 2059916 | t          | bdr_postgres_bdrgroup_decoder | 0000000000000000000000000000000014000000000000000
(3 rows)
```

If `is_using_lcr` is `FALSE`, `decoder_slot_name` / `lcr_file_name` will be `NULL`. This will be the case if the Decoding Worker is not enabled, or the WAL sender is serving a [logical standby](#).

Additionally, information about the Decoding Worker can be monitored via the function `bdr.get_decoding_worker_stat`, e.g.:

```
postgres=# SELECT * FROM bdr.get_decoding_worker_stat();
 pid | decoded_upto_lsn | waiting | waiting_for_lsn
-----+-----+-----+-----
 1153091 | 0/1E5EEE8      | t       | 0/1E5EF00
(1 row)
```

Monitoring BDR Replication Workers

All BDR workers show up in the system view `bdr.stat_activity`, which has the same columns and information content as `pg_stat_activity`. So this view offers these insights into the state of a BDR system:

- The `wait_event` column has enhanced information, if the reason for waiting is related to BDR.
- The `query` column will be blank in BDR workers, except when a writer process is executing DDL

The `bdr.workers` view shows BDR worker specific details, that are not available from `bdr.stat_activity`.

The view `bdr.worker_errors` shows errors (if any) reported by any worker. BDR 3.7 depended explicitly on `pglogical 3.7` 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 3.7 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. In case of global advisory locks, `lock_type` column shows `GLOBAL_LOCK_ADVISORY` and `relation` column shows the advisory key(s) on which the lock is acquired.

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 FROM bdr.global_locks;
-[ RECORD 1 ]-----
lock_type | GLOBAL_LOCK_DDL
relation  | [NULL]
pid       | 15534
-[ RECORD 2 ]-----
lock_type | GLOBAL_LOCK_DML
relation  | someschema.sometable
pid       | 15534
```

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 `bdr.conflict_history`. Since this contains full details of conflicting data, the rows are protected by row-level security to ensure they are visible only by owners of replicated tables. Owners should expect conflicts and analyze them to see which, if any, might be considered as problems to be resolved.

For monitoring purposes use `bdr.conflict_history_summary`, which does not contain user data. An example query to count the number of conflicts seen within the current day using an efficient query plan is:

```
SELECT count(*)
FROM bdr.conflict_history_summary
WHERE local_time > date_trunc('day',
current_timestamp)
AND local_time < date_trunc('day', current_timestamp + '1
day');
```

External Monitoring

User supplied metadata can be stored to allow monitoring tools to understand and monitor the EDB Postgres Distributed cluster. By centralizing this information, external tools can access any single node and read details about the whole cluster, such as network cost and warning/alarm thresholds for specific connections.

`bdr_superuser` has the privileges on these functions and tables. The view `bdr.network_monitoring` is also accessible by the `bdr_read_all_stats` role.

bdr.set_node_location

This function inserts node metadata into `bdr.node_location`

Synopsis

```
bdr.set_node_location(
    node_group_name text,
    node_name text,
    node_region
text,
    node_location
text);
```

Parameters

- `node_group_name` - name of the BDR group
- `node_name` - name of the node
- `node_region` - the datacenter site or Region
- `node_location` - the server name, availability zone etc..

bdr.set_network_path_info

This function inserts network path metadata for network paths between nodes into the table `bdr.network_path_info`.

Synopsis

```
bdr.set_network_path_info(
    node_group_name text,
    region1 text,
    region2 text,
    location1 text,
    location2 text,
    network_cost numeric,
    warning_threshold numeric,
    alarm_threshold numeric)
```

Parameters

- `node_group_name` - name of the BDR group
- `region1` - the origin server name
- `region2` - the remote server name
- `location1` - the origin datacente name
- `location2` - the remote datacenter name
- `network_cost` - an abstract value representing the cost of network transfer
- `warning_threshold` - a delay above which a threshold should be raised
- `alarm_threshold` - a delay above which an alarm should be raised

bdr.network_monitoring view

This view collects information about the network path between nodes.

The configuration of logging is defined by the `bdr.alter_node_set_log_config` function.

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](configuration.md#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.

Stats counters for these views can be reset to zero using the functions `bdr.reset_subscription_stats` and `bdr.reset_relation_stats`.

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.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.group_versions_details;
 node_name | postgres_version | pglogical_version |
bdr_version
```

```
-----+-----+-----+-----
 node1    | 13.3             | 3.7.10            |
3.7.10
 node2    | 13.3             | 3.7.10            |
3.7.10
```

The recommended setup is to try to have all nodes running the same latest versions as soon as possible. It is recommended that the cluster does not run different BDR versions for too long. BDR and pglogical versions may differ on the same node without problem.

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.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 EDB Postgres Distributed cluster without Raft Consensus working might be as follows:

- BDR data changes replication may still be working correctly
- Global DDL/DML locks will not work
- Galloc sequences will eventually run out of chunks
- Eager Replication 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.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.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 pqlogical 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.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>`, which can be accessed using the function `bdr.local_group_slot_name()`.

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 is however inactive most of the time. BDR maintains this slot and advances its LSN when all other peers have already

consumed the corresponding transactions. Consequently it is not necessary to monitor the status of the group slot.

The function `bdr.monitor_local_replslots()` provides a summary of whether all BDR node replication slots are working as expected, e.g.:

```
bdrdb=# SELECT * FROM bdr.monitor_local_replslots();
 status |
message |
-----+-----
  OK    | All BDR replication slots are working
correctly
```

One of the following status summaries will be returned:

- UNKNOWN: This node is not part of any BDR group
- OK: All BDR replication slots are working correctly
- OK: This node is part of a subscriber-only group
- CRITICAL: There is at least 1 BDR replication slot which is inactive
- CRITICAL: There is at least 1 BDR replication slot which is missing

Monitoring 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 `bdr.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`.

2.24 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 described 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`) or Enterprise Edition (`EE`); this function can be used to check which of those is currently installed. Deprecated.

`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 version and patch release into a single numerical value:

```
MAJOR_VERSION * 10000 + MINOR_VERSION * 100 + PATCH_RELEASE
```

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.

`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.

Note

This is only available on EDB Postgres Extended.

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.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. By default, the connection is created with `bdr.ddl_replication = off`, since the command are already being sent to all of the nodes in the cluster.

Be careful when using this function since you risk breaking replication and causing inconsistencies between nodes. Use either transparent DDL replication or `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=node1 port=5432 dbname=bdrdb user=postgres ",
    "node_id": "2232128708",
    "response": {
      "command_status": "SELECT 1",
      "command_tuples": [
        {
          "origin_name": "node1",
          "target_name": "node2",
          "local_slot_name": "bdr_bdrdb_bdrgroup_node2",
          "replay_lag_size": "0 bytes"
        }
      ]
    }
  },
  {
    "node_name": "node1"
  },
  {
    "dsn": "host=node2 port=5432 dbname=bdrdb user=postgres ",
    "node_id": "2058684375",
    "response": {
      "command_status": "SELECT 1",
      "command_tuples": [
        {
          "origin_name": "node2",
          "target_name": "node1",
          "local_slot_name": "bdr_bdrdb_bdrgroup_node1",
          "replay_lag_size": "0 bytes"
        }
      ]
    }
  },
  {
    "node_name": "node2"
  }
]
```

```
}
]
```

bdr.run_on_nodes

Function to run a query on a specified list of nodes.

Warning

This function will run an arbitrary query on remote nodes 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_nodes(node_names text[], query text)
```

Parameters

- `node_names` - text ARRAY of node names where query will be executed.
- `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. By default, the connection is created with `bdr.ddl_replication = off`, since the command are already being sent to all of the nodes in the cluster.

Be careful when using this function since you risk breaking replication and causing inconsistencies between nodes. Use either transparent DDL replication or `bdr.replicate_ddl_command()` to replicate DDL. DDL may be blocked in a future release.

bdr.run_on_group

Function to run a query on a group of nodes.

Warning

This function will run an arbitrary query on remote nodes 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_group(node_group_name text, query text)
```

Parameters

- `node_group_name` - name of node group where query will be executed.
- `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. By default, the connection is created with `bdr.ddl_replication = off`, since the command are already being sent to all of the nodes in the cluster.

Be careful when using this function since you risk breaking replication and causing inconsistencies between nodes. Use either transparent DDL replication or `bdr.replicate_ddl_command()` to replicate DDL. DDL may be blocked in a future release.

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.wait_for_xid_progress

This function can be used to wait for the given transaction (identified by it's XID) originated at the given node (identified by it's node id) to make enough progress on the cluster. The progress is defined as the transaction being applied on a node and this node having seen all other replication changes done before the transaction is applied.

Synopsis

```
bdr.wait_for_xid_progress(origin_node_id oid, origin_topxid int4, allnodes boolean DEFAULT true)
```

Parameters

- `origin_node_id` - node id of the node where the transaction was originated.
- `origin_topxid` - XID of the transaction.
- `allnodes` - if `true` then wait for the transaction to progress on all nodes. Otherwise only wait for the current node.

Notes

The function can be used only for those transactions that have replicated a DDL command because only those transactions are tracked currently. If a wrong `origin_node_id` or `origin_topxid` is supplied, the function may wait forever or until `statement_timeout` is hit.

`bdr.local_group_slot_name`

Returns the name of the group slot on the local node.

Example

```
bdrdb=# SELECT bdr.local_group_slot_name();
 local_group_slot_name
-----
 bdr_bdrdb_bdrgroup
```

`bdr.node_group_type`

Returns the type of the given node group. Returned value is same as what was passed to `bdr.create_node_group()` when the node group was created, except `normal` is returned if the `node_group_type` was passed as NULL when the group was created.

Example

```
bdrdb=# SELECT bdr.node_group_type('bdrgroup');
 node_group_type
-----
 normal
```

Global Advisory Locks

BDR supports global advisory locks. These locks are very similar to the advisory locks available in PostgreSQL except that the advisory locks supported by BDR are global in nature. They follow semantics similar to DDL locks. So an advisory lock is obtained by majority consensus and hence can be used even if one or more nodes are down or lagging behind, as long as a majority of all nodes can work together.

Currently we only support EXCLUSIVE locks. So if another node or another backend on the same node has already acquired the advisory lock on the object, then other nodes or backends must wait for the lock to be released.

Advisory lock is transactional in nature. So the lock is automatically released when the transaction ends unless it is explicitly released before the end of the

transaction, in which case it will be available as soon as it's released. Session level advisory locks are not currently supported.

Global advisory locks are re-entrant. So if the same resource is locked three times it must then be unlocked three times to be released for other sessions' use.

`bdr.global_advisory_lock`

This function acquires an EXCLUSIVE lock on the provided object. If the lock is not available, then it will wait until the lock becomes available or the `bdr.global_lock_timeout` is reached.

Synopsis

```
bdr.global_advisory_lock(key bigint)
```

parameters

- `key` - the object on which an advisory lock is acquired.

Synopsis

```
bdr.global_advisory_lock(key1 integer, key2 integer)
```

parameters

- `key1` - first part of the composite key.
- `key2` - second part of the composite key.

`bdr.global_advisory_unlock`

This function released previously acquired lock on the application defined source. The lock must have been previously obtained in the same transaction by the application, otherwise an ERROR is raised.

Synopsis

```
bdr.global_advisory_unlock(key bigint)
```

parameters

- `key` - the object on which advisory lock is acquired.

Synopsis

```
bdr.global_advisory_unlock(key1 integer, key2 integer)
```

parameters

- `key1` - first part of the composite key.
- `key2` - second part of the composite key.

Monitoring functions

`bdr.monitor_group_versions`

To provide a cluster-wide version check, this function uses BDR version information returned from the view `bdr.group_version_details`.

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`

To provide a cluster-wide Raft check, this function uses BDR Raft information returned from the view `bdr.group_raft_details`.

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_local_replslots`

This function uses replication slot status information returned from the 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_local_replslots()
```

Notes

This function returns a record with fields `status` and `message`, as explained in [Monitoring replication slots](#).

bdr.wal_sender_stats

If the [decoding worker](#) is enabled, this function shows information about the decoder slot and current LCR (logical change record) segment file being read by each WAL sender.

Synopsis

```
bdr.wal_sender_stats()
```

Output columns

- `pid` – PID of the WAL sender (corresponds to `pg_stat_replication`'s `pid` column).
- `is_using_lcr` – Whether the WAL sender is sending LCR files. The next columns are `NULL` if `is_using_lcr` is `FALSE`.
- `decoder_slot_name` – The name of the decoder replication slot.
- `lcr_file_name` – The name of the current LCR file.

bdr.get_decoding_worker_stat

If the [decoding worker](#) is enabled, this function shows information about the state of the decoding worker associated with the current database. This also provides more granular information about decoding worker progress than is available via `pg_replication_slots`.

Synopsis

```
bdr.get_decoding_worker_stat()
```

Output columns

- `pid` – The PID of the decoding worker (corresponds to the column `active_pid` in `pg_replication_slots`).

- `decoded_upto_lsn` – LSN up to which the decoding worker read transactional logs.
- `waiting` – Whether the decoding worker is waiting for new WAL.
- `waiting_for_lsn` – The LSN of the next expected WAL.

Notes

For further details, see [Monitoring WAL senders using LCR](#).

2.25 Appendix B: Conflict Details

2.26 Appendix B: 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.

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 a unique identifier.

- If the resolver for the `update_origin_change` conflict is set to `skip`, and `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.
- 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.
- Upgrades on 2ndQPostgres 13 from BDR 3.7.7 are only supported by adding new nodes, and **not** through in-place upgrade of the same data directory.
- The `bdr.monitor_local_replslots()` function may return CRITICAL result saying "There is at least 1 BDR replication slot which is missing" even if all slots exist in presence of logical standbys or subscribe-only node groups.
- Decoding Worker feature does not work with CAMO/EAGER
- Decoding Worker works only with the default replication sets

2.27 Appendix C: CAMO Reference Client Implementations

3 High Availability Routing for Postgres (HARP)

High Availability Routing for Postgres (HARP) is a cluster-management tool for [Bi-directional Replication \(BDR\)](#) clusters. The core design of the tool is to route all application traffic in a single data center or region to only one node at a time. This node, designated the lead master, acts as the principle write target to reduce the potential for data conflicts.

HARP leverages a distributed consensus model to determine availability of the BDR nodes in the cluster. On failure or unavailability of the lead master, HARP elects a new lead master and redirects application traffic.

Together with the core capabilities of BDR, this mechanism of routing application traffic to the lead master node enables fast failover and switchover without risk of data loss.

HARP requires BDR versions 3.6 and later.

3.1 Release Notes

High Availability Routing for Postgres (HARP) is a cluster-management tool for [Bi-directional Replication \(BDR\)](#) clusters. The core design of the tool is to route all application traffic in a single data center or region to only one node at a time. This node, designated the lead master, acts as the principle write target to reduce the potential for data conflicts.

The release notes in this section provide information on what was new in each release.

Version	Release Date
2.4.1	16 May 2024
2.4.0	05 Mar 2024
2.3.2	17 Oct 2023
2.3.1	27 Jul 2023
2.3.0	12 Jul 2023
2.2.3	16 May 2023
2.2.2	30 Mar 2023
2.2.1	16 Nov 2022
2.2.0	22 Aug 2022
2.1.1	21 Jun 2022
2.1.0	17 May 2022
2.0.3	31 Mar 2022
2.0.2	24 Feb 2022
2.0.1	31 Jan 2021
2.0.0	01 Dec 2021

3.1.1 Version 2.4.1

Released: 16 May 2024

This is a patch release of HARP 2 that includes internal maintenance fixes.

Type	Description
Change	Routine security library upgrades

3.1.2 Version 2.4.0

This is a minor release of HARP 2 that includes internal maintenance fixes.

Type	Description
Change	Routine security library upgrades and refreshed build toolchain

3.1.3 Version 2.3.2

This is a patch release of HARP 2 that includes fixes for issues identified in previous versions.

Type	Description
Bug fix	Multiple fixes related to network partition in BDR DCS.
Bug fix	Fix intermittent harp-manager hang issue.
Bug fix	In Go driver calls, ensure rows.Err() is checked every time after rows.Next() returns false.
Change	Improve the BDR DCS's List method performance.

3.1.4 Version 2.3.1

This is a patch release of HARP 2 that includes fixes for issues identified in previous versions.

Type	Description
Bug fix	HARP Proxy cannot determine the leader after an extended outage - (BDR-3768).
Bug fix	Upgrade database driver library version which fixes <code>connect_timeout</code> issue when <code>sslmode=allow</code> or <code>sslmode=prefer</code> .

3.1.5 Version 2.3.0

This is a patch release of HARP 2 that includes enhancements and fixes for issues identified in previous versions.

Type	Description
------	-------------

Type	Description
Bug fix	Fix the CAMO lag computation issue - (BDR-3341).
Bug fix	Fix the Etcd TLS issue when only <code>ssl_ca_file</code> is set - (BDR-3582).
Feature	Add HTTP(S) health check probes for HARP.

3.1.6 Version 2.2.3

This is a patch release of HARP 2 that includes fixes for issues identified in previous versions.

Type	Description
Bug fix	Update consensus check to use <code>bdr.get_raft_status</code> instead of <code>bdr.monitor_group_raft</code> (BDR-3383)

3.1.7 Version 2.2.2

This is a patch release of HARP 2 that includes fixes for issues identified in previous versions.

Type	Description
Change	Upgrade 3rd party dependencies to fix Github dependabot alerts

3.1.8 Version 2.2.1

This is a patch release of HARP 2 that includes fixes for issues identified in previous versions.

Type	Description
Bug Fix	Fix connection leak issue (BDR-2530).

3.1.9 Version 2.2.0

This is a patch release of HARP 2 that includes fixes for issues identified in previous versions.

Type	Description
Enhancement	Add support for <code>sslpassword</code> in a connection string, including processing hook (HNG-626).

3.1.10 Version 2.1.1

This is a patch release of HARP 2 that includes fixes for issues identified in previous versions.

Type	Description
Enhancement	Log a warning on loss of DCS connection
Enhancement	Log a warning when metadata refresh is taking too long - usually due to high latency network
Bug Fix	Restart harp_proxy.service on a failure
Bug Fix	Fix concurrency issue with connection management in harpctl
Bug Fix	Don't try to proxy connections to previous leader on unmanaged cluster
Bug Fix	Don't panic in harpctl when location is empty

3.1.11 Version 2.1.0

This is a minor release of HARP 2 that includes new features as well as fixes for issues identified in previous versions.

Type	Description
	The BDR DCS now uses a push notification from the consensus rather than through polling nodes.
Feature	This change reduces the time for new leader selection and the load that HARP does on the BDR DCS since it doesn't need to poll in short intervals anymore.
Feature	TPA now restarts each HARP Proxy one by one and wait until they come back to reduce any downtime incurred by the application during software upgrades.
	The support for embedding PGBouncer directly into HARP Proxy is now deprecated and will be removed in the next major release of HARP.
Feature	It's now possible to configure TPA to put PGBouncer on the same node as HARP Proxy and point to that HARP Proxy.
Bug Fix	<code>harpctl promote <node_name></code> would occasionally promote a different node than the one specified. This has been fixed. (RT75406)
Bug Fix	Fencing would sometimes fail when using BDR as the Distributed Consensus Service. This has been corrected.
Bug Fix	<code>harpctl apply</code> no longer turns off routing for leader after the cluster has been established. (RT80790)
Bug Fix	Harp-manager no longer exits if it cannot start a failed database. Harp-manager will keep retrying with randomly increasing periods. (RT78516)
Bug Fix	The internal pgbouncer proxy implementation had a memory leak. This has been remediated.

3.1.12 Version 2.0.3

This is a patch release of HARP 2 that includes fixes for issues identified in previous versions.

Type	Description
Enhancement	HARP Proxy supports read-only user dedicated TLS Certificate (RT78516)
Bug Fix	HARP Proxy continues to try and connect to DCS instead of exiting after 50 seconds. (RT75406)

3.1.13 Version 2.0.2

This is a patch release of HARP 2 that includes fixes for issues identified in previous versions.

Type	Description
	BDR consensus now generally available.
Enhancement	HARP offers multiple options for Distributed Consensus Service (DCS) source: etcd and BDR. The BDR consensus option can be used in deployments where etcd isn't present. Use of the BDR consensus option is no longer considered beta and is now supported for use in production environments.
	Transport layer proxy now generally available.
Enhancement	HARP offers multiple proxy options for routing connections between the client application and database: application layer (L7) and transport layer (L4). The network layer 4 or transport layer proxy simply forwards network packets, and layer 7 terminates network traffic. The transport layer proxy, previously called simple proxy, is no longer considered beta and is now supported for use in production environments.

3.1.14 Version 2.0.1

This is a patch release of HARP 2 that includes fixes for issues identified in previous versions.

Type	Description
Enhancement	Support for selecting a leader per location rather than relying on DCS like etcd to have separate setup in different locations. This still requires a majority of nodes to survive loss of a location, so an odd number of both locations and database nodes is recommended.
Enhancement	The BDR DCS now uses a push notification from the consensus rather than through polling nodes. This change reduces the time for new leader selection and the load that HARP does on the BDR DCS since it doesn't need to poll in short intervals anymore.
Enhancement	TPA now restarts each HARP Proxy one by one and wait until they come back to reduce any downtime incurred by the application during software upgrades.
Enhancement	The support for embedding PGBouncer directly into HARP Proxy is now deprecated and will be removed in the next major release of HARP. It's now possible to configure TPA to put PGBouncer on the same node as HARP Proxy and point to that HARP Proxy.
Bug Fix	<code>harpcctl promote <node_name></code> would occasionally promote a different node than the one specified. This has been fixed. [Support Ticket #75406]
Bug Fix	Fencing would sometimes fail when using BDR as the Distributed Consensus Service. This has been corrected.
Bug Fix	<code>harpcctl apply</code> no longer turns off routing for leader after the cluster has been established. [Support Ticket #80790]
Bug Fix	Harp-manager no longer exits if it cannot start a failed database. Harp-manager will keep retrying with randomly increasing periods. [Support Ticket #78516]
Bug Fix	The internal pgbouncer proxy implementation had a memory leak. This has been remediated.

3.1.15 Version 2.0.0

This is new major release of HARP that constitutes of complete rewrite of the product.

Type	Description
Engine	Complete rewrite of system in golang to optimize all operations
Engine	Cluster state can now be bootstrapped or revised via YAML
Feature	Configuration now in YAML, configuration file changed from <code>harp.ini</code> to <code>config.yml</code>

Type	Description
	HARP Proxy deprecates need for HAProxy in supported architecture.
Feature	The use of HARP Router to translate DCS contents into appropriate online or offline states for HTTP-based URI requests meant a load balancer or HAProxy was necessary to determine the lead master. HARP Proxy now does this automatically without periodic iterative status checks.
	Utilizes DCS key subscription to respond directly to state changes.
Feature	With relevant cluster state changes, the cluster responds immediately, resulting in improved failover and switchover times.
	Compatibility with etcd SSL settings.
Feature	It is now possible to communicate with etcd through SSL encryption.
	Zero transaction lag on switchover.
Feature	Transactions are not routed to the new lead node until all replicated transactions are replayed, thereby reducing the potential for conflicts.
	Experimental BDR Consensus layer.
Feature	Using BDR Consensus as the Distributed Consensus Service (DCS) reduces the amount of change needed for implementations.
	Experimental built-in proxy.
Feature	Proxy implementation for increased session control.

3.2 HARP functionality overview

HARP is a new approach to high availability for BDR clusters. It leverages a consensus-driven quorum to determine the correct connection endpoint in a semi-exclusive manner to prevent unintended multi-node writes from an application.

The importance of quorum

The central purpose of HARP is to enforce full quorum on any Postgres cluster it manages. Quorum is a term applied to a voting body that mandates a certain minimum of attendees are available to make a decision. More simply: majority rules.

For any vote to end in a result other than a tie, an odd number of nodes must constitute the full cluster membership. Quorum, however, doesn't strictly demand this restriction; a simple majority is enough. This means that in a cluster of N nodes, quorum requires a minimum of $N/2+1$ nodes to hold a meaningful vote.

All of this ensures the cluster is always in agreement regarding the node that is "in charge." For a EDB Postgres Distributed cluster consisting of multiple nodes, this determines the node that is the primary write target. HARP designates this node as the lead master.

Reducing write targets

The consequence of ignoring the concept of quorum, or not applying it well enough, can lead to a "split brain" scenario where the "correct" write target is ambiguous or unknowable. In a standard Postgres cluster, it's important that only a single node is ever writable and sending replication traffic to the remaining nodes.

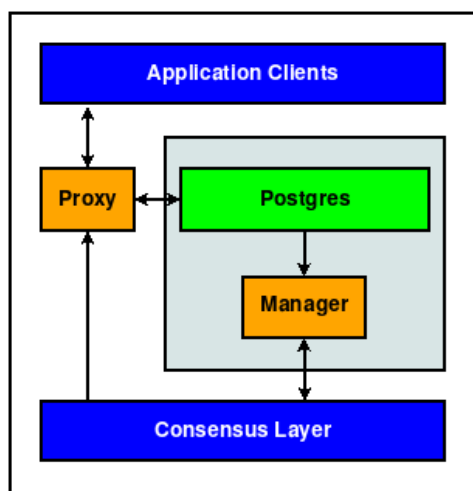
Even in multi-master-capable approaches such as BDR, it can be help to reduce the amount of necessary conflict management to derive identical data across the cluster. In clusters that consist of multiple BDR nodes per physical location or region, this usually means a single BDR node acts as a "leader" and remaining nodes are "shadow." These shadow nodes are still writable, but writing to them is discouraged unless absolutely necessary.

By leveraging quorum, it's possible for all nodes to agree on the exact Postgres node to represent the entire cluster or a local BDR region. Any nodes that lose contact with the remainder of the quorum, or are overruled by it, by definition can't become the cluster leader.

This restriction prevents split-brain situations where writes unintentionally reach two Postgres nodes. Unlike technologies such as VPNs, proxies, load balancers, or DNS, you can't circumvent a quorum-derived consensus by misconfiguration or network partitions. So long as it's possible to contact the consensus layer to determine the state of the quorum maintained by HARP, only one target is ever valid.

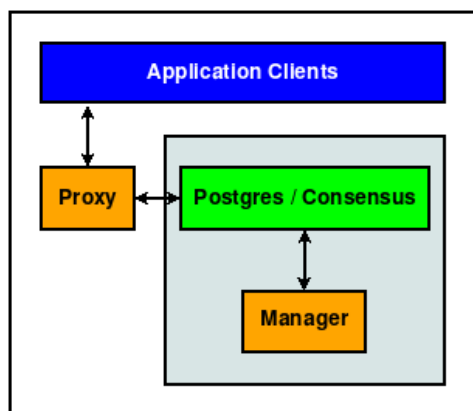
Basic architecture

The design of HARP comes in essentially two parts, consisting of a manager and a proxy. The following diagram describes how these interact with a single Postgres instance:



The consensus layer is an external entity where Harp Manager maintains information it learns about its assigned Postgres node, and HARP Proxy translates this information to a valid Postgres node target. Because Proxy obtains the node target from the consensus layer, several such instances can exist independently.

While using BDR as the consensus layer, each server node resembles this variant instead:

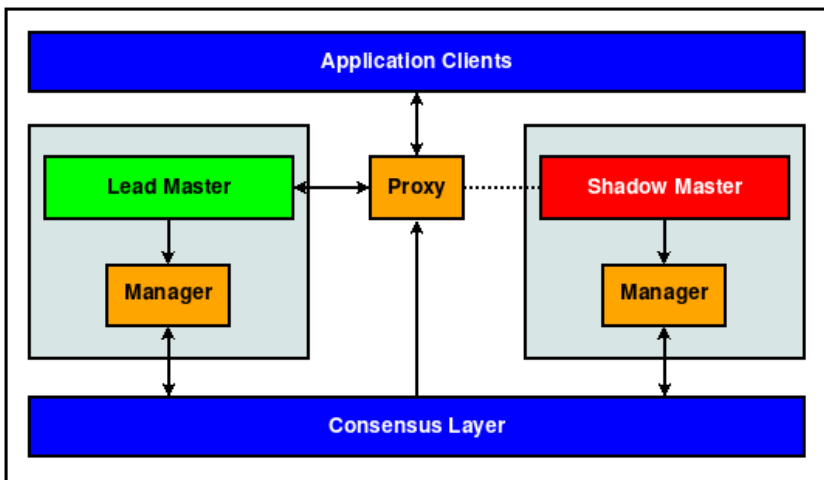


In either case, each unit consists of the following elements:

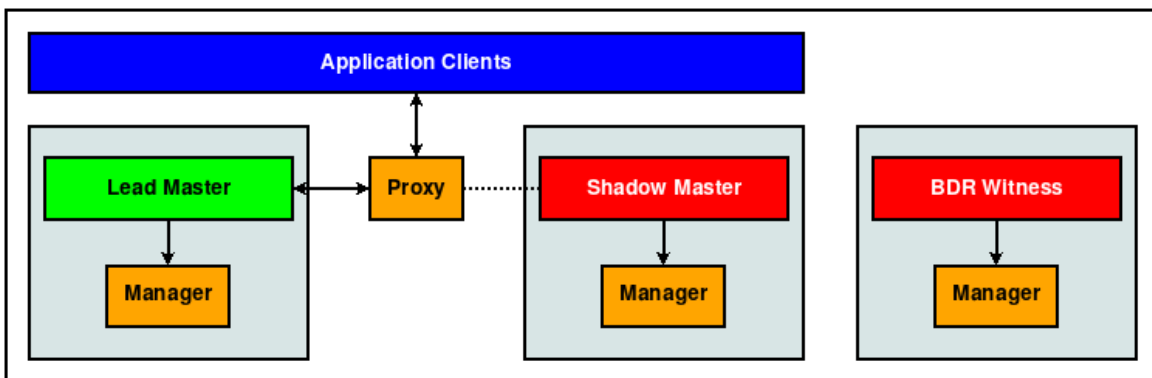
- A Postgres or EDB instance
- A consensus layer resource, meant to track various attributes of the Postgres instance
- A HARP Manager process to convey the state of the Postgres node to the consensus layer
- A HARP Proxy service that directs traffic to the proper lead master node, as derived from the consensus layer

Not every application stack has access to additional node resources specifically for the Proxy component, so it can be combined with the application server to simplify the stack.

This is a typical design using two BDR nodes in a single data center organized in a lead master/shadow master configuration:



When using BDR as the HARP consensus layer, at least three fully qualified BDR nodes must be present to ensure a quorum majority. (Not shown in the diagram are connections between BDR nodes.)



How it works

When managing a EDB Postgres Distributed cluster, HARP maintains at most one leader node per defined location. This is referred to as the lead master. Other BDR nodes that are eligible to take this position are shadow master state until they take the leader role.

Applications can contact the current leader only through the proxy service. Since the consensus layer requires quorum agreement before conveying leader state, proxy services direct traffic to that node.

At a high level, this mechanism prevents simultaneous application interaction with multiple nodes.

Determining a leader

As an example, consider the role of lead master in a locally subdivided BDR Always-On group as can exist in a single data center. When any Postgres or Manager resource is started, and after a configurable refresh interval, the following must occur:

1. The Manager checks the status of its assigned Postgres resource.
 - o If Postgres isn't running, try again after configurable timeout.
 - o If Postgres is running, continue.

2. The Manager checks the status of the leader lease in the consensus layer.
 - o If the lease is unclaimed, acquire it and assign the identity of the Postgres instance assigned to this manager. This lease duration is configurable, but setting it too low can result in unexpected leadership transitions.
 - o If the lease is already claimed by us, renew the lease TTL.
 - o Otherwise do nothing.

A lot more occurs, but this simplified version explains what's happening. The leader lease can be held by only one node, and if it's held elsewhere, HARP Manager gives up and tries again later.

Note

Depending on the chosen consensus layer, rather than repeatedly looping to check the status of the leader lease, HARP subscribes to notifications. In this case, it can respond immediately any time the state of the lease changes rather than polling. Currently this functionality is restricted to the etcd consensus layer.

This means HARP itself doesn't hold elections or manage quorum, which is delegated to the consensus layer. A quorum of the consensus layer must acknowledge the act of obtaining the lease, so if the request succeeds, that node leads the cluster in that location.

Connection routing

Once the role of the lead master is established, connections are handled with a similar deterministic result as reflected by HARP Proxy. Consider a case where HARP Proxy needs to determine the connection target for a particular backend resource:

1. HARP Proxy interrogates the consensus layer for the current lead master in its configured location.
2. If this is unset or in transition:
 - o New client connections to Postgres are barred, but clients accumulate and are in a paused state until a lead master appears.
 - o Existing client connections are allowed to complete current transactions and are then reverted to a similar pending state as new connections.
3. Client connections are forwarded to the lead master.

The interplay shown in this case doesn't require any interaction with either HARP Manager or Postgres. The consensus layer is the source of all truth from the proxy's perspective.

Colocation

The arrangement of the work units is such that their organization must follow these principles:

1. The manager and Postgres units must exist concomitantly in the same node.
2. The contents of the consensus layer dictate the prescriptive role of all operational work units.

This arrangement delegates cluster quorum responsibilities to the consensus layer, while HARP leverages it for critical role assignments and key/value storage. Neither storage nor retrieval succeeds if the consensus layer is inoperable or unreachable, thus preventing rogue Postgres nodes from accepting connections.

As a result, the consensus layer generally exists outside of HARP or HARP-managed nodes for maximum safety. Our reference diagrams show this separation, although it isn't required.

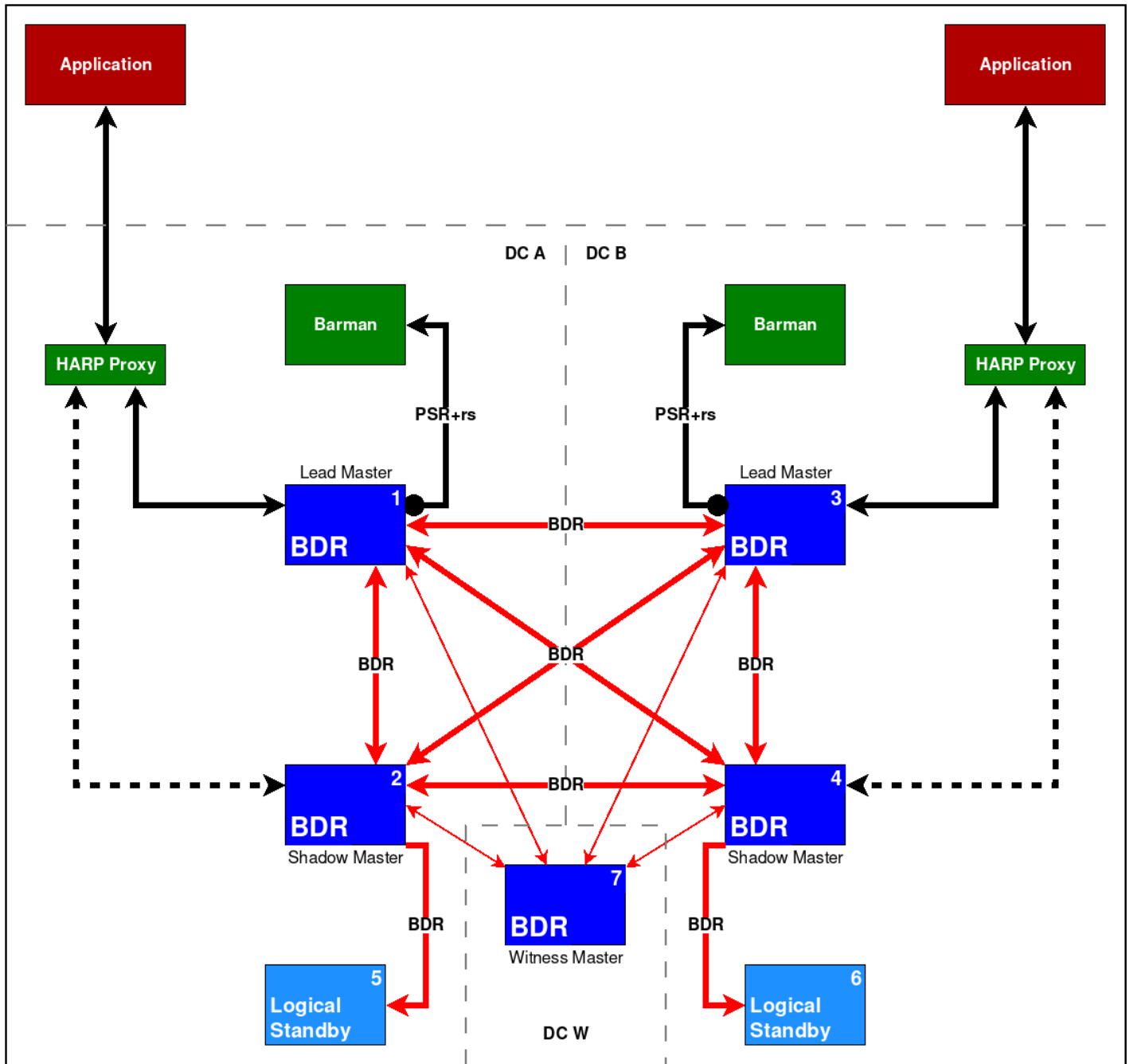
Note

To operate and manage cluster state, BDR contains its own implementation of the Raft Consensus model. You can configure HARP to leverage this same layer to reduce reliance on external dependencies and to preserve server resources. However, certain drawbacks to this approach are discussed in [Consensus layer](#).

Recommended architecture and use

HARP was primarily designed to represent a BDR Always-On architecture that resides in two or more data centers and consists of at least five BDR nodes. This configuration doesn't count any logical standby nodes.

The following diagram shows the current and standard representation:



In this diagram, HARP Manager exists on BDR Nodes 1-4. The initial state of the cluster is that BDR Node 1 is the lead master of DC A, and BDR Node 3 is the lead master of DC B.

This configuration results in any HARP Proxy resource in DC A connecting to BDR Node 1 and the HARP Proxy resource in DC B connecting to BDR Node 3.

Note

While this diagram shows only a single HARP Proxy per DC, this is an example only and should not be considered a single point of failure. Any number of HARP Proxy nodes can exist, and they all direct application traffic to the same node.

Location configuration

For multiple BDR nodes to be eligible to take the lead master lock in a location, you must define a location in the `config.yml` configuration file.

To reproduce the BDR Always-On reference architecture shown in the diagram, include these lines in the `config.yml` configuration for BDR Nodes 1 and 2:

```
location:
dca
```

For BDR Nodes 3 and 4, add:

```
location:
dcb
```

This applies to any HARP Proxy nodes that are designated in those respective data centers as well.

Health check

HARP provides the following HTTP(S) health check API endpoints. These APIs are `GET` requests and don't require a request body. See [Configurations](#) for more details about enabling and configuring this feature.

```
GET /health/is-ready
GET /health/is-live
```

Readiness

Readiness endpoint is available only for `harp-proxy`. On receiving a valid `'GET'` request, the proxy checks if it can successfully route connections to the current write leader. If the check returns successfully, then the API gives a response with a body containing `true` and a HTTP status code `200 (OK)`. Otherwise, it returns a body containing `false` with the HTTP status code `500 (Internal Server Error)`.

Liveness

Liveness checks return either `true` with HTTP status code `200 (OK)` or an error. They never return `false` because the HTTP server listening for requests is stopped if the corresponding HARP service fails to start or exists. This API endpoint is available for `harp-manager` and `harp-proxy` services.

BDR 3.7 compatibility

BDR 3.7 and later offers more direct location definition by assigning a location to the BDR node. This is done by calling the following SQL API function while connected to the BDR node. So for BDR Nodes 1 and 2, you might do this:

```
SELECT bdr.set_node_location('dca');
```

And for BDR Nodes 3 and 4:

```
SELECT bdr.set_node_location('dcb');
```

3.3 Installation

A standard installation of HARP includes two system services:

- HARP Manager (`harp-manager`) on the node being managed
- HARP Proxy (`harp-proxy`) elsewhere

There are two ways to install and configure these services to manage Postgres for proper quorum-based connection routing.

Software versions

HARP has dependencies on external software. These must fit a minimum version as listed here.

Software	Min version
etcd	3.4
PgBouncer	1.14

TPAExec

The easiest way to install and configure HARP is to use the EDB TPAExec utility for cluster deployment and management. For details on this software, see the [TPAExec product page](#).

Note

TPAExec is currently available only through an EULA specifically dedicated to EDB Postgres Distributed cluster deployments. If you can't access the TPAExec URL, contact your sales or account representative.

Configure TPAExec to recognize that cluster routing is managed through HARP by ensuring the TPA `config.yml` file contains these attributes:

```
cluster_vars:
  failover_manager: harp
```

Note

Versions of TPAExec earlier than 21.1 require a slightly different approach:

```
cluster_vars:
  enable_harp: true
```

After this, install HARP by invoking the `tpaexec` commands for making cluster modifications:

```
tpaexec provision ${CLUSTER_DIR}
tpaexec deploy
${CLUSTER_DIR}
```

No other modifications are necessary apart from cluster-specific considerations.

Package installation

Currently CentOS/RHEL packages are provided by the EDB packaging infrastructure. For details, see the [HARP product page](#).

etcd packages

Currently `etcd` packages for many popular Linux distributions aren't available by their standard public repositories. EDB has therefore packaged `etcd` for RHEL and CentOS versions 7 and 8, Debian, and variants such as Ubuntu LTS. You need access to our HARP package repository to use these libraries.

Consensus layer

HARP requires a distributed consensus layer to operate. Currently this must be either `bdr` or `etcd`. If using fewer than three BDR nodes, you might need to rely on `etcd`. Otherwise any BDR service outage reduces the consensus layer to a single node and thus prevents node consensus and disables Postgres routing.

etcd

If you're using `etcd` as the consensus layer, `etcd` must be installed either directly on the Postgres nodes or in a separate location they can access.

To set `etcd` as the consensus layer, include this code in the HARP `config.yml` configuration file:

```

dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379

```

When using TPAExec, all configured etcd endpoints are entered here automatically.

BDR

The `bdr` native consensus layer is available from BDR 3.6.21 and 3.7.3. This consensus layer model requires no supplementary software when managing routing for a EDB Postgres Distributed cluster.

To ensure quorum is possible in the cluster, always use more than two nodes so that BDR's consensus layer remains responsive during node maintenance or outages.

To set BDR as the consensus layer, include this in the `config.yml` configuration file:

```

dcs:
  driver:
bdr
  endpoints:
    - host=host1 dbname=bdrdb
user=harp_user
    - host=host2 dbname=bdrdb
user=harp_user
    - host=host3 dbname=bdrdb
user=harp_user

```

The endpoints for a BDR consensus layer follow the standard Postgres DSN connection format.

3.4 Configuring HARP for cluster management

The HARP configuration file follows a standard YAML-style formatting that was simplified for readability. This file is located in the `/etc/harp` directory by default and is named `config.yml`

You can explicitly provide the configuration file location to all HARP executables by using the `-f / --config` argument.

Standard configuration

HARP essentially operates as three components:

- HARP Manager
- HARP Proxy
- harpctl

Each of these use the same standard `config.yml` configuration format, which always include the following sections:

- `cluster.name` – The name of the cluster to target for all operations.
- `dc`s – DCS driver and connection configuration for all endpoints.

This means a standard preamble is always included for HARP operations, such as the following:

```
cluster:
  name: mycluster

dc
```

Other sections are optional or specific to the named HARP component.

Cluster name

The `name` entry under the `cluster` heading is required for all interaction with HARP. Each HARP cluster has a name for both disambiguation and for labeling data in the DCS for the specific cluster.

HARP Manager writes information about the cluster here for consumption by HARP Proxy and harpctl. HARP Proxy services direct traffic to nodes in this cluster. The `harpctl` management tool interacts with this cluster.

DCS settings

Configuring the consensus layer is key to HARP functionality. Without the DCS, HARP has nowhere to store cluster metadata, can't hold leadership elections, and so on. Therefore this portion of the configuration is required, and certain elements are optional.

Specify all elements under a section named `dc`s with these multiple supplementary entries:

- `driver` : Required type of consensus layer to use. Currently can be `etcd` or `bdr` . Support for `bdr` as a consensus layer is experimental. Using `bdr` as the consensus layer reduces the additional software for consensus storage but expects a minimum of three full BDR member nodes to maintain quorum during database maintenance.
- `endpoints` : Required list of connection strings to contact the DCS. List every node of the DCS here if possible. This ensures HARP continues to function as long as a majority of the DCS can still operate and be reached by the network.

Format when using `etcd` as the consensus layer is as follows:

```

dcs:
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379

```

Format when using the experimental `bdr` consensus layer is as follows:

```

dcs:
  # only DSN format is
  # supported
  endpoints:
    - "host=host1 port=5432 dbname=bdrdb
      user=postgres"
    - "host=host2 port=5432 dbname=bdrdb
      user=postgres"
    - "host=host3 port=5432 dbname=bdrdb
      user=postgres"

```

Currently, `bdr` consensus layer requires the first endpoint to point to the local postgres instance.

- `request_timeout`: Time in milliseconds to consider a request as failed. If HARP makes a request to the DCS and receives no response in this time, it considers the operation as failed. This can cause the issue to be logged as an error or retried, depending on the nature of the request. Default: 250.

The following DCS SSL settings apply only when `driver: etcd` is set in the configuration file:

- `ssl`: Either `on` or `off` to enable SSL communication with the DCS. Default: `off`
- `ssl_ca_file`: Client SSL certificate authority (CA) file.
- `ssl_cert_file`: Client SSL certificate file.
- `ssl_key_file`: Client SSL key file.

Example

This example shows how to configure HARP to contact an `etcd` DCS consisting of three nodes:

```

dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379

```

Health check API settings

HARP provides [HTTP\(S\) health check APIs](#). Add the following settings only if health check APIs are required. Specify all elements under a section named `http` with these multiple supplementary entries:

- `enable`: Set to `true` to enable HTTP(S) health check APIs. Defaults to `false`.

- `host` : Host name of HTTP(S) API server. Typically the same as the hostname of the HARP service as it's running within the HARP service. Defaults to `0.0.0.0`.
- `port` : Port number of HTTP(S) server. Defaults to `8080`.
- `probes.timeout` : Timeout value for the API endpoints. Defaults to `10s`.
- `secure` : Enable HTTPS. Defaults to `false`.

The following `http` settings apply and are mandatory if `secure: true` is set:

- `cert_file` : HTTP server SSL certificate file. Mandatory if `secure` is set to `true`.
- `key_file` : HTTP server SSL key file. Mandatory if `secure` is set to `true`.

Example

```
http:
  enable: true
  host: "0.0.0.0"
  port: 8080
  secure: false
  cert_file: ""
  key_file: ""
  probes:
    timeout:
10s
```

HARP Manager specific

Besides the generic service options required for all HARP components, Manager needs other settings:

- `log_level` : One of `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `CRITICAL`, which might alter the amount of log output from HARP services.
- `name` : Required name of the Postgres node represented by this Manager. Since the Manager can only represent a specific node, that node is named here and it is used as the name of this Manager. If this is a BDR node, it must match the value used at node creation when executing the `bdr.create_node(node_name, ...)` function and as reported by the `bdr.local_node_summary.node_name` view column. Alphanumeric characters and underscores only.
- `start_command` : This can be used instead of the information in DCS for starting the database to monitor. This is required if using bdr as the consensus layer.
- `status_command` : This can be used instead of the information in DCS for the HARP Manager to determine whether the database is running. This is required if using bdr as the consensus layer.
- `stop_command` : This can be used instead of the information in DCS for stopping the database.
- `db_retry_wait_min` : The initial time in seconds to wait if HARP Manager cannot connect to the database before trying again. HARP Manager will increase the wait time with each attempt, up to the `db_retry_wait_max` value.
- `db_retry_wait_max` : The maximum time in seconds to wait if HARP Manager cannot connect to the database before trying again.

Thus a complete configuration example for HARP Manager might look like this:

```

cluster:
  name: mycluster

dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379

manager:
  name: node1
  log_level: INFO

```

This configuration is essentially the DCS contact information, any associated service customizations, the name of the cluster, and the name of the node. All other settings are associated with the node and is stored in the DCS.

Read the [Node bootstrapping](#) for more about specific node settings and initializing nodes to be managed by HARP Manager.

HARP Proxy specific

Some configuration options are specific to HARP Proxy. These affect how the daemon operates and thus are currently located in `config.yml`.

Specify Proxy-based settings under a `proxy` heading, and include:

- `location`: Required name of location for HARP Proxy to represent. HARP Proxy nodes are directly tied to the location where they are running, as they always direct traffic to the current lead master node. Specify location for any defined proxy.
- `log_level`: One of `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `CRITICAL`, which might alter the amount of log output from HARP services.
 - Default: `INFO`
- `name`: Name of this specific proxy. Each proxy node is named to ensure any associated statistics or operating state are available in status checks and other interactive events.
- `type`: Specifies whether to use pgbouncer or the experimental built-in passthrough proxy. All proxies must use the same proxy type. We recommend to experimenting with only the simple proxy in combination with the experimental BDR DCS. Can be `pgbouncer` or `builtin`.
 - Default: `pgbouncer`
- `pgbouncer_bin_dir`: Directory where PgBouncer binaries are located. As HARP uses PgBouncer binaries, it needs to know where they are located. This can depend on the platform or distribution, so it has no default. Otherwise, the assumption is that the appropriate binaries are in the environment's `PATH` variable.

Example

HARP Proxy requires the cluster name, DCS connection settings, location, and name of the proxy in operation. For example:

```

cluster:
  name: mycluster

dcs:
  driver: etcd
  endpoints:

```

```

- host1:2379
- host2:2379
- host3:2379

proxy:
  name:
  proxy1
  location:
  dcl
  pgbouncer_bin_dir: /usr/sbin

```

All other attributes are obtained from the DCS on proxy startup.

Runtime directives

While it is possible to configure HARP Manager, HARP Proxy, or harpctl with a minimum of YAML in the `config.yml` file, some customizations are held in the DCS. These values must either be initialized via bootstrap or set specifically with `harpctl set` directives.

Cluster-wide

Set these settings under a `cluster` YAML heading during bootstrap, or modify them with a `harpctl set cluster` command.

- `event_sync_interval`: Time in milliseconds to wait for synchronization. When events occur in HARP, they do so asynchronously across the cluster. HARP managers start operating immediately when they detect metadata changes, and HARP proxies might pause traffic and start reconfiguring endpoints. This is a safety interval that roughly approximates the maximum amount of event time skew that exists between all HARP components.

For example, suppose Node A goes offline and HARP Manager on Node B commonly receives this event 5 milliseconds before Node C. A setting of at least 5 ms is then needed to ensure all HARP Manager services receive the event before they begin to process it.

This also applies to HARP Proxy.

Node directives

You can change most node-oriented settings and then apply them while HARP Manager is active. These items are retained in the DCS after initial bootstrap, and thus you can modify them without altering a configuration file.

Set these settings under a `node` YAML heading during bootstrap, or modify them with a `harpctl set node` command.

- `node_type`: The type of this database node, either `bdr` or `witness`. You can't promote a witness node to leader.
- `camo_enforcement`: Whether to strictly enforce CAMO queue state. When set to `strict`, HARP never allows switchover or failover to a BDR CAMO partner node unless it's fully caught up with the entire CAMO queue at the time of the migration. When set to `lag_only`, only standard lag thresholds such as `maximum_camo_lag` are applied.
- `dcx_reconnect_interval`: The interval, measured in milliseconds, between attempts that a disconnected node tries to reconnect to the DCS.
 - Default: 1000.
- `dsn`: Required full connection string to the managed Postgres node. This parameter applies equally to all HARP services and enables micro-architectures that run only one service per container.

Note

HARP sets the `sslmode` argument to `require` by default and prevents connections to servers that don't require SSL. To disable this behavior, explicitly set this parameter to a more permissive value such as `disable`, `allow`, or `prefer`.

- `db_data_dir`: Required Postgres data directory. This is required by HARP Manager to start, stop, or reload the Postgres service. It's also the default location for configuration files, which you can use later for controlling promotion of streaming replicas.
- `db_conf_dir`: Location of Postgres configuration files. Some platforms prefer storing Postgres configuration files away from the Postgres data directory. In these cases, set this option to that expected location.
- `db_log_file`: Location of Postgres log file.
 - Default: `/tmp/pg_ctl.out`
- `fence_node_on_dcs_failure`: If HARP can't reach the DCS, several readiness keys and the leadership lease expire. This implicitly prevents a node from routing consideration. However, such a node isn't officially fenced, and the Manager doesn't stop monitoring the database if `stop_database_when_fenced` is set to `false`.
 - Default: `False`
- `leader_lease_duration`: Amount of time in seconds the lead master lease persists if not refreshed. This allows any HARP Manager a certain grace period to refresh the lock, before expiration allows another node to obtain the lead master lock instead.
 - Default: `6`
- `lease_refresh_interval`: Amount of time in milliseconds between refreshes of the lead master lease. This essentially controls the time between each series of checks HARP Manager performs against its assigned Postgres node and when the status of the node is updated in the consensus layer.
 - Default: `2000`
- `max_dcs_failures`: The amount of DCS request failures before marking a node as fenced according to `fence_node_on_dcs_failure`. This setting prevents transient communication disruptions from shutting down database nodes.
 - Default: `10`
- `maximum_lag`: Highest allowable variance (in bytes) between last recorded LSN of previous lead master and this node before being allowed to take the lead master lock. This setting prevents nodes experiencing terminal amounts of lag from taking the lead master lock. Set to `-1` to disable this check.
 - Default: `-1`
- `maximum_camo_lag`: Highest allowable variance (in bytes) between last received LSN and applied LSN between this node and its CAMO partners. This applies only to clusters where CAMO is both available and enabled. Thus this applies only to BDR EE clusters where `pg2q.enable_camo` is set. For clusters with particularly stringent CAMO apply queue restrictions, set this very low or even to `0` to avoid any unapplied CAMO transactions. Set to `-1` to disable this check.
 - Default: `-1`
- `ready_status_duration`: Amount of time in seconds the node's readiness status persists if not refreshed. This is a failsafe that removes a node from being contacted by HARP Proxy if the HARP Manager in charge of it stops operating.
 - Default: `30`
- `db_bin_dir`: Directory where Postgres binaries are located. As HARP uses Postgres binaries, such as `pg_ctl`, it needs to know where they're located. This can depend on the platform or distribution, so it has no default. Otherwise, the assumption is that the appropriate binaries are in the

environment's `PATH` variable.

- `priority` : Any numeric value. Any node where this option is set to `-1` can't take the lead master role, even when attempting to explicitly set the lead master using `harpctl`.
 - Default: 100
- `stop_database_when_fenced` : Rather than removing a node from all possible routing, stop the database on a node when it is fenced. This is an extra safeguard to prevent data from other sources than HARP Proxy from reaching the database or in case proxies can't disconnect clients for some other reason.
 - Default: False
- `consensus_timeout` : Amount of milliseconds before aborting a read or write to the consensus layer. If the consensus layer loses quorum or becomes unreachable, you want near-instant errors rather than infinite timeouts. This prevents blocking behavior in such cases. When using `bdr` as the consensus layer, the highest recognized timeout is 1000 ms.
 - Default: 250
- `use_unix_socket` : Specifies for HARP Manager to prefer to use Unix sockets to connect to the database.
 - Default: False

All of these runtime directives can be modified via `harpctl`. Consider if you want to decrease the `lease_refresh_interval` to 100ms on `node1`:

```
harpctl set node node1
lease_refresh_interval=100
```

Proxy directives

You can change certain settings to the proxy while the service is active. These items are retained in the DCS after initial bootstrap, and thus you can modify them without altering a configuration file. Many of these settings are direct mappings to their PgBouncer equivalent, and we will note these where relevant.

Set these settings under a `proxies` YAML heading during bootstrap, or modify them with a `harpctl set proxy` command. Properties set by `harpctl set proxy` require a restart of the proxy.

- `auth_file` : The full path to a PgBouncer-style `userlist.txt` file. HARP Proxy uses this file to store a `pgbouncer` user that has access to PgBouncer's Admin database. You can use this for other users as well. Proxy modifies this file to add and modify the password for the `pgbouncer` user.
 - Default: `/etc/harp/userlist.txt`
- `auth_type` : The type of Postgres authentication to use for password matching. This is actually a PgBouncer setting and isn't fully compatible with the Postgres `pg_hba.conf` capabilities. We recommend using `md5`, `pam_cert`, or `scram-sha-256`.
 - Default: `md5`
- `auth_query` : Query to verify a user's password with Postgres. Direct access to `pg_shadow` requires admin rights. It's better to use a non-superuser that calls a `SECURITY DEFINER` function instead. If using TPAexec to create a cluster, a function named `pgbouncer_get_auth` is installed on all databases in the `pg_catalog` namespace to fulfill this purpose.
- `auth_user` : If `auth_user` is set, then any user not specified in `auth_file` is queried through the `auth_query` query from `pg_shadow` in the database, using `auth_user`. The password of `auth_user` is taken from `auth_file`.

- `client_tls_ca_file` : Root certificate file to validate client certificates. Requires `client_tls_sslmode` to be set.
- `client_tls_cert_file` : Certificate for private key. Clients can validate it. Requires `client_tls_sslmode` to be set.
- `client_tls_key_file` : Private key for PgBouncer to accept client connections. Requires `client_tls_sslmode` to be set.
- `client_tls_protocols` : TLS protocol versions allowed for client connections. Allowed values: `tlsv1.0`, `tlsv1.1`, `tlsv1.2`, `tlsv1.3`. Shortcuts: `all` (tlsv1.0,tlsv1.1,tlsv1.2,tlsv1.3), `secure` (tlsv1.2,tlsv1.3), `legacy` (all).
 - Default: `secure`
- `client_tls_sslmode` : Whether to enable client SSL functionality. Possible values are `disable`, `allow`, `prefer`, `require`, `verify-ca`, and `verify-full`.
 - Default: `disable`
- `database_name` : Required name that represents the database clients use when connecting to HARP Proxy. This is a stable endpoint that doesn't change and points to the current node, database name, port, etc., necessary to connect to the lead master. You can use the global value `*` here so all connections get directed to this target regardless of database name.
- `default_pool_size` : The maximum number of active connections to allow per database/user combination. This is for connection pooling purposes but does nothing in session pooling mode. This is a PgBouncer setting.
 - Default: 25
- `ignore_startup_parameters` : By default, PgBouncer allows only parameters it can keep track of in startup packets: `client_encoding`, `datestyle`, `timezone`, and `standard_conforming_strings`. All other parameters raise an error. To allow other parameters, you can specify them here so that PgBouncer knows that they are handled by the admin and it can ignore them. Often, you need to set this to `extra_float_digits` for Java applications to function properly.
 - Default: `extra_float_digits`
- `listen_address` : IP addresses where Proxy should listen for connections. Used by pgbouncer and builtin proxy.
 - Default: 0.0.0.0
- `listen_port` : System port where Proxy listens for connections. Used by pgbouncer and builtin proxy.
 - Default: 6432
- `max_client_conn` : The total maximum number of active client connections that are allowed on the proxy. This can be many orders of magnitude greater than `default_pool_size`, as these are all connections that have yet to be assigned a session or have released a session for use by another client connection. This is a PgBouncer setting.
 - Default: 100
- `monitor_interval` : Time in seconds between Proxy checks of PgBouncer. Since HARP Proxy manages PgBouncer as the actual connection management layer, it needs to periodically check various status and stats to verify it's still operational. You can also log or register some of this information to the DCS.
 - Default: 5
- `server_tls_protocols` : TLS protocol versions are allowed for server connections. Allowed values: `tlsv1.0`, `tlsv1.1`, `tlsv1.2`, `tlsv1.3`. Shortcuts: `all` (tlsv1.0,tlsv1.1,tlsv1.2,tlsv1.3), `secure` (tlsv1.2,tlsv1.3), `legacy` (all).

- Default: `secure`
- `server_tls_sslmode`: Whether to enable server SSL functionality. Possible values are `disable`, `allow`, `prefer`, `require`, `verify-ca`, and `verify-full`.
 - Default: `disable`
- `session_transfer_mode`: Method by which to transfer sessions. Possible values are `fast`, `wait`, and `reconnect`.
 - Default: `wait`

This property isn't used by the builtin proxy.

- `server_transfer_timeout`: The number of seconds Harp Proxy waits before giving up on a PAUSE and issuing a KILL command.
 - Default: 30

The following two options apply only when using the built-in proxy.

- `keepalive`: The number of seconds the built-in proxy waits before sending a keepalive message to an idle leader connection.
 - Default: 5
- `timeout`: The number of seconds the built-in proxy waits before giving up on connecting to the leader.
 - Default: 1

When using `harpctl` to change any of these settings for all proxies, use the `global` keyword in place of the proxy name. For example:

```
harpctl set proxy global
max_client_conn=1000
```

3.5 Cluster bootstrapping

To use HARP, a minimum amount of metadata must exist in the DCS. The process of "bootstrapping" a cluster essentially means initializing node, location, and other runtime configuration either all at once or on a per-resource basis.

This process is governed through the `harpctl apply` command. For more information, see [harpctl command-line tool](#).

Set up the DCS and make sure it is functional before bootstrapping.

Important

You can combine any or all of these example into a single YAML document and apply it all at once.

Cluster-wide bootstrapping

Some settings are applied cluster-wide and you can specify them during bootstrapping. Currently this applies only to the `event_sync_interval` runtime directive, but others might be added later.

The format is as follows:

```
cluster:
  name: mycluster
  event_sync_interval: 100
```

Assuming that file was named `cluster.yml`, you then apply it with the following:

```
harpctl apply
cluster.yml
```

If the cluster name isn't already defined in the DCS, this also initializes that value.

Important

The cluster name parameter specified here always overrides the cluster name supplied in `config.yml`. The assumption is that the bootstrap file supplies all necessary elements to bootstrap a cluster or some portion of its larger configuration. A `config.yml` file is primarily meant to control the execution of HARP Manager, HARP Proxy, or `harpctl` specifically.

Location bootstrapping

Every HARP node is associated with at most one location. This location can be a single data center, a grouped region consisting of multiple underlying servers, an Amazon availability zone, and so on. This is a logical structure that allows HARP to group nodes together such that only one represents the nodes in that location as the lead master.

Thus it is necessary to initialize one or more locations. The format for this is as follows:

```
cluster:
  name: mycluster

locations:
  - location:
    dc1
  - location:
    dc2
```

Assuming that file was named `locations.yml`, you then apply it with the following:

```
harpctl apply locations.yml
```

When performing any manipulation of the cluster, include the name as a preamble so the changes are directed to the right place.

Once locations are bootstrapped, they show up with a quick examination:

```
> harpctl get locations

Cluster  Location Leader Previous Leader Target Leader Lease Renewals
-----  -
mycluster dc1                                <nil>
mycluster dc2                                <nil>
```

Both locations are recognized by HARP and available for node and proxy assignment.

Node bootstrapping

HARP nodes exist in a named cluster and must have a designated name. Beyond this, all other settings are retained in the DCS, as they are dynamic and can affect how HARP interacts with them. To this end, bootstrap each node using one or more of the runtime directives discussed in [Configuration](#).

While bootstrapping a node, there are a few required fields:

- `name`
- `location`
- `dsn`
- `pg_data_dir`

Everything else is optional and can depend on the cluster. Because you can bootstrap multiple nodes at once, the format generally fits this structure:

```
cluster:
  name: mycluster

nodes:
  - name: node1
    location:
      dsn: host=node1 dbname=bdrdb user=postgres
      pg_data_dir: /db/pgdata
      leader_lease_duration: 10
      priority: 500
```

Assuming that file was named `node1.yml`, you then apply it with the following:

```
harpctl apply node1.yml
```

Once nodes are bootstrapped, they show up with a quick examination:

```
> harpctl get nodes
```

Cluster	Name	Location	Ready	Fenced	Allow	Routing	Routing	Status	Role	Type	Lock	Duration
mycluster	bdr1	dc1	true	false	true		ok		primary	bdr		

Proxy bootstrapping

Unlike locations or nodes, proxies can also supply a configuration template that is applied to all proxies in a location. These are stored in the DCS under the `global` designation. Each proxy also requires a name to exist as an instance, but no further customization is needed unless some setting needs a specific override.

This is because there are likely to be multiple proxies that have the same default configuration settings for the cluster, and repeating these values for every single proxy isn't necessary.

Additionally, when bootstrapping the proxy template, define at least one database for connection assignments. With these notes in mind, the format for this is as follows:

```
cluster:
  name: mycluster

proxies:
  monitor_interval: 5
  default_pool_size: 20
  max_client_conn: 1000
```

```

database_name: bdrdb
instances:
  - name:
proxy1
  - name:
proxy2
    default_pool_size: 50

```

This configures HARP for two proxies: `proxy1` and `proxy2`. Only `proxy2` has a custom `default_pool_size`, while using the global settings otherwise.

Assuming that file was named `proxy.yml`, you then apply it with the following:

```
harpctl apply proxy.yml
```

Once the proxy template is bootstrapped, it shows up with a quick examination:

```
> harpctl get proxies
```

Cluster	Name	Pool Mode	Auth Type	Max Client Conn	Default Pool Size
mycluster	global	session	md5	1000	20
mycluster	proxy1	session	md5	1000	20
mycluster	proxy2	session	md5	1000	50

3.6 HARP Manager

HARP Manager is a daemon that interacts with the local PostgreSQL/BDR node and stores information about its state in the consensus layer. Manager determines the node that currently holds leader status for a respective location and enforces configuration (lag, CAMO lag, etc.) constraints to prevent ineligible nodes from leader consideration.

Every Postgres node in the cluster must have an associated HARP Manager. Other nodes can exist, but they can't to participate as lead or shadow master roles or any other functionality that requires a HARP Manager.

Important

HARP Manager expects to be used to start and stop the database. Stopping HARP Manager will stop the database. Starting HARP Manager will start the database if it isn't already started. If another method is used to stop the database then HARP Manager will try and restart it.

How it works

Upon starting, HARP Manager uses `pg_ctl` to start Postgres if it isn't already running. After this, it periodically checks the server as defined by the `node.lease_refresh_interval` setting. HARP Manager collects various bits of data about Postgres including:

- The node's current LSN.
- If Postgres is running and accepting connections. This particular data point is considered a lease that must be periodically renewed. If it expires, HARP Proxy removes the node from any existing routing.
- The current apply LSN position for all upstream BDR peer nodes.
- If CAMO is enabled:
 - Name of the CAMO partner
 - Peer CAMO state (`is_ready`)
 - CAMO queue received and applied LSN positions
- Node type, such as whether the node is BDR or regular Postgres.

- The node's current role, such as a read/write, physical streaming replica, logical standby, and so on.
- BDR node state, which is `ACTIVE` except in limited cases.
- BDR Node ID for other metadata gathering.
- Other tracking values.

Important

When naming BDR nodes in HARP, the BDR node name must match the node name represented in the `node.name` configuration attribute. This occurs in the bootstrap process.

The data collected here is fully available to other HARP Manager processes and is used to evaluate lag, partner readiness, and other criteria that direct switchover and failover behavior.

After updating the node metadata, HARP Manager either refreshes the lead master lease if it's already held by the local node or seeks to obtain the lease if it's expired. Since the current state of all nodes is known to all other nodes, the node that was the previous lead master is given automatic priority ranking if present. If not, all other nodes list themselves by LSN lag, node priority, and other criteria, and the most qualified node seizes the lead master lease.

This procedure happens for every defined location where nodes are present. Thus for locations DC1 and DC2, there is a lead master node in each, with a separate lease and election process for both.

HARP Manager repeats these Postgres status checks, lease renewals, and elections repeatedly to ensure the cluster always has a lead master target for connections from HARP Proxy.

Configuration

HARP Manager expects the `dc`, `cluster`, and `manager` configuration stanzas. The following is a functional example:

```
cluster:
  name: mycluster

dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379

manager:
  name: node1
  postgres_bin_dir:
    /usr/lib/postgresql/13/bin
```

You can apply changes to the configuration file (default: `/etc/harp/config.yml`) by issuing `SIGHUP` to the running instance or by calling a service-level reload.

See [Configuration](#) for further details.

Usage

This is the basic usage for HARP Manager:

```
Usage of ./harp-manager:
  -f
  string
```

```

Optional path to config file
(shorthand)
--config
string
Optional path to config
file

```

There are no arguments to launch `harp-manager` as a forked daemon. This software is designed to be launched through `systemd` or in a container as a top-level process. This also means output is directed to `STDOUT` and `STDERR` for capture and access through `journald` or an attached container terminal.

Disabling and reenabling HARP Manager control of Postgres

You can temporarily pause HARP Manager control of Postgres. This results in a state where the daemon continues running but doesn't perform any operations that can affect existing behavior of the cluster. Reenabling management causes it to resume operation.

An example of temporarily disabling node management is:

```

harpctl unmanage node
node1

```

See [harpctl command-line tool](#) for more details.

Node management by HARP Manager is enabled by default.

3.7 HARP Proxy

HARP Proxy is a daemon that acts as an abstraction layer between the client application and Postgres. It interfaces with the consensus layer to obtain the identity of the current lead master node and directs traffic to that location. During a planned switchover or unplanned failover, it redirects to the new lead master node as dictated by the DCS.

You can select between `pgbouncer` or `builtin` for HARP Proxy. If you don't specify a proxy type, the default is `builtin`. When using `pgbouncer`, HARP Proxy is an interface layer between the DCS and PgBouncer. As such, PgBouncer is a prerequisite and must also be installed for HARP Proxy to fully manage its activity.

The builtin proxy doesn't require any additional software. When using builtin, HARP Proxy functions as a level 4 pass-through proxy.

Builtin proxy: how it works

Upon starting, HARP Proxy listens for incoming connections on the listening address and listening port specified in the bootstrap file per proxy instance. All application client traffic then passes through builtin proxy into the current lead master node for the location where this proxy is operating.

If the lead master lease isn't set, HARP Proxy disconnects all connection traffic until a new lead master is established. This also applies to circumstances when `harpctl promote` is used to invoke a planned transition to a new lead master. The disconnect is immediate.

Configuration

Choose the built-in proxy by setting the proxy type to `builtin`. The only other option that applies to the built-in proxy is `max_client_conn`, which specifies the maximum allowed client connections. If `max_client_conn` is higher than what the system can handle, it is lowered to a setting that's within the capability of the system that the proxy is on.

PgBouncer: how it works

Note

If you need more configurability of pgbouncer than what Harp Proxy provides, the recommended setup is to use builtin proxy and have pgbouncer point to it.

Upon starting, HARP Proxy launches PgBouncer if it's not already running and leaves client connections paused. After, it contacts the DCS to determine the identity of the lead master, configure PgBouncer to use this as the target for database connections, and resume connection activity. All application client traffic then passes through PgBouncer into the current lead master node for the location where this proxy is operating.

While PgBouncer is running, HARP Proxy checks its status based on the `monitor_interval` configuration setting in the DCS and stores it in the DCS for monitoring purposes. This configuration allows interrogation with `harpctl` to retrieve status of all configured proxies or any one proxy.

If the lead master lease isn't set, HARP Proxy pauses all connection traffic until a new lead master is established. This also applies to circumstances when `harpctl promote` is used to invoke a planned transition to a new lead master. It uses a PgBouncer `PAUSE` command for this, so existing sessions are allowed to complete any pending transactions before they're held in stasis.

PgBouncer configuration file

When HARP Proxy uses PgBouncer for connection management and redirection, a `pgbouncer.ini` file must exist. HARP Manager builds this file based on various runtime directives as defined in [Proxy directives](#).

This file is located in the same folder as the `config.yml` used by HARP Proxy. Any PgBouncer process launched by HARP Proxy uses this configuration file, and you can use it for debugging or information purposes. Modifications to this automatically generated `pgbouncer.ini` file are lost any time HARP Proxy restarts, so use `harpctl set proxy` to alter these settings instead. Calling `harpctl set proxy` doesn't update the `pgbouncer.ini` file until the proxy restarts.

Disabling and reenabling HARP Proxy node management

You can temporarily pause HARP Proxy control of PgBouncer. This results in a state where the daemon continues running but doesn't perform any operations that can affect existing behavior of the cluster. Reenabling management causes it to resume operation.

An example of temporarily disabling management of a specific proxy is:

```
harpctl unmanage proxy
proxy1
```

See [harpctl command-line tool](#) for more details.

Proxy node management is enabled by default.

Passthrough user authentication

With pgbouncer, we strongly recommend configuring HARP Proxy to use the `auth_user` and `auth_query` runtime directives. If these aren't set, the PgBouncer `userlist.txt` file must include username and password hash combinations for every user PgBouncer needs to authenticate on behalf of Postgres.

Do *not* use the `pgbouncer` user, as this is used by HARP Proxy as an admin-level user to operate the underlying PgBouncer service.

Configuration

HARP Proxy expects the `dc`, `cluster`, and `proxy` configuration stanzas. The following is a functional example:

```
cluster:
  name: mycluster

dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379

proxy:
  name:
  proxy1
```

Each proxy connects to the DCS to retrieve the hosts and ports to listen on for connections.

Usage

This is the basic usage for HARP Proxy:

```
Usage of ./harp-proxy:
  -f
string
    Optional path to config file
(shorthand)
  --config
string
    Optional path to config
file
```

There are no arguments to launch `harp-proxy` as a forked daemon. This software is designed to be launched through `systemd` or in a container as a top-level process. This also means output is directed to `STDOUT` and `STDERR` for capture and access through `journald` or an attached container terminal.

3.8 harpctl command-line tool

`harpctl` is a command-line tool for directly manipulating the consensus layer contents to fit desired cluster geometry. You can use it to, for example, examine node status, "promote" a node to lead master, disable/enable cluster management, bootstrap cluster settings, and so on.

Synopsis

```
$ harpctl --help

Usage:
  harpctl [command]

Available Commands:
  apply      Command to set up a
cluster
  completion generate the autocompletion script for the specified
shell
  fence      Fence specified node
```

```

  get          Command to get
resources
  help        Help about any
command
  manage      Manage
Cluster
  promote     Promotes the sepcified node to be
primary,
  set         Command to set
resources.
  unfence     unfence specified node
unmanage     Unmanage
Cluster
  version     Command to get version
information

```

Flags:

```

-c, --cluster string    name of
cluster.
-f, --config-file string  config file (default is
/etc/harp/config.yml)
-h, --help              help for
harpctl
-o, --output string      'json,
yaml'.
-t, --toggle            Help message for
toggle

```

Use `"harpctl [command] --help"` for more information about a command.

In addition to this basic synopsis, each of the available commands has its own series of allowed subcommands and flags.

Configuration

`harpctl` must interact with the consensus layer to operate. This means a certain minimum amount of settings must be defined in `config.yml` for successful execution. This includes:

- `dc.driver`
- `dc.endpoints`
- `cluster.name`

As an example using etcd:

```

cluster:
  name: mycluster
dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379

```

See [Configuration](#) for details.

Execution

Execute `harpctl` like this:

```
harpctl command [flags]
```

Each command has its own series of subcommands and flags. Further help for these are available by executing this command:

```
harpctl <command> --
help
```

harpctl apply

You must use an `apply` command to "bootstrap" a HARP cluster using a file that defines various attributes of the intended cluster.

Execute an `apply` command like this:

```
harpctl apply <filename>
```

This essentially creates all of the initial cluster metadata, default or custom management settings, and so on. This is done here because the DCS is used as the ultimate source of truth for managing the cluster, and this makes it possible to change these settings dynamically.

This can either be done once to bootstrap the entire cluster, once per type of object, or even on a per-node basis for the sake of simplicity.

This is an example of a bootstrap file for a single node:

```
cluster:
  name: mycluster
nodes:
  - name: firstnode
    dsn: host=host1 dbname=bdrdb
    user=harp_user
    location:
      dc1
    priority: 100
    db_data_dir: /db/pgdata
```

As seen here, it is good practice to always include a cluster name preamble to ensure all changes target the correct HARP cluster, in case several are operating in the same environment.

Once `apply` completes without error, the node is integrated with the rest of the cluster.

Note

You can also use this command to bootstrap the entire cluster at once since all defined sections are applied at the same time. However, we don't encourage this use for anything but testing as it increases the difficulty of validating each portion of the cluster during initial definition.

harpctl fence

Marks the local or specified node as fenced. A node with this status is essentially completely excluded from the cluster. HARP Proxy doesn't send it traffic, its representative HARP Manager doesn't claim the lead master lease, and further steps are also taken. If running, HARP Manager stops Postgres on the node as well.

Execute a `fence` command like this:


```
harpctl fence (<node-name>)
```

The node-name is optional; if omitted, `harpctl` uses the name of the locally configured node.

```
harpctl get
```

Fetches information stored in the consensus layer for various elements of the cluster. This includes nodes, locations, the cluster, and so on. The full list includes:

- `cluster` — Returns the cluster state.
- `leader` — Returns the current or specified location leader.
- `location` — Returns current or specified location information.
- `locations` — Returns list of all locations.
- `node` — Returns the specified Postgres node.
- `nodes` — Returns list of all Postgres nodes.
- `proxy` — Returns current or specified proxy information.
- `proxies` — Returns list of all proxy nodes.

```
harpctl get cluster
```

Fetches information stored in the consensus layer for the current cluster:

```
> harpctl get cluster
```

```
Name      Enabled
-----  -
mycluster true
```

```
harpctl get leader
```

Fetches node information for the current lead master stored in the DCS for the specified location. If no location is passed, `harpctl` attempts to derive it based on the location of the current node where it was executed.

Example:

```
> harpctl get leader
```

```
dc1
```

```
Cluster  Name   Ready Role   Type Location Fenced Lock
Duration
-----  -
mycluster mynode true  primary bdr  dc1      false
30
```

```
harpctl get location
```

Fetches location information for the specified location. If no location is passed, `harpctl` attempts to derive it based on the location of the current node where it was executed.

Example:

```
> harpctl get location
dc1

Cluster  Location Leader Previous Leader Target Leader Lease
Renewals
-----  -----
mycluster dc1      mynode mynode
<nil>
```

harpctl get locations

Fetches information for all locations currently present in the DCS.

Example:

```
> harpctl get locations

Cluster  Location Leader Previous Leader Target Leader Lease
Renewals
-----  -----
mycluster dc1      mynode mynode
<nil>
mycluster dc2      thatnode thatnode
<nil>
```

harpctl get node

Fetches node information stored in the DCS for the specified node.

Example:

```
> harpctl get node
mynode

Cluster  Name Location Ready Fenced Allow Routing Routing Status Role Type Lock
Duration
-----  -----
mycluster mynode dc1      true  false true      ok      primary bdr
30
```

harpctl get nodes

Fetches node information stored in the DCS for the all nodes in the cluster.

Example:

```
> harpctl get nodes

Cluster  Name Location Ready Fenced Allow Routing Routing Status Role Type Lock
Duration
-----  -----
myclusters bdra1 dc1      true  false true      ok      primary bdr
30
myclusters bdra2 dc1      true  false false     N/A     primary bdr
30
```

```
myclusters bdra3 dc1      true false false      N/A      primary bdr
30
```

harpctl get proxy

Fetches proxy information stored in the DCS for specified proxy. Specify `global` to see proxy defaults for this cluster.

Example:

```
> harpctl get proxy
proxy1

Cluster  Name  Pool Mode Auth Type Max Client Conn Default Pool
Size
-----  -
mycluster proxy1 session md5      1000
20
```

harpctl get proxies

Fetches proxy information stored in the DCS for all proxies in the cluster. Additionally, lists the `global` pseudo-proxy for default proxy settings.

Example:

```
> harpctl get proxies

Cluster  Name  Pool Mode Auth Type Max Client Conn Default Pool
Size
-----  -
mycluster global session md5      500
25
mycluster proxy1 session md5      1000
20
mycluster proxy2 session md5      1500
30
```

harpctl manage

If a cluster isn't in a managed state, instructs all HARP Manager services to resume monitoring Postgres and updating the consensus layer. Do this after maintenance is complete following HARP software updates or other significant changes that might affect the whole cluster.

Execute a `manage` command like this:

```
harpctl manage
cluster
```

Note

Currently you can enable or disable cluster management only at the `cluster` level. Later versions will also make it possible to do this for individual nodes or proxies.

harpctl promote

Promotes the next available node that meets leadership requirements to lead master in the current Location. Since this is a requested event, it invokes a smooth handover where:

1. The existing lead master releases the lead master lease, provided:
 - If CAMO is enabled, the promoted node must be up to date and CAMO ready, and the CAMO queue must have less than `node.maximum_camo_lag` bytes remaining to be applied.
 - Replication lag between the old lead master and the promoted node is less than `node.maximum_lag`.
2. The promoted node is the only valid candidate to take the lead master lease and does so as soon as it is released by the current holder. All other nodes ignore the unset lead master lease.
 - If CAMO is enabled, the promoted node temporarily disables client traffic until the CAMO queue is fully applied, even though it holds the lead master lease.
3. HARP Proxy, if using pgbouncer, will `PAUSE` connections to allow ongoing transactions to complete. Once the lead master lease is claimed by the promoted node, it reconfigures PgBouncer for the new connection target and resumes database traffic. If HARP Proxy is using the builtin proxy, it terminates existing connections and creates new connections to the lead master as new connections are requested from the client.

Execute a `promote` command like this:

```
harpctl promote (<node-name>)
```

Provide the `--force` option to forcibly set a node to lead master, even if it doesn't meet the criteria for becoming lead master. This circumvents any verification of CAMO status or replication lag and causes an immediate transition to the promoted node. This is the only way to specify an exact node for promotion.

The node must be online and operational for this to succeed. Use this option with care.

harpctl set

Sets a specific attribute in the cluster to the supplied value. This is used to tweak configuration settings for a specific node, proxy, location, or the cluster rather than using `apply`. You can use this for the following object types:

- `cluster` — Sets cluster-related attributes.
- `location` — Sets specific location attributes.
- `node` — Sets specific node attributes.
- `proxy` — Sets specific proxy attributes.

harpctl set cluster

Sets cluster-related attributes only.

Example:

```
harpctl set cluster
event_sync_interval=200
```

harpctl set node

Sets node-related attributes for the named node. Any options mentioned in [Node directives](#) are valid here.

Example:

```
harpctl set node mynode
priority=500
```

`harpctl set proxy`

Sets proxy-related attributes for the named proxy. Any options mentioned in the [Proxy directives](#) are valid here. Properties set this way require a restart of the proxy before the new value takes effect.

Example:

```
harpctl set proxy proxy1
max_client_conn=750
```

Use `global` for cluster-wide proxy defaults:

```
harpctl set proxy global
default_pool_size=10
```

`harpctl unfence`

Removes the `fenced` attribute from the local or specified node. This removes all previously applied cluster exclusions from the node so that it can again receive traffic or hold the lead master lease. Postgres is also started if it isn't running.

Execute an `unfence` command like this:

```
harpctl unfence (<node-name>)
```

The node-name is optional. If you omit it, `harpctl` uses the name of the locally configured node.

`harpctl unmanage`

Instructs all HARP Manager services in the cluster to remain running but no longer actively monitoring Postgres, or modify the contents of the consensus layer. This means that any ordinary failover event such as a node outage doesn't result in a leadership migration. This is intended for system or HARP maintenance prior to making changes to HARP software or other significant changes to the cluster.

Execute an `unmanage` command like this:

```
harpctl unmanage
cluster
```

Note

Currently you can enable or disable cluster management at only the `cluster` level. Later versions will also make it possible to do this for individual nodes or proxies.

3.9 Consensus layer considerations

HARP is designed so that it can work with different implementations of consensus layer, also known as Distributed Control Systems (DCS).

Currently the following DCS implementations are supported:

- etcd
- BDR

This information is specific to HARP's interaction with the supported DCS implementations.

BDR driver compatibility

The `bdr` native consensus layer is available from BDR versions [3.6.21](#) and [3.7.3](#).

For the purpose of maintaining a voting quorum, BDR Logical Standby nodes don't participate in consensus communications in a EDB Postgres Distributed cluster. Don't count these in the total node list to fulfill DCS quorum requirements.

Maintaining quorum

Clusters of any architecture require at least $n/2 + 1$ nodes to maintain consensus via a voting quorum. Thus a three-node cluster can tolerate the outage of a single node, a five-node cluster can tolerate a two-node outage, and so on. If consensus is ever lost, HARP becomes inoperable because the DCS prevents it from deterministically identifying the node that is the lead master in a particular location.

As a result, whichever DCS is chosen, more than half of the nodes must always be available *cluster-wide*. This can become a non-trivial element when distributing DCS nodes among two or more data centers. A network partition prevents quorum in any location that can't maintain a voting majority, and thus HARP stops working.

Thus an odd-number of nodes (with a minimum of three) is crucial when building the consensus layer. An ideal case distributes nodes across a minimum of three independent locations to prevent a single network partition from disrupting consensus.

One example configuration is to designate two DCS nodes in two data centers coinciding with the primary BDR nodes, and a fifth DCS node (such as a BDR witness) elsewhere. Using such a design, a network partition between the two BDR data centers doesn't disrupt consensus thanks to the independently located node.

Multi-consensus variant

HARP assumes one lead master per configured location. Normally each location is specified in HARP using the `location` configuration setting. By creating a separate DCS cluster per location, you can emulate this behavior independently of HARP.

To accomplish this, configure HARP in `config.yml` to use a different DCS connection target per desired Location.

HARP nodes in DC-A use something like this:

```
location:
  dca
dcs:
  driver: etcd
  endpoints:
    - dcs-a1:2379
    - dcs-a2:2379
    - dcs-a3:2379
```

While DC-B uses different hostnames corresponding to nodes in its canonical location:

```
location:
dcb
dcs:
  driver: etcd
  endpoints:
    - dcs-a1:2379
    - dcs-a2:2379
    - dcs-a3:2379
```

There's no DCS communication between different data centers in this design, and thus a network partition between them doesn't affect HARP operation. A consequence of this is that HARP is completely unaware of nodes in the other location, and each location operates essentially as a separate HARP cluster.

This isn't possible when using BDR as the DCS, as BDR maintains a consensus layer across all participant nodes.

A possible drawback to this approach is that `harpctl` can't interact with nodes outside of the current location. It's impossible to obtain node information, get or set the lead master, or perform any other operation that targets the other location. Essentially this organization renders the `--location` parameter to `harpctl` unusable.

TPAexec and consensus

These considerations are integrated into TPAexec as well. When deploying a cluster using etcd, it constructs a separate DCS cluster per location to facilitate high availability in favor of strict consistency.

Thus this configuration example groups any DCS nodes assigned to the `first` location together, and the `second` location is a separate cluster:

```
cluster_vars:
  failover_manager: harp
  harp_consensus_protocol: etcd

locations:
  - Name: first
  - Name:
second
```

To override this behavior, configure the `harp_location` implicitly to force a particular grouping.

Thus this example returns all etcd nodes into a single cohesive DCS layer:

```
cluster_vars:
  failover_manager: harp
  harp_consensus_protocol: etcd

locations:
  - Name: first
  - Name:
second
  - Name: all_dcs

instance_defaults:
  vars:
    harp_location: all_dcs
```

The `harp_location` override might also be necessary to favor specific node groupings when using cloud providers such as Amazon that favor availability zones in regions over traditional data centers.

3.10 Security and roles

Beyond basic package installation and configuration, HARP requires Postgres permissions to operate. These allow it to gather information about Postgres or BDR as needed to maintain node status in the consensus layer.

Postgres permissions

Create the role specified in the `node.dsn` parameter in one of the following ways:

- `CREATE USER ...`
- `CREATE ROLE ... WITH LOGIN`

This syntax ensures the role can log into the database to gather diagnostic information.

Similarly, an entry must exist in `pg_hba.conf` for this role. You can do this in many ways. As an example, consider a VPN subnet where all database hosts are located somewhere in `10.10.*`. In such a case, the easiest approach is to add a specific line:

#	TYPE	DATABASE	USER	ADDRESS	METHOD
	hostssl	all	harp_user	10.10.1.1/16	scram-sha-256

Note

In this case we've used the more modern `scram-sha-256` authentication rather than `md5`, which is now deprecated. We've also elected to require SSL authentication by specifying `hostssl`.

BDR permissions

BDR nodes have metadata and views that are visible only when certain roles are granted to the HARP-enabled user. In this case, the HARP user requires the following:

```
GRANT bdr_monitor TO
harp_user;
```

The `bdr_monitor` BDR role is meant for status monitoring tools to maintain ongoing information on cluster operation, thus it is well-suited to HARP.

BDR consensus permissions

When the `dcs.driver` configuration parameter is set to `bdr`, HARP uses BDR as the consensus layer. As such, it requires access to API methods that are currently available only to the `bdr_superuser` role. This means the HARP-enabled user requires the following:

```
GRANT bdr_superuser TO foobar;
```

Currently access to the BDR consensus model requires superuser equivalent permission.

Important

BDR superusers *are not* Postgres superusers. The `bdr_superuser` role is merely granted elevated privileges in BDR, such as access to restricted functions, tables, views, and other objects.

4 pglogical 3.7

The pglogical 3.7 extension provides logical streaming replication for PostgreSQL, using a publish/subscribe model. It is based on technology developed as part of the BDR3 project.

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

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

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

pglogical is new technology utilizing the latest in-core features, so we require both provider and subscriber nodes to run PostgreSQL 10 or newer.

Use cases supported are:

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

Architectural details:

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

4.1 Release Notes for pglogical 3.7

pglogical 3.7.23

This is a maintenance release for pglogical 3.7 which includes minor improvements as well as fixes for issues identified previously.

Resolved Issues

- Fix row filter failure for partitions created after a table column was dropped (RT95149).

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.7.9 and higher
- 3.6.29 and higher
- 2.4.0 and 2.4.

pglogical 3.7.22

This is a maintenance release for pglogical 3.7 which includes minor improvements as well as fixes for issues identified previously.

Resolved Issues

- Fixed a bug in handling of memory context in presence of triggers (RT95948).
- Ensured that table slots are allocated in query context and not in per-tuple context
- Fixed a bug during binary upgrade to avoid issue during hook execution.

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.7.9 and higher
- 3.6.29 and higher
- 2.4.0 and 2.4.1

pglogical 3.7.21

This is a maintenance release for pglogical 3.7 which includes minor improvements as well as fixes for issues identified previously.

Resolved Issues

- Fix join and replication when SDW and standby_slot_names are set (BDR-3176, RT89702, RT89536)
- Fix writer_name resolution in pglogical.alter_subscription_writer_options (BDR-3429, RT90679) When a writer_name other than subscription_name is specified as the second argument to pglogical.alter_subscription_writer_options, the function throws an error "writer for subscription not found". Fix the function match the writer_name correctly.

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.7.9 and higher
- 3.6.29 and higher
- 2.4.0 and 2.4.1

pglogical 3.7.20

This is a maintenance release for PGLogical 3.7. It is equivalent to 3.7.19, but still gets a release and a version bump to match the BDR version number.

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.7.9 and higher
- 3.6.29 and higher
- 2.4.0 and 2.4.1

pglogical 3.7.19

This is a maintenance release for pglogical 3.7 which includes minor improvements as well as fixes for issues identified previously.

Resolved Issues

- Adjust docs related to `drop_subscription` (RT86700, BDR-2705) The function accepts 3 arguments, specify it in the documentation.

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.7.9 and higher
- 3.6.29 and higher
- 2.4.0 and 2.4.1

pglogical 3.7.18

This is a maintenance release for pglogical 3.7 which includes minor improvements as well as fixes for issues identified previously.

Resolved Issues

- Add Windows specific workaround for `wal_sender_timeout`
- Resolve hang related to `standby_slots_min_confirmed` (BDR-2445, RT80692) Allow the WAL sender to proceed without the need to wait for the timeout after a Subscriber Only node is parted, while using `standby_slot_names`.
- Don't replicate TRUNCATE as global message (BDR-2821, RT87453) The TRUNCATE command now takes the replication set into account.

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.7.9 and higher
- 3.6.29 and higher
- 2.4.0 and 2.4.1

pglogical 3.7.17

This is a maintenance release for pglogical 3.7 which includes minor improvements as well as fixes for issues identified previously.

Resolved Issues

- Fix spurious pglogical receiver timeouts on idle connections (RT82315) When there is nothing to replicate and apply, the pglogical receiver may misinterpret the state as timeout and exit with error "ERROR: 08006: terminating pglogical receiver due to timeout". Correct this misinterpretation to prevent spurious restarts of the receiver and writer processes.

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.7.9 and higher
- 3.6.29 and higher
- 2.4.0 and 2.4.1

pglogical 3.7.16

This is a maintenance release for pglogical 3.7 which includes minor improvements as well as fixes for issues identified previously.

Resolved Issues

- Keep the `lock_timeout` as configured on non-CAMO-partner BDR nodes (BDR-1916) A CAMO partner uses a low `lock_timeout` when applying transactions from its origin node. This was inadvertently done for all BDR nodes rather than just the CAMO partner, which may have led to spurious `lock_timeout` errors on pglogical writer processes on normal BDR nodes.
- Prevent walsender processes spinning when facing lagging standby slots (RT80295, RT78290) Correct signaling to reset a latch so that a walsender process does not consume 100% of a CPU in case one of the standby slots is lagging behind.
- Provide a proper error when attempting to use `pglogical.use_spi` (RT76368)
- Reduce log information when switching between writer processes (BDR-2196)
- Eliminate a memory leak when replicating partitioned tables (RT80981, BDR-2194)

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.7.9 and higher
- 3.6.29 and higher
- 2.4.0 and 2.4.1

pglogical 3.7.15

This is a maintenance release for pglogical 3.7 which includes minor improvements as well as fixes for issues identified previously.

Improvements

- Add `pglogical.max_worker_backoff_delay` (BDR-1767) This changes the handling of the backoff delay to exponentially increase from `pglogical.min_worker_backoff_delay` to `pglogical.max_worker_backoff_delay` in presence of repeated errors. This reduces log spam and in some cases also prevents unnecessary connection attempts.

Resolved Issues

- For the view `pglogical.subscriptions`, ensure the correct value is displayed in the `provider_dsn` column (BDR-1859) Previously the subscriber node's DSN was being displayed.
- Fix handling of `wal_receiver_timeout` (BDR-1848) The `wal_receiver_timeout` has not been triggered correctly due to a regression in BDR 3.7 and 4.0.
- Limit the `pglogical.standby_slot_names` check when reporting flush position only to physical slots (RT77985, RT78290) Otherwise flush progress is not reported in presence of disconnected nodes when using `pglogical.standby_slot_names`.
- Confirm LSN of LCR slot progress records when recovering LCR segments (BDR-1264)
- Fix replication of data types created during bootstrap (BDR-1784)

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.7.9 and higher
- 3.6.29 and higher
- 2.4.0 and 2.4.1

pglogical 3.7.14

This is a maintenance release for pglogical 3.7 which includes minor improvements as well as fixes for issues identified previously.

Improvements

- Implement buffered read for LCR segment file (BDR-1422) This should reduce both I/O operations done by decoding worker as well as the amount of signalling between the decoding worker and wal senders which results in improved performance when decoding worker is used.

Resolved Issues

- Invalidate output plugin's replication set cache when published replication sets change (BDR-1715) This ensures that published replicated sets are always up to date when user changes what should be published on node level. Before this fix, the old settings would have been applied until downstream has reconnected. This primarily affects BDR in situations where the `bdr.alter_node_replication_sets` was called on multiple nodes in quick succession.

- Don't try to retry transaction replication when writer receives SIGTERM or SIGINT This would unnecessarily hold back the shutdown of postgres when parallel apply is enabled.
- Log correct error message during initial schema synchronization error (BDR-1598) Previously we'd always log ERROR with message "Success" which is not helpful for troubleshooting.

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.7.9 and higher
- 3.6.29 and higher
- 2.4.0
- 2.4.1

pglogical 3.7.13

This is a maintenance release for pglogical 3.7 which includes minor improvements as well as fixes for issues identified previously.

Improvements

- Allow the BDR consensus process to use a (sub) replorigin (BDR-1613) For parallel apply, multiple writers could use individual origins that would be merged, thus act kind of as child origins for a main or parent origin. Extend this mechanism to allow the BDR consensus process to also register an origin participating in the group.

Resolved Issues

- Fix the replication protocol handling of some strings (RT74123) Certain strings like e.g. relation schema, relation name, attribute name, and message prefix were mistakenly changed to not include a terminating NULL character. This led to a protocol incompatibility when interoperating with PGLogical 3.5.3 and older or when WAL messages are used (e.g. due to the use of BDR features CAMO or Eager All Node Replication).
- Fix segfault when creating a subscription with the RabbitMQ writer plugin (BDR-1482, RT72434)
- Don't wait on own replication slot when waiting for `standby_slot_names` (RT74036) The walsenders that use slots named in `standby_slot_names` should not wait for anything, otherwise it might wait forever.
- Reduce logging when WAL decoder is not available (BDR-1041) `pgl_xact_fully_decoded()` will log a message when it finds that the WAL decoder is not running. WAL decoder may not start for long duration and thus this log line will repeat and increase the log file size. There are other ways to know whether WAL decoder is running. So this line is not required.
- Handle deprecated `update_differing` and `update_recently_deleted` (BDR-1610, RT74973) While upgrading to 3.7, take care of the deprecated conflict types otherwise, as they would be unknown to the new version, it would not know how to handle them and break the upgrade process.
- Enable async conflict resolution for explicit 2PC (BDR-1609, RT71298) Continue applying the transaction using the async conflict resolution for explicit two phase commit.

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.7.9 and higher
- 3.6.28
- 2.4.0
- 2.4.1

pglogical 3.7.12

This is a maintenance release for pglogical 3.7 which includes minor improvements as well as fixes for issues identified previously.

Improvements

- Add replication status monitoring (BDR-865) Track the connection establishment and drops, the number of transactions committed as well as the restart LSN to detect repeated restarts and diagnose stalled replication.
- Improve performance when used in combination with synchronous replication (BDR-1398) Override `synchronous_commit` to `local` for all PGLogical processes performing only bookkeeping transactions which do not need to be replicated. Only the PGLogical writer processes applying user transactions need to follow the `synchronous_commit` mode configured for synchronous replication.
- Internal improvements and additional hooks to better support BDR

Resolved Issues

- Performance improvements for Decoding Worker (BDR-1311, BDR-1357) Speed up lookups of the WAL decoder worker, reduce delays after reaching up to the LSN previously known to be decoded by the Decoding Worker, and reduce number of system calls when writing one LCR chunk to an LCR segment file.
- Improve compatibility with Postgres 13 (BDR-1396) Adjust to an API change in `ReplicationSlotAcquire` that may have led to unintended blocking when non-blocking was requested and vice versa. This version of PGLogical eliminates this potential problem, which has not been observed on production systems so far.

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.7.9 and higher
- 3.6.27
- 2.4.0

pglogical 3.7.11

This is a maintenance release for pglogical 3.7 which includes minor improvements as well as fixes for issues identified previously.

Resolved Issues

- Add protection against malformed parameter values in pgl output plugin This fixes potential crash when some parameters sent to the output plugin were malformed.
- Get copy of slot tuple when logging conflict (BDR-734) Otherwise we could materialize the row early causing wrong update in presence of additional columns on the downstream.
- Use a separate memory context for processing LCRs (BDR-1237, RT72165) This fixes memory leak when using the decoding worker feature of BDR.
- Truncate LCR segment file after recovery (BDR-1236, BDR-1259) This fixes memory errors reported by the decoding worker after crash.
- Ensure `pg_read_and_filter_lcr` will exit when postmaster dies (BDR-1226, BDR-1209, RT72083) Solves issues with hanging decoder worker on shutdown.
- Fix memory leak in the pglogical COPY handler (BDR-1219, RT72091) This fixes memory leak when synchronizing large tables.
- Allow binary and internal protocol on more hardware combinations. This currently only affects internal testing.

Upgrades

This release supports upgrading from following versions of pglogical:

- 3.7.9 and higher
- 3.6.27
- 2.4.0

pglogical 3.7.10

This is a maintenance release for pglogical 3.7 which includes minor improvements as well as fixes for issues identified previously.

Improvements

- Windows support improvements (BDR-976, BDR-1083)

Resolved Issues

- Fix potential crash during cleanup of bulk copy replication (BDR-1168)
- Fix issues in generic WAL message handling when the WAL message was produced by something other than pglogical (BDR-670)
- Redefine `werr_age` of `pglogical.worker_error_summary` to report correct age
- Only use key attributes of covering unique index when used as replica identity This only affects what is being sent over network, no logic change.

pglogical 3.7.9

This is a maintenance release for pglogical 3.7 which includes minor improvements as well as fixes for issues identified previously.

Improvements

- Support two-phase commit transaction with Decoding Worker (BDR-811)

A two-phase commit transaction is completely decoded and sent downstream when processing its PREPARE WAL record. COMMIT/ROLLBACK PREPARED is replicated separately when processing the corresponding WAL record.

- Reduce writes to `pg_replication_origin` when using parallel apply (RT71077) Previously, especially on EPAS pglogical could produce thousands of dead rows in `pg_replication_origin` system catalog if it had connection problems to upstream.

Resolved Issues

- Fix flush queue truncation (BDR-890) During queue compaction flush to correct LSN instead of always truncating whole flush queue.
- Fix the `pglogical.worker_error` columns `werr_worker_index` and `werr_time` Used to report wrong values.
- Fix snapshot handling in internal executor usage (BDR-904) Row filtering didn't correctly pushed snapshot in some situations.

Caught thanks to upcoming change in PostgreSQL that double checks for this.

- Fix handling of domains over arrays and composite types (BDR-29) The replication protocol previously could not always handle columns using domains over arrays and domains over composite types.

pglogical 3.7.8

This is a first stable release of the pglogical 3.7.

pglogical 3.7 is a major release of pglogical. This release includes major new features as well as smaller enhancements.

Upgrades are supported from pglogical 3.6.25 and 3.7.7 in this release.

Improvements

- Allow parallel apply on EDB Advanced Server (EE)

Resolved Issues

- Fix divergence after physical failover (BDR-365, RT68894 and RM19886) Make sure that we don't report back LSN on subscriber that hasn't been received by named standbys (`pglogical.standby_slot_names`).

This will ensure provider side keeps slot position behind far enough so that if subscriber is replaced by one of said named standbys, the standby will be able to fetch any missing replication stream from the original provider.

- Fix crash when ERROR happened during fast shutdown of pglogical writer
- Don't re-enter worker error handling loop recursively (BDR-667) This should help make what happens clearer in any cases where we do encounter errors during error processing.
- Assign collation to the index scan key (BDR-561) When doing lookups for INSERT/UPDATE/DELETE, either to find conflicts or key for the operation to

be applied, we should use correct collation.

This fixes index lookups for index on textual fields on Postgres 12+.

- Use `name` data type for known fixed length fields (BDR-561) This solves potential index collation issues with pglogical catalogs.
- Progress WAL sender's slot based on WAL decoder input (BDR-567) Without slot progression the server will eventually stop working.
- Fix begin of transaction write when LCR file does not have enough space (BDR-606)
- Restart decoding a transaction that was not completed in single decoding worker (BDR-247) If we crashed during a long transaction that spawns on more than one lcr file we start decoding again from the lsn of the beginning of the transaction and find the last lcr file where we can write into.
- Generate temp slot with correct length in subscription sync Otherwise the name of the slot might be shortened by Postgres leading to confusion on the slot cleanup.
- Improve detection of mixing temporary and nontemporary objects in DDLs (BDR-485) These can break replication so it's important to not allow them.
- Fix pre-commit message handling in Eager replication (BDR-492)
- Override GUCs in all pglogical workers, not just in writers.

pglogical 3.7.7

This is a beta release of the pglogical 3.7.

pglogical 3.7 is a major release of pglogical. This release includes major new features as well as smaller enhancements.

Upgrades are supported from pglogical 3.6.25 and 3.7.6 in this release.

Improvements

- Use parallel apply during initial sync during logical join
- Add worker process index to the worker_error catalog (BDR-229) The column `werr_worker_index` in worker_error table keeps track of the writers for the same subscription.
- Various improvements for WAL decoder/sender coordination (BDR-232, BDR-335)
- Name LCR segment similar to XLOG segments (BDR-236, BDR-253, BDR-321, BDR-322) An LCR segment file is identified by five 8-digit hex numbers. Timeline first group, XLOG the next two groups and file number the last two groups.

Resolved Issues

- Restrict adding queue table to the replication set. (EDBAS, EBD-45)
- Fix deadlock between receiver and writer during queue flush (BDR-483)
- Force and wait for all writers to exit when one writer dies (BDR-229)

- Name LCR directory after the replication slot (BDR-60) Logical output plugin may be used by multiple replication slots. Store the LCRs from a given replication slot in a directory named after that replication slot to avoid mixing LCRs for different slots.
- Fix EXPLAIN...INTO TABLE replication issue (EBC-46)

pglogical 3.7.6

This is a beta release of the pglogical 3.7.

pglogical 3.7 is a major release of pglogical. This release includes major new features as well as smaller enhancements.

Upgrades are supported from pglogical 3.6.25 in this release.

Improvements

- Enable parallel apply for CAMO and Eager (RM17858)
- Improve error handling in Eager/CAMO, especially with parallel apply (BDR-106)
- Reduce severity of Eager/CAMO feedback errors
- Add infrastructure necessary for allowing separation of WAL decoding from WalSender process in BDR (RM18868, BDR-51, BDR-60)

Resolved Issues

- Improve relcache invalidation handling in heap writer This should solve missed invalidations after opening table for DML apply.
- Wait for the writer that has XID assigned rather than one that doesn't (BDR-137) This fixes deadlock in parallel apply when there is a mix of empty and non-empty transactions where the non-empty ones conflict.
- Correct writer state tracking for Eager/CAMO (BDR-107)
- Correct and improve CAMO misconfiguration handling (BDR-105) Properly abort the transaction in case of sync CAMO, so the internal state of the PGL writer is cleared
- Fix transaction state tracking for CAMO/Eager Do not abort the transaction at pre-commit time for non-CAMO nodes, but keep it open until the final commit. Adjust the transaction state tracking accordingly
- Fix MERGE handling in 2ndQPostgres 12 and 13 We'd before allow the MERGE command on replicated tables without appropriate REPLICIA IDENTITY which could result in broken replication.
- Fix CAMO feedback sending (RM17858) Fixes stalls in CAMO feedback, improving performance compared to previous 3.7 beta releases. This is especially visible with parallel apply enabled.

pglogical 3.7.5

This is a beta release of the pglogical 3.7.

pglogical 3.7 is a major release of pglogical. This release includes major new features as well as smaller enhancements.

Improvements

- Optimize utility command processing (RT69617) For commands that won't affect any DB objects and don't affect pglogical we can skip the processing early without reading any pglogical or system catalogs or calling to DDL replication plugin interfaces. This is optimization for systems with large number of such utility command calls (that is primarily applications that do explicit transaction management).
- Add upgrade path from pglogical 2.

Resolved Issues

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

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

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

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

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

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

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

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

- Fix `pglogical_create_subscriber` when "-v" is passed. It will make `pg_ctl` emit meaningful information, making it easier to debug issues where `pg_ctl` fails

pglogical 3.7.4

This is a beta release of the pglogical 3.7.

pglogical 3.7 is a major release of pglogical. This release includes major new features as well as smaller enhancements.

Improvements

- Support PostgreSQL 13
- Add `pglogical.replication_origin_status` view (RM17074) Same as `pg_replication_origin_status` but does not require superuser permissions to access it.
- Beta support of upgrades from 3.6 (currently from 3.6.22)
- Improved SystemTAP support

Resolved Issues

- Fix race condition in replication table filtering which could cause crash (RM18839) The cached info about table might get invalidated while used which would crash the backend during one of the following operations:
 - reading the `pglogical.tables` view
 - new subscription creation
 - table resynchronization
- Don't do transparent DDL replication on commands that affect temporary objects (RM19491, RT69170) These are likely to not exist on the subscription.
- Only run pgl specific code in deadlock detector when inside writer (RM18402) It's not relevant in user backends and would cause ERRORS there.

pglogical 3.7.3

This is a beta release of the pglogical 3.7.

pglogical 3.7 is a major release of pglogical. This release includes major new features as well as smaller enhancements.

Improvements

- Parallel Apply (RM6503) Allows configuring number of writers per subscriptions. By default this is still 1, which mean parallel apply is off by default but can be enabled both globally (`pglogical.writers_per_subscription`) and per subscription (`num_writers` option in `pglogical.create_subscription()` and `pglogical.alter_subscription_num_writers()`).
- Split "replicating" subscription status into two statuses One is still "replicating" and is reported only if something was actually replicated by the subscription since last worker start. The other is "started" and just means that the worker for the subscription is running at the time the status query was executed. This should reduce confusion where subscription would report "replicating" status but worker is in restart loop due to an apply error.
- Substantial test and testing framework improvements
- Improve 2ndQPostgres and BDR version dependency handling (RM17024)
- Add PostgreSQL 12 support to `pglogical_create_subscriber`
- Rework table resynchronization to use separate receiver process from the main one This improves performance of both main apply (primarily latency) and the resynchronization itself. It also fixes potential issue where table could be considered synchronized before the catchup finished completely.

Resolved Issues

- Fix crash on resynchronization of large partitioned tables (RM18154, RM15733, RT68455, RT68352) The resync process would keep crashing due to cache invalidation race condition if the `COPY` took very long or if there was DDL activity on the copied table during the `COPY`.
- Prohibit MERGE and UPSERT on a table without replica identity (RM17323, RT68146) These commands can end up doing `UPDATE` which will break replication if the table has no replica identity as the downstream has no way to find the matching row for updating.
- Resolve relcache reference leak reports (RM16956) Close the relation correctly in `pglogical.show_repset_tables_info()`
- Resolve rare crash in HeapWriter row cleanup code (RM16956)
- Resolve rare crash on worker exit (RM11686) If a pglogical worker exited before it finished initialization it could crash instead of exiting cleanly.
- Fix apply errors parallel index rebuild after `TRUNCATE` (RM17602)

pglogical 3.7.2

This is a beta release of the pglogical 3.7.

pglogical 3.7 is a major release of pglogical. This release includes major new features as well as smaller enhancements.

pglogical 3.7 introduces several major new features as well as architectural changes some of which affect backward compatibility with existing applications.

Important Notes

- Beta software is not supported in production - for application test only
- Upgrade from 3.6 is not supported in this release, yet.

Improvements

- Add support for Postgres 12, deprecate support for older versions pglogical 3.7 now requires at least Postgres 10 and supports up to Postgres 12.

Resolved Issues

- Keep open the connection until `pglogical_create_subscriber` finishes (RM13649) Set `idle_in_transaction_session_timeout` to 0 so we avoid any user setting that could close the connection and invalidate the snapshot.

4.2 Nodes

Each database that participates in pglogical replication must be represented by its own node. Each node must have a name that is unique amongst all nodes that will interact with and communicate with each other.

A pglogical node associates an operator-supplied node name with a connection string that pglogical subscriber nodes may use to communicate with the node. It also serves to mark a database as pglogical-enabled and as something with which to associate other pglogical state like replication sets or subscriptions.

Creating a pglogical node in an existing PostgreSQL database using the `pglogical.create_node` function causes pglogical to launch a manager process for the node but doesn't cause any sort of replication to begin.

A node can be set up as a pglogical provider by having replication sets created and/or tables and DDL filters added to the default replication set. A node can be set up as a pglogical subscriber by having a subscription created on it after the node is created. Any given node can be a provider, a subscriber, or both; nothing in node creation marks the node as one or the other.

Setting up a node as a provider

Newly created pglogical nodes have empty replication sets and do not have any DDL replication filters enabled.

[Replication sets](#) specify what data from which tables and sequences should be sent to any subscriptions that use this node as a provider.

[DDL replication filters](#) capture and replicate schema changes to subscribers so that subscribers' definitions of tables, types, etc remain in sync with the provider.

The simplest way to set up a new node as a provider is to re-create the 'default' replication set with all existing and future tables and sequences auto-added to it. Then add a DDL replication filter to it to enable automatic schema replication. For example:

```
SELECT pglogical.create_node('mynode');
SELECT
pglogical.drop_replication_set('default');
SELECT pglogical.create_replication_set('default', autoadd_tables := true, autoadd_sequences := true,
autoadd_existing := true);
SELECT pglogical.replication_set_add_ddl('default', 'all_ddl', '.*',
NULL);
```

A replication set drop and create is used here because `pglogical.alter_replication_set` cannot autoadd existing tables.

Setting up a node as a subscriber

Newly created nodes do not have any subscriptions so they won't receive changes from anywhere. Use `pglogical.create_subscription(...)` to populate a node with initial data and start receiving streams of changes. Or use the separate `pglogical_create_subscriber` to create a node and subscription from a basebackup or physical streaming replica.

Creating subscriber nodes is discussed in detail in "[Subscriptions](#)".

Node information

`pglogical.local_node`

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

`pglogical.node`

This table lists all PGL nodes.

pglogical.node Columns

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

pglogical.node_interface

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

pglogical.node_interface Columns

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

Node management

Nodes can be added and removed dynamically using SQL calls to functions provided by pglogical.

pglogical.create_node

Creates a pglogical node. Only one pglogical node may exist on a given PostgreSQL database. There's nothing special to mark a node as a provider or subscriber - one node may be either, or both.

Synopsis

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

Parameters

- `node_name` - name of the new node; only one node is allowed per database
- `dsn` - connection string to the node. For nodes that are supposed to be providers; this should be reachable from outside

pglogical.drop_node

Removes the node.

Synopsis

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

Parameters

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

`pglogical.alter_node_add_interface`

Adds an interface to a node.

When a node is created, the interface for it is also created with the `dsn` specified in the `create_node` and with the same name as the node.

If the address or connection options required for a subscriber to connect to a provider change, create a new interface *for* the provider *on* the subscriber, then associate it with the subscription so that the subscriber switches to the new connection string. For example:

```
SELECT pglogical.alter_node_add_interface('providername', 'providername_newconnname', 'connstr');
SELECT pglogical.alter_subscription_interface('subscription_name',
'providername_newconnname');
```

At this time creating a new interface on the provider side has no effect on any subscribers. Subscribers use the connection string specified at subscription creation time, not the connection string declared for a node.

Synopsis

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

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

Parameters

- `node_name` - name of an existing node
- `interface_name` - name of a new interface to be added
- `dsn` - connection string to the node used for the new interface

`pglogical.alter_node_drop_interface`

Remove an existing named interface from a node.

Synopsis

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

Parameters

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

4.3 Replication sets

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

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

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

*Note: Table are **not** added automatically to the "default" replication set, the name "default" just means it exists by default. This behavior can be changed using `pglogical.alter_replication_set`.*

Behavior of partitioned tables

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

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

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

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

Older versions of PostgreSQL

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

Replication set manipulation interfaces

The following functions are provided for managing the replication sets:

`pglogical.create_replication_set`

This function creates a new replication set.

Synopsis

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

Parameters

- `set_name` - name of the set, must be unique
- `replicate_insert` - specifies if `INSERT` is replicated; default true
- `replicate_update` - specifies if `UPDATE` is replicated; default true
- `replicate_delete` - specifies if `DELETE` is replicated; default true
- `replicate_truncate` - specifies if `TRUNCATE` is replicated; default true
- `autoadd_tables` - specifies if newly created tables should be automatically added to the new replication set; default false
- `autoadd_sequences` - specifies if newly created sequences should be automatically added to the new replication set; default false
- `autoadd_existing` - this in combination with `autoadd_tables` or `autoadd_sequences` specifies if any existing tables and sequences should be added as well

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

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

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

`pglogical.alter_replication_set`

This function changes the parameters of the existing replication set.

Synopsis

```

pglogical.alter_replication_set
(
    set_name
name,
    replicate_insert
boolean,
    replicate_update
boolean,
    replicate_delete
boolean,
    replicate_truncate boolean,
    autoadd_tables
boolean,
    autoadd_sequences boolean
)

```

Parameters

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

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

pglogical.drop_replication_set

Removes the replication set.

Synopsis

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

Parameters

- `set_name` - name of the existing replication set

pglogical.replication_set_add_table

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

Synopsis

```

pglogical.replication_set_add_table
(

```

```

    set_name
name,
    relation
regclass,
    synchronize_data
boolean,
    columns text[],
    row_filter text
)

```

Parameters

- `set_name` - name of the existing replication set
- `relation` - name or OID of the table to be added to the set
- `synchronize_data` - if true, the table data is synchronized on all subscribers which are subscribed to given replication set; default false
- `columns` - list of columns to replicate. Normally when all columns should be replicated, this will be set to NULL which is the default.
- `row_filter` - row filtering expression; default NULL (no filtering). See [Row Filtering On Provider](#) for more info.

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

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

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

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

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

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

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

pglogical.replication_set_add_all_tables

Adds all tables in given schemas.

Synopsis

```

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

```

Only existing tables are added; any tables created later will not be added automatically. To see how to automatically add tables to the correct replication set at creation time, see [Automatic assignment of replication sets for new tables](#).

Parameters

- `set_name` - name of the existing replication set
- `schema_names` - array of names name of existing schemas from which tables should be added
- `synchronize_data` - if true, the table data is synchronized on all subscribers which are subscribed to the given replication set; default false

`pglogical.replication_set_remove_table`

Removes a table from a specified existing replication set.

Synopsis

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

Parameters

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

`pglogical.replication_set_add_sequence`

Adds a sequence to a replication set.

Synopsis

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

Parameters

- `set_name` - name of the existing replication set
- `relation` - name or OID of the sequence to be added to the set
- `synchronize_data` - if true, the sequence value will be synchronized immediately; default false

`pglogical.replication_set_add_all_sequences`

Adds all sequences from the given schemas.

Synopsis

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

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

Parameters

- `set_name` - name of the existing replication set
- `schema_names` - array of names of existing schemas from which tables should be added
- `synchronize_data` - if true, the sequence value will be synchronized immediately; default false

`pglogical.replication_set_remove_sequence`

Remove a sequence from a replication set.

Synopsis

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

Parameters

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

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

Automatic assignment of replication sets for new tables

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

Example:

```
CREATE OR REPLACE FUNCTION pglogical_assign_repset()
RETURNS event_trigger AS $$
DECLARE obj
record;
BEGIN
    FOR obj IN SELECT * FROM
pg_event_trigger_ddl_commands()
    LOOP
```

```

        IF obj.object_type = 'table' THEN
            IF obj.schema_name = 'config' THEN
                PERFORM pglogical.replication_set_add_table('configuration',
obj.objid);
            ELSIF NOT obj.in_extension
THEN
                PERFORM pglogical.replication_set_add_table('default',
obj.objid);
            END IF;
        END IF;
    END LOOP;
END;
$$ LANGUAGE
plpgsql;

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

```

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

Additional functions

`pglogical.synchronize_sequence`

Push sequence state to all subscribers.

Synopsis

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

Unlike the subscription and table synchronization function, this function should be run on the provider. It forces an update of the tracked sequence state which will be consumed by all subscribers (replication set filtering still applies) once they replicate the transaction in which this function has been executed.

Parameters

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

Row Filtering on Provider

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

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

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

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

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

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

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

Writing safer row filters

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

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

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

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

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

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

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

Changing row filters

To change a row-filter expression on a table, use a single transaction to remove the table from the replication set, then add it again with the new row filter

expression. Do not specify data sync and make sure to explicitly repeat the set of replicated columns. You can check the `pglogical.tables` view for the old column set and row filter.

See `pglogical.replication_set_add_table`.

4.4 DDL Replication

pglogical3 supports *transparent DDL replication*, where schema change commands run on provider node(s) are automatically captured and replicated to subscriber node(s) without the need for wrapper functions like `pglogical.replicate_ddl_command` or external schema management tools.

The main difference from normal replication of table rows ("table replication") is that DDL replication replicates statements themselves rather than the effects of those statements. Normal data replication replicates the changes made by a statement, e.g. it sends the rows that got `UPDATE` d by an `UPDATE` command rather than replicating and executing the `UPDATE` statement itself. pglogical's DDL replication captures, replicates and executes *the text of the DDL statement itself*.

Minimal example

Enabling DDL replication on a pglogical provider can be as simple as:

```
SELECT * FROM pglogical.replication_set_add_ddl('ddl_sql', 'all', NULL,
NULL);
```

to replicate any captureable DDL statements executed by any user on the provider database to any subscriber(s) that subscribe to the enabled-by-default `ddl_sql` replication set.

However it's generally *recommended* to enable replication of a targeted subset of DDL instead.

There are also caveats relating to replication of changes to "global objects" like roles, the handling of some `ALTER TABLE` modes, etc that are important to understand. See [Restrictions](#) below.

How to use DDL replication

Transparent DDL replication in pglogical builds on the same [Replication Sets](#) model that's used by replication of table contents. The same replication set(s) may be used for both replicating table contents and for DDL.

To replicate future DDL commands on a provider, a DDL replication filter must be added to the replication set(s) used by subscribers that should receive and apply the DDL.

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

Functions for managing DDL replication filters

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

pglogical.replication_set_add_ddl

Adds a DDL replication filter to a replication set.

Synopsis

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

Parameters

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

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

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

pglogical.replication_set_remove_ddl

Remove a DDL replication filter from replication set.

Synopsis

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

Parameters

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

Additional functions and views

```
pglogical.ddl_replication
```

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

`pglogical.ddl_replication` Columns

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

`pglogical.ddl_replication`

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

`pglogical.ddl_replication` Columns

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

`pglogical.replicate_ddl_command`

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

In most cases `pglogical.replicate_ddl_command` is rendered obsolete by `pglogical`'s support for DDL replication filters.

Synopsis

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

Parameters

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

Restrictions

When the DDL replication filter matches a DDL command it will modify the `search_path` configuration parameter used to execute the DDL on both provider and subscriber(s) to include only `pg_catalog` i.e. the system catalogs. This means that all the user objects referenced in the query must be fully schema qualified. For example `CREATE TABLE foo...` will raise an error when executed and has to be rewritten as `CREATE TABLE public.foo...`.

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

```
ERROR: no schema has been selected to create
in
STATEMENT: CREATE TABLE foo (id
integer);
```

or will raise an `ERROR` message complaining that data types, tables, etc referenced by the DDL statement do not exist even when they can be queried normally, are shown by `psql`, etc.

For example, attempting to drop some table `public.a` will fail:

```
ERROR: table "a" does not exist
STATEMENT: DROP TABLE a;
```

and must be reframed as:

```
DROP TABLE public.a;
```

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

For testing purposes it can be useful to simulate the behaviour of DDL replication capture manually in `psql`. To do so, set the `search_path` to the empty string, e.g.

```
BEGIN;
SET LOCAL search_path =
'';
CREATE TABLE mytable(id
integer);
COMMIT;
```

will fail with `ERROR: no schema selected to create in`.

Considerations with global objects

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

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

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

pglogical.tables

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

pglogical.tables Columns

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

pglogical.queue

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

pglogical.queue_truncate

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

4.5 Subscription Overview

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

Subscription information

pglogical.stat_subscription

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

pglogical.stat_subscription Columns

Column	Type	Description
sub_name	name	Name of the subscription
subid	oid	Oid of the subscription
nconnect	bigint	Number of times this subscription has connected upstream
ncommit	bigint	Number of commits this subscription did
nabort	bigint	Number of aborts writer did for this subscription
nerror	bigint	Number of errors writer has hit for this subscription
nskippedtx	bigint	Number of transactions skipped by writer for this subscription (currently normally 0 for pgl subscription)
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
ndeadlocks	bigint	Number of errors that were caused by deadlocks
nretries	bigint	Number of retries the writer did (without going for full restart/reconnect)
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 <code>track_io_timing</code> is enabled, otherwise zero)
blk_write_time	double precision	Total time the subscription spent writing blocks, in milliseconds (if <code>track_io_timing</code> is enabled, otherwise zero)
connect_time	timestamp with time zone	Time when the current upstream connection was established, NULL if not connected
last_disconnect_time	timestamp with time zone	Time when the last upstream connection was dropped
start_lsn	pg_lsn	LSN from which this subscription requested to start replication from the upstream
retries_at_same_lsn	bigint	Number of attempts the subscription was restarted from the same LSN value
curr_ncommit	bigint	Number of commits this subscription did after the current connection was established

pglogical.stat_relation

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

pglogical.stat_relation Columns

Column	Type	Description
nspname	name	Name of the relation's schema
relname	name	Name of the relation
relid	oid	OID of the relation

Column	Type	Description
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 <code>track_io_timing</code> is enabled, otherwise zero)
blk_write_time	double precision	Total time spent writing blocks for the relation, in milliseconds (if <code>track_io_timing</code> is enabled, otherwise zero)
lock_acquire_time	double precision	Total time spent acquiring locks on the relation (if <code>pglogical.track_apply_lock_timing</code> is enabled, otherwise zero)

`pglogical.replication_status`

Replication status view for each subscription. We consider replication to be blocked when the subscription has restarted from the same LSN at least twice and not a single transaction is yet applied after the current upstream connection was established. If the very first transaction after restart is very big and still being applied, the `replication_blocked` result maybe wrong.

`pglogical.replication_status` Columns

Column	Type	Description
sub_name	name	Name of the subscription
connected	boolean	Is the subscription connected to the upstream?
replication_blocked	boolean	Is the replication currently blocked?
connect_time	timestamp with time zone	Time when the current connection was established
disconnect_time	timestamp with time zone	Time when the last connection was dropped
uptime	interval	Duration since the current connection is active

`pglogical.local_sync_status`

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

`pglogical.show_workers`

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

SQL interfaces

`pglogical.create_subscription`

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

Synopsis

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

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

Subscription setup is asynchronous. `pglogical.create_subscription` returns after creating the replication slot (unless `create_slot` is false) but before the subscription is synchronized and streaming. Use `pglogical.wait_for_subscription_sync_complete` to wait until the subscription is up and has completed any requested schema and/or data sync.

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

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

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

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

Note: `pglogical.create_subscription` will appear to hang (it will wait indefinitely without returning) if the database pointed to by `provider_dsn` is on the same PostgreSQL instance as the subscriber and `create_slot` is true. This happens because the replication slot creation command on the provider waits for all transactions that were in-progress at the time it started to commit, but the transaction running `pglogical.create_subscription` cannot commit until after the create replication slot command returns, so the two deadlock. The PostgreSQL deadlock detector does not identify this condition as it is not a deadlock on heavyweight locks, and is not visible in the `pg_locks` view. To make this

fairly uncommon use case work, manually create a logical replication slot for the pglogical subscription on the provider using the 'pglogical_output' output plugin, e.g.

```
SELECT pg_catalog.create_logical_replication_slot(
pglogical.gen_slot_name(
'SUBSCRIBER_DBNAME',
'PROVIDER_NODE_NAME',
'SUBSCRIPTION_NAME'
), 'pglogical_output');
```

then specify `create_slot := false` to `pglogical.create_subscription()`. You may alternately choose your own replication slot name instead of using `gen_slot_name` then pass it as the `slot_name` parameter to `create_subscription`.

Parameters

- `subscription_name` - name of the subscription; must be unique
- `provider_dsn` - connection string to a provider
- `replication_sets` - array of replication sets to subscribe to, these must already exist, default is "{default,ddl_sql}"
- `synchronize_structure` - specifies if to synchronize structure from provider to the subscriber; default false
- `synchronize_data` - specifies if to synchronize data from provider to the subscriber; default true
- `create_slot` - set to false to suppress automatic creation of a logical replication slot on the provider in order to use a pre-created one; default true
- `slot_name` - override the autogenerated replication slot name pglogical generates in order to supply your own; default is same as that generated by `pglogical.pglogical_gen_slot_name()`
- `forward_origins` - array of replication origin names to forward. Currently the only supported values are: an empty array meaning don't forward any changes that didn't originate on provider node (this is useful for two-way replication between the nodes); or "{all}" which means replicate all changes no matter what is their origin. The default is "{all}"
- `apply_delay` - how much to delay replication; default is 0 seconds. Mainly used for application testing, but also useful for delayed standbys.
- `forward_origins` - array of origin names to forward; currently only supported values are empty array meaning don't forward any changes that didn't originate on provider node (this is useful for two-way replication between the nodes), or "{all}" which means replicate all changes no matter what is their origin; default is "{all}"
- `strip_origins` - determines whether to remove origin names from forwarded data, making it look like the data originate from local node, and allowing to forward the data to a subscription in the same instance (default is "false" which keeps origin info). The negative effect is it makes it impossible to redirect the subscription to the first node.
- `num_writers` - number of parallel writers for this subscription, -1 means the subscription will use the default as specified by the GUC `pglogical.writers_per_subscription`. Valid values are either -1 or a positive integer.
- `writer` - which writer to use for writing the data from the replication stream. Only `HeapWriter` is supported in this version of PGLogical. `local` is an alias for backwards compatibility.
- `writer_options` - writer-specific options as an array of keys and values

pglogical_create_subscriber

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

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

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

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

Synopsis

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

Options

General Options

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

Configuration Files Override

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

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

`pglogical.drop_subscription`

Disconnects the subscription and removes it from the catalog.

Synopsis

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

Parameters

- `subscription_name` - name of the existing subscription
- `ifexists` - if true, error is not thrown when subscription does not exist; default is false

`pglogical.alter_subscription_disable`

Disables a subscription and disconnects it from the provider.

Synopsis

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

Parameters

- `subscription_name` - name of the existing subscription
- `immediate` - if true, the subscription is stopped immediately, otherwise it will be only stopped at the end of the current transaction; default is false

pglogical.alter_subscription_enable

Enables disabled subscription.

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

Parameters

- `subscription_name` - name of the existing subscription
- `immediate` - if true, the subscription is started immediately, otherwise it will be only started at the end of current transaction; default is false

pglogical.alter_subscription_num_writers

Changes the number of writers for a subscription.

```
pglogical.alter_subscription_num_writers(subscription_name name, num_writers int, immediate bool)
```

Parameters

- `subscription_name` - name of the existing subscription
- `num_writers` - number of writers for this subscription, -1 means the subscription will use value set by `pglogical.writers_per_subscription` GUC
- `immediate` - if true, the subscription is started immediately, otherwise it will be only started at the end of current transaction, default is false

pglogical.alter_subscription_interface

Switch the subscription to use a different interface to connect to the provider node. This is how you change the address, port etc that a subscription uses when connecting to a provider.

See `pglogical.alter_node_create_interface()` for usage.

Synopsis

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

Parameters

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

`pglogical.alter_subscription_synchronize`

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

Synopsis

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

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

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

Parameters

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

`pglogical.alter_subscription_resynchronize_table`

Asynchronously resynchronize one existing table.

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

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

Synopsis

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

Parameters

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

pglogical.show_subscription_status

Shows status and basic information about a subscription.

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

Parameters

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

pglogical.show_subscription_table

Shows the synchronization status of a table.

Synopsis

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

Parameters

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

pglogical.show_subscription_clock_drift

Shows clock drift between provider and subscriber.

On the subscriber at apply time, we track the commit timestamp received from the provider and the current local timestamp. When the above function is

invoked, we generate a diff (interval) of these values. A negative value will indicate clock drift.

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

Parameters

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

pglogical.alter_subscription_add_replication_set

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

Synopsis

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

Parameters

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

pglogical.alter_subscription_remove_replication_set

Removes one replication set from a subscriber.

Synopsis

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

Parameters

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

pglogical.wait_for_subscription_sync_complete

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

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

Synopsis

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

Parameters

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

`pglogical.wait_for_table_sync_complete`

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

Synopsis

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

Parameters

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

```
pglogical.wait_slot_confirm_lsn(name, pg_lsn)
```

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

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

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

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

No timeout is offered, use a `statement_timeout`.

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

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

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

Synopsis

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

Typically it's sufficient to use:

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

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

Parameters

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

```
pglogical.standby_wait_replay_upstream_lsn(pg_lsn)
```

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

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

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

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

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

pglogical.alter_subscription_skip_changes_upto

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

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

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

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

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

This function only works on disabled subscriptions.

The usual sequence of steps is:

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

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

Synopsis

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

Example

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

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

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

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

```
SELECT subscription_name, slot_name
FROM pglogical.subscriptions
s
```

```
WHERE replication_origin_id =
1
```

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

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

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

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

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

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

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

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

Finally, re-enable the subscription:

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

`pglogical.alter_subscription_writer_options`

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

Synopsis

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

);
```

Example

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

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

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

```
SELECT pglogical.alter_subscription_writer_options(sub_name, 'super', '{UPDATE,INSERT,DELETE,}') FROM pglogical.subscription;
```

pglogical.alter_subscription_set_conflict_resolver

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

Synopsis

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

Parameters

- `sub_name` - name of subscription to change
- `conflict_type` - type of conflict to configure (see below)
- `conflict_resolver` - which resolver to use for the given conflict type (see below)

Conflict type can be one of:

- `insert_exists` - the row being inserted exists locally
- `update_missing` - the row being updated does not exist locally
- `delete_missing` - the row being deleted does not exist locally
- `update_origin_change` - the row being updated was updated on a different origin
- `target_table_missing` - the table corresponding to the change does not exist locally
- `target_column_missing` - the column being updated or inserted to does not exist locally
- `source_column_missing` - a column that exists locally is not available in the updated or inserted row replicated
- `delete_recently_updated` - the row being deleted was updated locally recently
- `update_pkey_exists` - the updated primary key exists locally
- `apply_error` - an error occurred while applying the change locally
- `apply_error_trigger` - an error occurred while firing a trigger locally after applying the change

- `apply_error_ddl` - an error occurred during applying a DDL that was replicated
- `apply_error_dml` - an error occurred while applying a DML that was

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

Conflict resolver can be one of:

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

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

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

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

Example

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

```
SELECT pglogical.alter_subscription_set_conflict_resolver(`sub_name`, 'insert_exists',
'update_if_newer')
```

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

Listing Conflict Resolution Configurations

The catalog `pglogical.sub_cfl_res` show non-default conflict resolution configuration settings for each subscription.

`pglogical.alter_subscription_add_log`

Add conflict logging configuration for a subscription.

This can primarily be used to log conflicts into the `pglogical.apply_log` table.

Synopsis

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

Listing Conflict Logging Configurations

The catalog `pglogical.sub_log` shows all the logging configurations. It lists the name of the logging configuration, where it logs and which conflicts type and resolution it logs.

Parameters

- `sub_name` - name of the subscription that is being changed
- `log_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

`pglogical.alter_subscription_remove_log`

Remove existing conflict logging configuration for a subscription.

Synopsis

```
pglogical.alter_subscription_remove_log(
    sub_name text,
    log_name text
)
```

Parameters

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

4.5.1 pglogical writer

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

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

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

Conflict handling

In case the node is subscribed to multiple providers, or when local writes happen on a subscriber, conflicts can arise for the incoming changes. These are automatically detected and can be acted on depending on the configuration.

The configuration of the conflicts resolver is done using `pglogical.alter_subscription_set_conflict_resolver()`.

Row versioning

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

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

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

Configuration options

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

`pglogical.conflict_log_level`

Sets the log level for reporting detected conflicts.

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

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

The default is `LOG`.

`pglogical.conflict_ignore_redundant_updates`

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

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

The default is `false`.

`pglogical.conflict_check_full_tuple`

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

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

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

The default is `false`.

`pglogical.batch_inserts`

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

The batch inserts will improve replication performance of transactions that perform many inserts into one table. pglogical will switch to batch mode when the transaction performed more than 5 INSERTs, or 5 rows within a COPY.

It's only possible to switch to batch mode when there are no `INSTEAD OF INSERT` and `BEFORE INSERT` triggers on the table and when there are no defaults with volatile expressions for columns of the table.

The default is `true`.

`config.session_replication_role`

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

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

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

Restrictions

There are some additional restrictions imposed by pglogical writer over the standard set of [Restrictions](#).

Only one unique index/constraint/PK

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

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

Deferrable unique indexes

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

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

Foreign Keys

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

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

TRUNCATE

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

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

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

Triggers

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

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

4.6 PostgreSQL settings which affect pglogical

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

PostgreSQL must be configured for logical replication:

```
wal_level = 'logical'
```

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

```
shared_preload_libraries = 'pglogical'
```

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

```
max_worker_processes = 10
```

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

- one per database + two per subscription on the subscriber (downstream).

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

```
max_replication_slots = 10
```

One per subscription on both provider and subscriber is needed.

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

```
max_wal_senders = 10
```

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

If you are using PostgreSQL 9.5+ (this won't work on 9.4) and want to handle conflict resolution with last/first update wins (see [pglogical writer](#)), you can add this additional option to postgresql.conf:

```
track_commit_timestamp = on
```

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

pglogical specific settings

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

`pglogical.synchronous_commit`

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

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

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

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

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

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

The default is off.

`pglogical.track_subscription_apply`

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

Collecting statistics requires additional CPU resources on the subscriber.

The default is on.

`pglogical.track_relation_apply`

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

Collecting statistics requires additional CPU resources on the subscriber.

The default is off.

`pglogical.track_apply_lock_timing`

This tracks lock timing when tracking statistics for relations.

The default is off.

`pglogical.temp_directory`

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

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

`pglogical.extra_connection_options`

This option may be set to assign connection options that apply to all connections made by `pglogical`. This can be a useful place to set up custom keepalive options, etc.

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

`pglogical.synchronize_failover_slot_names`

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

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

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

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

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

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

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

`pglogical.synchronize_failover_slots_drop`

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

The default value is `true`.

`pglogical.synchronize_failover_slots_dsn`

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

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

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

`pglogical.standby_slot_names`

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

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

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

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

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

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

and/or

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

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

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

`pglogical.standby_slots_min_confirmed`

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

`pglogical.writer_input_queue_size`

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

`pglogical.writer_output_queue_size`

This option is used to specify the size of the shared memory queue used by the receiver to receive data from the writer process. Since the writer is not expected to send a large amount of data, a relatively smaller sized queue should be enough. The default is 32kB and the maximum allowed size is 1MB. While any storage size specifier can be used to set the GUC, the default is kB.

`pglogical.min_worker_backoff_delay` and `pglogical.max_worker_backoff_delay`

Rate limit pglogical background worker launches by preventing a given worker from being relaunched more often than every `pglogical.min_worker_backoff_delay` milliseconds. On repeated errors, the back-off increases exponentially with added jitter up to maximum of `pglogical.max_worker_backoff_delay`.

Time-unit suffixes are supported.

Note

This setting currently only affects receiver worker, which means it primarily affects how fast a subscription will try to reconnect on error or connection failure.

The default for `pglogical.min_worker_backoff_delay` is 1 second, for `pglogical.max_worker_backoff_delay` it is 1 minute.

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

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

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

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

`pglogical.max_writers_per_subscription`

Specifies the maximum number of parallel writers that a subscription may use. Values between 1 and 64 are allowed, with the default being 8. When set to 1, parallel apply is effectively disabled.

`pglogical.writers_per_subscription`

Sets the default number of parallel writers for subscriptions without an explicitly set value. Values between 1 and 64 are allowed, with the default being 4.

4.7 Failover with pglogical3

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

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

Provider failover setup

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

Given a topology like this:

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

```
|
[Subscriber1]
```

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

- Provider1 and Provider2 run PostgreSQL 10 or newer
- The connection between Provider1 and Provider2 uses streaming replication with hot standby feedback and a physical replication slot. It's OK if WAL archiving and a `restore_command` is configured as a fallback.
- Provider2 has:
 - `recovery.conf`:
 - `primary_conninfo` pointing to Provider1
 - `primary_slot_name` naming a physical replication slot on Provider1 to be used only by Provider2
 - `postgresql.conf`:
 - `pglogical` in its `shared_preload_libraries`
 - `hot_standby = on`
 - `hot_standby_feedback = on`
 - `pglogical.synchronize_failover_slot_names` can be modified to specify which slots should be synchronized (default is all pglogical/bdr slots)
- Provider1 has:
 - `postgresql.conf`:
 - `pglogical.standby_slot_names` lists the physical replication slot used for Provider2's `primary_slot_name`. Promotion will still work if this is not set, but subscribers may be inconsistent per the linked documentation on the setting.
- Provider2 has had time to sync and has created a copy of Subscriber1's logical replication slot. pglogical3 creates master slots on replicas automatically once the replica's resource reservations can satisfy the master slot's requirements, so just check that all pglogical slots on the master exist on the standby, and have `confirmed_flush_lsn` set.
- Provider2 takes over Provider1's IP address or hostname *or* Subscriber1's existing subscription is reconfigured to connect to Provider2 using `pglogical.alter_node_add_interface` and `pglogical.alter_subscription_interface`.

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

The post-failover topology is:

```
XXXXXXXXXXXX
xProvider1x          [Provider2]
XXXXXXXXXXXX          ^
                    |
                    |
                    |
                    |
|-----|
|
| <- logical
|   replication
|
|
[Subscriber1]
```

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

Normally when a PostgreSQL instance is replaced by a promoted physical replica of the same instance, any replication slots on that node are lost. Replication slot status is not itself replicated along physical replication connections and does not appear in WAL. So if the failed-and-replaced node was the upstream provider of any logical subscribers, those subscribers stop being able to receive data and cannot recover. Physical failover breaks logical replication connections.

To work around this, pglogical3 running on the failover-candidate replica syncs the state of the master provider's logical replication slot(s) to the replica. It

also sends information back to the master to ensure that those slots guarantees' (like `catalog_xmin`) are respected by the master. That synchronization requires a physical replication slot to avoid creating excessive master bloat and to ensure the reservation is respected by the master even if the replication connection is broken.

Subscriber failover setup

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

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

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

Given the server topology:



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

For replication to resume promptly it may be necessary to explicitly terminate the walsender for Subscriber1 on Provider1 if the connection failure is not detected promptly by Provider1. pglogical enables TCP keepalives by default so in the absence of manual action it should exit and release the slot automatically in a few minutes.

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

After failover the topology is:



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

Additional functions

pglogical.sync_failover_slots()

Signal the supervisor to restart the mechanism to synchronize the failover slots specified in the `pglogical.synchronize_failover_slot_names`

Synopsis

```
pglogical.syncfailover_slots();
```

This function should be run on the subscriber.

Legacy: Provider failover with pglogical2 using failover slots

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

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

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

4.8 Restrictions

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

Superuser is required

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

UNLOGGED and TEMPORARY not replicated

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

One database at a time

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

PRIMARY KEY or REPLICA IDENTITY required

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

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

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

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

DDL

There are several limitations of DDL replication in pglogical, for details check the [DDL Replication](#) chapter.

Sequences

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

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

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

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

PostgreSQL Version differences

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

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

Replicating between different minor versions makes no difference at all.

pglogical.pglogical_version

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

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

pglogical.pglogical_version_num

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

Database encoding differences

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

Large objects

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

Additional restrictions

Please note that additional restrictions may apply depending on which writer is being used and which version of PostgreSQL is being used. These additional restrictions are documented in their respective sections (i.e., every writer documents its own additional restrictions).

4.9 Error handling in pglogical

The main tool for troubleshooting is the PostgreSQL log file.

On the upstream side, monitoring uses the views:

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

On the subscriber side there are numerous helper functions and views that may be consulted to gain insight into pglogical's configuration and behaviour.

Start with the configuration and status summary views:

Configuration can be fetched from:

```
SELECT * FROM
pglogical.subscriptions;
SELECT * FROM pglogical.TABLES;
SELECT * FROM pglogical.SEQUENCES;
```

```
SELECT * FROM
pglogical.DDL_REPLICATION;
```

Observe worker activity with:

```
SELECT * FROM pglogical.workers;
SELECT * FROM
pglogical.worker_error_summary;
SELECT * FROM pglogical.apply_log;
SELECT * FROM pglogical.apply_log_summary;
SELECT * FROM pglogical.worker_locks;
```

Statistics are reported by:

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

Other views provide logs and details:

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

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

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

`pglogical.worker_error` and `pglogical.worker_error_summary`

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

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

`pglogical.worker_error_summary` is a convenience view over `pglogical.worker_error` available in 3.7 and above.

`pglogical.worker_tasks`

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

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

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

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

`pglogical.apply_log` and `pglogical.apply_log_summary`

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

`pglogical.sub_log`

The `pglogical.sub_log` table contains *conflict log filter definitions* that are applied when recording entries in `pglogical.apply_log`, controlling whether conflicts are recorded to a log table and/or postgres log, or silently dropped. It's managed by `pglogical.alter_subscription_add_log(...)` and `pglogical.alter_subscription_remove_log()`.

If you aren't seeing expected conflict information when debugging an issue, check to make sure you have not filtered it out.

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

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

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

For example, if an apply worker tries to apply an `UPDATE` and the new row violates a secondary unique constraint on the target table, the apply worker will report the unique violation error and exit. The error information will be visible in `pglogical.worker_error_summary` (3.7+, `pglogical.worker_error` on 3.6). The walsender worker on the peer end will exit automatically as well. The apply worker will be relaunched by the manager worker for the database in a few seconds and will retry the failed transaction from the beginning. If the conflicting row has since been removed the transaction will apply normally and replication will resume. If not, the worker will error again and the cycle will repeat until the cause of the error is fixed. In this case the fix would typically be for another subscription or a local application write to replicate a change that clears the unhandled conflict condition or for the administrator to intervene to change the conflicting row.

Diagnosing and fixing errors

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

Common problems

Some issues that arise when operating pglogical include:

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

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

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

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

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

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

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

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

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

Multiple data source issues

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

These include:

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