



EDB

BDR Bi-Directional Replication

Release 4.1.0

EDB

Jun 05, 2022

CONTENTS

1	Overview	1
2	Supported PostgreSQL database servers	3
2.1	Release Notes	3
2.1.1	BDR 4.1.0	4
2.1.2	BDR 4.0.2	8
2.1.3	BDR 4.0.1	10
2.1.4	BDR 4.0.0	12
2.2	Architectural Overview	15
2.2.1	Basic Architecture	16
2.2.2	Architectural Options & Performance	17
2.2.3	Deployment	18
2.2.4	Clocks and Timezones	19
2.3	Application use	19
2.3.1	Application behavior	19
2.3.2	Nonreplicated statements	20
2.3.3	DML and DDL replication	21
2.3.4	Replicating between different release levels	21
2.3.5	Replicating between nodes with differences	21
2.3.6	Comparison between nodes with differences	22
2.3.7	General rules for applications	23
2.3.8	Timing considerations and synchronous replication	23
2.3.9	Application testing	23
2.3.10	Performance testing and tuning	27
2.3.11	Assessing suitability	27
2.4	PostgreSQL configuration for BDR	29
2.4.1	PostgreSQL settings for BDR	29
2.4.2	BDR-specific settings	30
2.5	Node Management	37
2.5.1	Creating and Joining a BDR Group	38
2.5.2	Witness Nodes	40
2.5.3	Logical Standby Nodes	40
2.5.4	Physical Standby Nodes	41
2.5.5	Sub-Groups	43
2.5.6	Decoding Worker	44
2.5.7	Node Restart and Down Node Recovery	45
2.5.8	Removing a Node From a BDR Group	47
2.5.9	Listing BDR Topology	48
2.5.10	Node Management Interfaces	49
2.5.11	Node Management Commands	57

2.6	DDL Replication	59
2.6.1	DDL Replication Options	60
2.6.2	Executing DDL on BDR Systems	61
2.6.3	DDL Locking Details	61
2.6.4	Minimizing the Impact of DDL	62
2.6.5	Handling DDL With Down Nodes	63
2.6.6	Statement Specific DDL Replication Concerns	64
2.6.7	BDR DDL Command Handling Matrix	64
2.7	Security and Roles	77
2.7.1	Catalog Tables	78
2.7.2	BDR Functions & Operators	78
2.7.3	Granting privileges on catalog objects	78
2.7.4	Role Management	78
2.7.5	Roles and Replication	79
2.7.6	Connection Role	79
2.7.7	Privilege Restrictions	79
2.7.8	BDR Default/Predefined Roles	81
2.7.9	Verification	83
2.8	Conflicts	85
2.8.1	How conflicts happen	85
2.8.2	Types of conflict	86
2.8.3	Avoiding or tolerating conflicts	95
2.8.4	Conflict detection	96
2.8.5	Conflict resolution	99
2.8.6	Conflict logging	102
2.8.7	Data verification with LiveCompare	103
2.9	Sequences	104
2.9.1	BDR Global Sequences	104
2.9.2	UUIDs, KSUUIDs and Other Approaches	109
2.9.3	Global Sequence Management Interfaces	111
2.9.4	KSUUID v2 Functions	115
2.9.5	KSUUID v1 Functions	116
2.10	Column-level conflict detection	117
2.10.1	Enabling and disabling column-level conflict resolution	118
2.10.2	DDL locking	119
2.10.3	Current versus commit timestamp	119
2.10.4	Inspecting column timestamps	120
2.10.5	Handling column conflicts using CRDT data types	121
2.10.6	Notes	121
2.11	Conflict-free Replicated Data Types	122
2.11.1	State-based and operation-based CRDTs	125
2.11.2	Disk-Space Requirements	126
2.11.3	CRDT Types vs Conflicts Handling	126
2.11.4	CRDT Types vs. Conflict Reporting	126
2.11.5	Resetting CRDT Values	127
2.11.6	Implemented CRDT data types	127
2.12	Transaction streaming	133
2.12.1	BDR enhancements	133
2.12.2	Configuration	134
2.12.3	Monitoring	135
2.13	Durability & Performance Options	136
2.13.1	Overview	136
2.13.2	Terms and Definitions	136
2.13.3	Comparison	137

2.13.4	Internal Timing of Operations	138
2.13.5	Configuration	138
2.13.6	Planned Shutdown and Restarts	139
2.13.7	Legacy Synchronous Replication using BDR	139
2.14	Group Commit	140
2.14.1	Requirements	140
2.14.2	Configuration	140
2.14.3	Reference	141
2.15	Eager Replication	143
2.15.1	Requirements	143
2.15.2	Usage	143
2.15.3	Error handling	144
2.15.4	Effects of Eager Replication in General	145
2.16	Commit At Most Once	145
2.16.1	Requirements	146
2.16.2	Configuration	146
2.16.3	Failure scenarios	147
2.16.4	Application use	149
2.16.5	Interaction with DDL and global locks	153
2.16.6	CAMO limitations	153
2.16.7	Performance implications	154
2.16.8	Client application testing	154
2.16.9	CAMO versus group commit	154
2.17	Lag Control	154
2.17.1	Requirements	155
2.17.2	Configuration	155
2.17.3	Overview	155
2.17.4	Transaction Application	156
2.17.5	Limitations	156
2.17.6	Caveats	156
2.17.7	Meeting Organizational Objectives	156
2.18	AutoPartition	157
2.18.1	Auto Creation of Partitions	157
2.19	Timestamp-Based Snapshots	163
2.20	Replication Sets	164
2.20.1	Using Replication Sets	164
2.20.2	Behavior of Partitioned Tables	164
2.20.3	Behavior with Foreign Keys	165
2.20.4	Replication Set Management	166
2.20.5	Replication Set Membership	169
2.20.6	Listing Replication Sets	171
2.20.7	DDL Replication Filtering	172
2.21	Stream Triggers	175
2.21.1	Trigger execution during Apply	175
2.21.2	Terminology of row-types	176
2.21.3	Conflict Triggers	177
2.21.4	Transform Triggers	178
2.21.5	Stream Triggers Variables	178
2.21.6	Information functions	179
2.21.7	Row Contents	182
2.21.8	Triggers Notes	182
2.21.9	Stream Triggers Manipulation Interfaces	182
2.21.10	Stream Triggers Examples	184
2.22	Explicit Two-Phase Commit (2PC)	186

2.22.1	Usage	187
2.23	Catalogs and views	187
2.23.1	User-visible catalogs and views	187
2.23.2	Internal catalogs and views	217
2.24	BDR System Functions	219
2.24.1	Version Information Functions	219
2.24.2	System Information Functions	219
2.24.3	System and Progress Information Parameters	219
2.24.4	Consensus Function	220
2.24.5	Utility Functions	222
2.24.6	Global Advisory Locks	229
2.24.7	Monitoring Functions	231
2.24.8	Internal Functions	233

OVERVIEW

BDR is a PostgreSQL extension providing 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.

Detailed overview about how BDR works is described in the [Architectural Overview](#) chapter.

SUPPORTED POSTGRESQL DATABASE SERVERS

BDR is compatible with PostgreSQL, EDB Postgres Extended and EDB Postgres Advanced flavors of PostgreSQL database servers and can be deployed as a standard PG extension. See [Compatibility Matrix](#) for details of supported version combinations.

It is important to note that some key BDR features depend on certain core capabilities being available within the targeted PostgreSQL database server. Therefore, it is essential for the BDR customers to also adopt the PostgreSQL database server flavor that is best suited to their business needs. For example, if having the BDR feature “Commit At Most Once (CAMO)” is mission critical to a BDR customer’s use case, they should not adopt the community PostgreSQL flavor for it does not have the core capability required to handle CAMO. The full feature matrix compatibility can be found in [Choosing a Postgres distribution](#).

BDR offers close to native PostgreSQL compatibility. However, some access patterns don’t necessarily work as well in multi-node setup as they do on single instance. There are also some limitations in what can be safely replicated in multi-node setting. The chapter [Application Usage](#) goes into detail on how BDR behaves from application development perspective.

2.1 Release Notes

BDR is a PostgreSQL extension providing 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 a very high availability.

The release notes in this section provide information on what was new in each release.

Version	Release Date
<i>BDR 4.1.0</i>	2022 May 17
<i>BDR 4.0.2</i>	2022 Feb 15
<i>BDR 4.0.1</i>	2022 Jan 18
<i>BDR 4.0.0</i>	2021 Dec 01

2.1.1 BDR 4.1.0

This is a minor release of BDR 4 which includes new features as well as fixes for issues identified in previous versions.

Type	Category	Description
Feature	Reliability and operability	Support in-place major upgrade of Postgres on a BDR node This BDR release include as new command-line utility <i>bdr_pg_upgrade</i> which uses <i>pg_upgrade</i> to do a major version upgrade of Postgres on a BDR node. This reduces the time and network bandwidth necessary to do major version upgrades of Postgres in a BDR cluster.
Feature	Performance and scalability	Replication Lag Control Add configuration for a replication lag threshold after which the transaction commits get throttled. This allows limiting RPO without incurring the latency impact on every transaction that comes with synchronous replication.
Feature	UX / Initial experience	Distributed sequences by default The default value of <i>bdr.default_sequence_kind</i> has been changed to <i>'distributed'</i> which is new kind of sequence that uses SnowflakeId for <i>bigserial</i> and Galloc sequences for <i>serial</i> column type.
Feature	UX	Simplified synchronous replication configuration New syntax for specifying the synchronous replication options, with focus on BDR groups and SQL based management (as opposed to config file). In future versions this will also replace the current Eager Replication and CAMO configuration options.
Feature	High availability and disaster recovery	Group Commit The initial kind of synchronous commit that can be configured via the new configuration syntax.
Feature	High availability and disaster recovery	Allow a Raft request to be required for CAMO switching to Local Mode (RT78928) Add a <i>require_raft</i> flag to the CAMO pairing configuration which controls the behavior of switching from CAMO protected to Local Mode, introducing the option to require a majority of nodes to be connected to allow to switch to Local Mode.
Feature	High availability and disaster recovery	Allow replication to continue on <i>ALTER TABLE ... DETACH PARTITION CONCURRENTLY</i> of already detached partition (RT78362) Similarly to how BDR 4 handles <i>CREATE INDEX CONCURRENTLY</i> when same index already exists, we now allow replication to continue when <i>ALTER TABLE ... DETACH PARTITION CONCURRENTLY</i> is receiver for partition that has been already detached.

continues on next page

Table 2 – continued from previous page

Type	Category	Description
Feature	User Experience	Add additional filtering options to DDL filters. DDL filters allow for replication of different DDL statements to different replication sets. Similar to how table membership in replication set allows DML on different tables to be replicated via different replication sets. This release adds new controls that make it easier to use the DDL filters: - query_match - if defined query must match this regex - exclusive - if true, other matched filters are not taken into consideration (i.e. only the exclusive filter is applied), when multiple exclusive filters match, we throw error
Feature	User Experience	Add <i>bdr.lock_table_locking</i> configuration variable. When enabled this changes behavior of <i>LOCK TABLE</i> command to take a global DML lock
Feature	Performance and scalability	Implement buffered write for LCR segment file This should reduce I/O and improve CPU usage of the Decoding Worker.
Feature	User Experience	Add support for partial unique index lookups for conflict detection (RT78368). Indexes on expression are however still not supported for conflict detection.
Feature	User Experience	Add additional statistics to <i>bdr.stat_subscription</i> : - nstream_insert => the count of INSERTs on streamed transactions - nstream_update => the count of UPDATEs on streamed transactions - nstream_delete => the count of DELETEs on streamed transactions - nstream_truncate => the count of TRUNCATEs on streamed transactions - npre_commit_confirmations => the count pre-commit confirmations, when using CAMO - npre_commit => the count of pre-commits - ncommit_prepared => the count of prepared commits with 2PC - nabort_prepared => the count of aborts of prepared transactions with 2PC
Feature	User Experience	Add <i>execute_locally</i> option to <i>bdr.replicate_ddl_command</i> (RT73533). This allows optional queueing of ddl commands for replication to other groups without executing it locally.
Feature	User Experience	Add <i>fast</i> argument to <i>bdr.alter_subscription_disable()</i> (RT79798) The argument only influences the behavior of <i>immediate</i> . When set to <i>true</i> (default) it will stop the workers without letting them finish the current work.

continues on next page

Table 2 – continued from previous page

Type	Category	Description
Feature	User Experience	Keep the <i>bdr.worker_error</i> records permanently for all types of workers. BDR used to remove receiver and writer errors when those workers managed to replicate the LSN that was previously resulting in error. However this was inconsistent with how other workers behaved, as other worker errors were permanent and it also made the troubleshooting of past issues harder. So keep the last error record permanently for every worker type.
Feature	User Experience	Simplify <i>bdr.{add,remove}_camo_pair</i> functions to return void.
Feature	Initial Experience	Add connectivity/lag check before taking global lock. So that application or user does not have to wait for minutes to get lock timeout when there are obvious connectivity issues. Can be set to DEBUG, LOG, WARNING (default) or ERROR.
Feature	Initial Experience	Only log conflicts to conflict log table by default. They are no longer logged to the server log file by default, but this can be overridden.
Feature	User Experience	Improve reporting of remote errors during node join.
Feature	Reliability and operability	Make autopartition worker's max naptime configurable.
Feature	User Experience	Add ability to request partitions upto the given upper bound with autopartition.
Feature	Initial Experience	Don't try replicate DDL run on subscribe-only node. It has nowhere to replicate so any attempt to do so will fail. This is same as how logical standbys behave.
Feature	User Experience	Add <i>bdr.accept_connections</i> configuration variable. When <i>false</i> , walsender connections to replication slots using BDR output plugin will fail. This is useful primarily during restore of single node from backup.
Bug fix	Reliability and operability	Keep the <i>lock_timeout</i> as configured on non-CAMO-partner BDR nodes A CAMO partner uses a low <i>lock_timeout</i> 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 <i>lock_timeout</i> errors on pglogical writer processes on normal BDR nodes.
Bug fix	User Experience	Show a proper wait event for CAMO / Eager confirmation waits (RT75900) Show correct "BDR Prepare Phase"/"BDR Commit Phase" in <i>bdr.stat_activity</i> instead of the default "unknown wait event".
Bug fix	User Experience	Reduce log for <i>bdr.run_on_nodes</i> (RT80973) Don't log when setting <i>bdr.ddl_replication</i> to off if it's done with the "run_on_nodes" variants of function. This eliminates the flood of logs for monitoring functions.
Bug fix	Reliability and operability	Fix replication of arrays of composite types and arrays of builtin types that don't support binary network encoding

continues on next page

Table 2 – continued from previous page

Type	Category	Description
Bug fix	Reliability and operability	Fix replication of data types created during bootstrap
Bug fix	Performance and scalability	Confirm end LSN of the running transactions record processed by WAL decoder so that the WAL decoder slot remains up to date and WAL sender get the candidate in timely manner.
Bug fix	Reliability and operability	Don't wait for autopartition tasks to complete on parting nodes
Bug fix	Reliability and operability	Limit the <i>bdr.standby_slot_names</i> check when reporting flush position only to physical slots (RT77985, RT78290) Otherwise flush progress is not reported in presence of disconnected nodes when using <i>bdr.standby_slot_names</i> .
Bug fix	Reliability and operability	Request feedback reply from walsender if we are close to <i>wal_receiver_timeout</i>
Bug fix	Reliability and operability	Don't record dependency of auto-partitioned table on BDR extension more than once. This resulted in "ERROR: unexpected number of extension dependency records" errors from auto-partition and broken replication on conflicts when this happens. Note that existing broken tables need to still be fixed manually by removing the double dependency from <i>'pg_depend'</i>
Bug fix	Reliability and operability	Improve keepalive handling in receiver. Don't update position based on keepalive when in middle of streaming transaction as we might lose data on crash if we do that. There is also new flush and signalling logic that should improve latency in low TPS scenarios.
Bug fix	Reliability and operability	Only do post <i>CREATE</i> commands processing when BDR node exists in the database.
Bug fix	Reliability and operability	Don't try to log ERROR conflicts to conflict history table.
Bug fix	Reliability and operability	Fixed segfault where a conflict_slot was being used after it was released during multi-insert (COPY) (RT76439).
Bug fix	Reliability and operability	Prevent walsender processes spinning when facing lagging standby slots (RT80295, RT78290). Correct signaling to reset a latch so that a walsender process does consume 100% of a CPU in case one of the standby slots is lagging behind.
Bug fix	Reliability and operability	Fix handling of <i>wal_sender_timeout</i> when <i>bdr.standby_slot_names</i> are used (RT78290)
Bug fix	Reliability and operability	Make ALTER TABLE lock the underlying relation only once (RT80204).
Bug fix	User Experience	Fix reporting of disconnected slots in <i>bdr.monitor_local_replslots</i> . They could have been previously reported as missing instead of disconnected.

continues on next page

Table 2 – continued from previous page

Type	Category	Description
Bug fix	Reliability and operability	Fix apply timestamp reporting for down subscriptions in <i>bdr.get_subscription_progress()</i> function and in the <i>bdr.subscription_summary</i> that uses that function. It would report garbage value before.
Bug fix	Reliability and operability	Fix snapshot handling in various places in BDR workers.
Bug fix	User Experience	Be more consistent about reporting timestamps and LSNs as NULLs in monitoring functions when there is no available value for those.
Bug fix	Reliability and operability	Reduce log information when switching between writer processes.
Bug fix	Reliability and operability	Don't do superuser check when configuration parameter was specified on PG command-line. We can't do transactions there yet and it's guaranteed to be superuser changed at that stage.
Bug fix	Reliability and operability	Use 64 bits for calculating lag size in bytes. To eliminate risk of overflow with large lag.

Upgrades

This release supports upgrading from the following versions of BDR:

- 4.0.0 and higher
- 3.7.15
- 3.7.16

Please make sure you read and understand the process and limitations described in the Upgrade Guide before upgrading.

2.1.2 BDR 4.0.2

This is a maintenance release for BDR 4.0 which includes minor improvements as well as fixes for issues identified in previous versions.

Type	Category	Description
Improvement	Reliability and operability	Add <i>bdr.max_worker_backoff_delay</i> (BDR-1767) This changes the handling of the backoff delay to exponentially increase from <i>bdr.min_worker_backoff_delay</i> to <i>bdr.max_worker_backoff_delay</i> in presence of repeated errors. This reduces log spam and in some cases also prevents unnecessary connection attempts.
Improvement	User Experience	Add <i>execute_locally</i> option to <i>bdr.replicate_ddl_command()</i> (RT73533) This allows optional queueing of ddl commands for replication to other groups without executing it locally.

continues on next page

Table 3 – continued from previous page

Type	Category	Description
Improvement	User Experience	Change ERROR on consensus issue during JOIN to WARNING The reporting of these transient errors was confusing as they were also shown in bdr.worker_errors. These are now changed to WARNINGS.
Bug fix	Reliability and operability	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.
Bug fix	Reliability and operability	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.
Bug fix	User Experience	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.
Bug fix	Reliability and operability	Fix locking and snapshot use during node management in the BDR manager process (RT74789) When processing multiple actions in the state machine, make sure to reacquire the lock on the processed node and update the snapshot to make sure all updates happening through consensus are taken into account.
Bug fix	Reliability and operability	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.
Bug fix	User Experience	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.
Bug fix	Reliability and operability	Improve handling of various timeouts and sleeps in consensus This reduces the amount of new consensus votes needed when processing many consensus requests or time consuming consensus requests, for example during join of a new node.
Bug fix	Reliability and operability	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.
Bug fix	Reliability and operability	Limit the bdr.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 bdr.standby_slot_names.

continues on next page

Table 3 – continued from previous page

Type	Category	Description
Bug fix	Reliability and operability	Fix replication of data types created during bootstrap (BDR-1784)
Bug fix	Reliability and operability	Fix replication of arrays of builtin types that don't have binary transfer support (BDR-1042)
Bug fix	Reliability and operability	Prevent CAMO configuration warnings if CAMO is not being used (BDR-1825)

Upgrades

This release supports upgrading from the following versions of BDR:

- 4.0.0 and higher

The upgrade path from BDR 3.7 is not currently stable and needs to be considered beta. Tests should be performed with at least BDR 3.7.15.

Please make sure you read and understand the process and limitations described in the Upgrade Guide before upgrading.

2.1.3 BDR 4.0.1

This is a maintenance release for BDR 4.0 which includes minor improvements as well as fixes for issues identified in previous versions.

Type	Category	Description
Improvement	Reliability and operability	Reduce frequency of CAMO partner connection attempts. 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.
Improvement	Performance and scalability	Implement buffered read for LCR segment file (BDR-1422) Implement LCR segment file buffering so that multiple LCR chunks can be read at a time. This should reduce I/O and improve CPU usage of Wal Senders when using the Decoding Worker.
Improvement	Performance and scalability	Avoid unnecessary LCR segment reads (BDR-1426) BDR now attempts to only read new LCR segments when there is at least one available. This reduces I/O load when Decoding Worker is enabled.
Improvement	Performance and scalability	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.
Bug fix	Performance and scalability	Fix the parallel apply worker selection (BDR-1761) This makes parallel apply work again. In 4.0.0 parallel apply was never in effect due to this bug.

continues on next page

Table 4 – continued from previous page

Type	Category	Description
Bug fix	Reliability and operability	Fix Raft snapshot handling of <i>bdr.camo_pairs</i> (BDR-1753) The previous release would not correctly propagate changes to the CAMO pair configuration when they were received via Raft snapshot.
Bug fix	Reliability and operability	Correctly handle Raft snapshots from BDR 3.7 after upgrades (BDR-1754)
Bug fix	Reliability and operability	Upgrading a CAMO configured cluster taking into account the <i>bdr.camo_pairs</i> in the snapshot while still excluding the ability to perform in place upgrade of a cluster (due to upgrade limitations unrelated to CAMO).
Bug fix	Reliability and operability	Switch from CAMO to Local Mode only after time-outs (RT74892) Do not use the <i>catchup_interval</i> 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).
Bug fix	Reliability and operability	Fix replication set cache invalidation when published replication set list have changed (BDR-1715) In previous versions we could use stale information about which replication sets (and as a result which tables) should be published until the subscription has reconnected.
Bug fix	Reliability and operability	Prevent duplicate values generated locally by gallo sequence in high concurrency situations when the new chunk is used (RT76528) The gallo 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.
Bug fix	Reliability and operability	Address memory leak on streaming transactions (BDR-1479) For large transaction this reduces memory usage and I/O considerably when using the streaming transactions feature. This primarily improves performance of COPY replication.
Bug fix	Reliability and operability	Don't leave slot behind after PART_CATCHUP phase of node parting when the catchup source has changed while the node was parting (BDR-1716) When node is being removed (parted) from BDR group, we do so called catchup in order to forward any missing changes from that node between remaining nodes in order to keep the data on all nodes consistent. This requires an additional replication slot to be created temporarily. Normally this replication slot is removed at the end of the catchup phase, however in certain scenarios where we have to change the source node for the changes, this slot could have previously been left behind. From this version, this slot is always correctly removed.

continues on next page

Table 4 – continued from previous page

Type	Category	Description
Bug fix	Reliability and operability	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 a prolonged period of time. This is not recommended setup but the WAL accumulation was not intentional.
Bug fix	Reliability and operability	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.

Upgrades

This release supports upgrading from the following versions of BDR:

- 3.7.14
- 4.0.0 and higher

Please make sure you read and understand the process and limitations described in the Upgrade Guide before upgrading.

2.1.4 BDR 4.0.0

BDR 4.0 is a new major version of BDR and adopted with this release number is semantic versioning (for details see semver.org). The two previous major versions are 3.7 and 3.6.

Type	Category	Description
Feature	Compatibility	BDR on EDB Postgres Advanced 14 now supports following features which were previously only available on 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 - Eager Replication - synchronizes between the nodes of the cluster before committing a transaction to provide conflict free replication - Decoding Worker - separation of decoding into separate worker from wal senders allowing for better scalability with many nodes - Estimates for Replication Catch-up times - Timestamp-based Snapshots - providing consistent reads across multiple nodes for retrieving data as they appeared or will appear at a given time - Automated dynamic configuration of row freezing to improve consistency of UPDATE/DELETE conflicts resolution in certain corner cases - Assesment checks - Support for handling missing partitions as conflicts rather than errors - Advanced DDL Handling for NOT VALID constraints and ALTER TABLE

continues on next page

Table 5 – continued from previous page

Type	Category	Description
Feature	Compatibility	BDR on community version of PostgreSQL 12-14 now supports following features which were previously only available on EDB Postgres Advanced or EDB Postgres Extended: - Conflict-free Replicated Data Types - additional data types which provide mathematically proven consistency in asynchronous multi-master update scenarios - Column Level Conflict Resolution - ability to use per column last-update wins resolution so that UPDATES on different fields can be “merged” without losing either of them - Transform Triggers - triggers that are executed on the incoming stream of data providing ability to modify it or to do advanced programmatic filtering - Conflict triggers - triggers which are called when conflict is detected, providing a way to use custom conflict resolution techniques - CREATE TABLE AS replication - Parallel Apply - allow multiple writers to apply the incoming changes
Feature	Performance	Support streaming of large transactions. This allows BDR to stream a large transaction (greater than <i>logical_decoding_work_mem</i> in size) either to a file on the downstream or to a writer process. This ensures that the transaction is decoded even before it’s committed, thus improving parallelism. Further, the transaction can even be applied concurrently if streamed straight to a writer. This improves parallelism even more. When large transactions are streamed to files, they are decoded and the decoded changes are sent to the downstream even before they are committed. The changes are written to a set of files and applied when the transaction finally commits. If the transaction aborts, the changes are discarded, thus wasting resources on both upstream and downstream. Sub-transactions are also handled automatically. This feature is available on PostgreSQL 14, EDB Postgres Extended 13+ and EDB Postgres Advanced 14, see Choosing a Postgres distribution appendix for more details on which features can be used on which versions of Postgres.
Feature	Compatibility	The differences that existed in earlier versions of BDR between standard and enterprise edition have been removed. With BDR 4.0 there is one extension for each supported Postgres distribution and version, i.e., PostgreSQL v12-14, EDB Postgres Extended v12-14, and EDB Postgres Advanced 12-14. Not all features are available on all versions of PostgreSQL, the available features are reported via feature flags using either <i>bdr_config</i> command line utility or <i>bdr.bdr_features()</i> database function. See Choosing a Postgres distribution for more details.

continues on next page

Table 5 – continued from previous page

Type	Category	Description
Feature	User Experience	There is no pglogical 4.0 extension that corresponds to the BDR 4.0 extension. BDR no longer has a requirement for pglogical. This means also that only BDR extension and schema exist and any configuration parameters were renamed from <i>pglogical.</i> to <i>bdr.</i>
Feature	Initial experience	Some configuration options have change defaults for better post-install experience: - Parallel apply is now enabled by default (with 2 writers). Allows for better performance, especially with streaming enabled. - <i>COPY</i> and <i>CREATE INDEX CONCURRENTLY</i> are now streamed directly to writer in parallel (on Postgres versions where streaming is supported) to all available nodes by default, eliminating or at least reducing replication lag spikes after these operations. - The timeout for global locks have been increased to 10 minutes - The <i>bdr.min_worker_backoff_delay</i> now defaults to 1s so that subscriptions retry connection only once per second on error
Feature	Reliability and operability	Greatly reduced the chance of false positives in conflict detection during node join for table that use origin based conflict detection
Feature	Reliability and operability	Move configuration of CAMO pairs to SQL catalogs To reduce chances of misconfiguration and make CAMO pairs within the BDR cluster known globally, move the CAMO configuration from the individual node's postgresql.conf to BDR system catalogs managed by Raft. This for example can prevent against inadvertently dropping a node that's still configured to be a CAMO partner for another active node. Please see the Upgrades chapter for details on the upgrade process. This deprecates GUCs <i>bdr.camo_partner_of</i> and <i>bdr.camo_origin_for</i> and replaces the functions <i>bdr.get_configured_camo_origin_for()</i> and <i>get_configured_camo_partner_of</i> with <i>bdr.get_configured_camo_partner</i> .

Upgrades

This release supports upgrading from the following version of BDR:

- 3.7.13.1

Please make sure you read and understand the process and limitations described in the Upgrade Guide before upgrading.

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.



Fig. 1: node diagram

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

2.2.1 Basic Architecture

Multiple Groups

A BDR node is a member of at least one **NodeGroup**, 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.

Asynchronous, by default

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

Mesh Topology

BDR is structured around a mesh network where every node connects to every other node and all nodes exchange data directly with each other. There is no forwarding of data within BDR except in special circumstances such as node addition and node removal. Data may arrive from outside the BDR cluster or be sent onwards using 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, BDR clusters can use rolling upgrades, even for major versions of database software.

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

Limits

BDR 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 currently has hard limit of no more than 1000 active nodes as this is the current maximum Raft connections allowed.

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 BDR cluster is 3, because with 2 nodes the consensus stops working if one of the node stops working.

2.2.2 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 use 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.

2.2.3 Deployment

BDR 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.

2.2.4 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 [Column-level conflict detection](#).

2.3 Application use

Learn about the application from a user perspective.

2.3.1 Application behavior

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

BDRs, by default, replicate all changes from INSERT, UPDATE, DELETE and TRUNCATE operations from the source node to other nodes. Only the final changes are sent, after all triggers and rules are processed. For example, `INSERT ... ON CONFLICT UPDATE` sends either an insert or an update depending on what occurred on the origin. If an update or delete affects zero rows, then no changes are sent.

INSERT can be replicated without any preconditions.

For updates and deletes to replicate 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 an explicit REPLICATION IDENTITY defined on specific columns. If one of those isn't defined, a warning is generated, and later updates or deletes are explicitly blocked. If REPLICATION IDENTITY FULL is defined for a table, then a unique index isn't required. In that case, updates and deletes are allowed and use the first non-unique index that is live, valid, not deferred, and doesn't have expressions or WHERE clauses. Otherwise, a sequential scan is used.

You can use TRUNCATE even without a defined replication identity. Replication of TRUNCATE commands is supported, but take care when truncating groups of tables connected by foreign keys. When replicating a truncate action, the subscriber truncates the same group of tables that was truncated on the origin, either explicitly specified or implicitly collected by CASCADE, except in cases where replication sets are defined. See [Replication Sets](#) for further details and examples. This works 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 aren't part of the same (or any) replication set, then applying the truncate action on the subscriber fails.

Row-level locks taken implicitly by INSERT, UPDATE, and DELETE commands are replicated as the changes are made. Table-level locks taken implicitly by INSERT, UPDATE, DELETE, and TRUNCATE commands are also replicated. Explicit row-level locking (`SELECT ... FOR UPDATE/FOR SHARE`) by user sessions isn't replicated, nor are advisory locks. Information stored by transactions running in SERIALIZABLE mode isn't replicated to other nodes. The transaction isolation level of SERIALIZABLE is supported, but transactions aren't serialized across nodes in the presence of concurrent transactions on multiple nodes.

If DML is executed on multiple nodes concurrently, then potential conflicts might occur if executing with asynchronous replication. These must be either handled or avoided. Various avoidance mechanisms are possible, discussed in [Handling column conflicts using CRDT data types](#).

Sequences need special handling, described in `:ref:`bdr.sequences`<bdr.sequences>`.

Binary data in BYTEA columns is replicated normally, allowing “blobs” of data up to 1 GB in size. Use of the PostgreSQL “large object” facility isn’t supported in BDR.

Rules execute only on the origin node so aren’t executed during apply, even if they’re enabled for replicas.

Replication is possible only 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 result in an error. DML changes that are made through updatable views are resolved 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 are 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 you apply the change on the target node. You can specify for triggers to execute on both the origin node at execution time and on the target when it’s 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 aren’t 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 can assume other `search_path` settings that then fail when they execute on apply. To prevent this from occurring, resolve object references clearly using either only the default `search_path`, 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 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 doesn’t have any effect except where unique indexes are explicitly defined with nonmatching collation qualifiers. Row filters might be affected by differences in collations if collatable expressions were used.

BDR handling of very long “toasted” data in PostgreSQL is transparent to the user. The TOAST “chunkid” values likely differ between the same row on different nodes, but that doesn’t cause any problems.

BDR can’t 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 causes data divergence.

BDR doesn’t restrict the use of foreign keys. Cascading FKs are allowed.

2.3.2 Nonreplicated 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)

Since the NOTIFY SQL command and the `pg_notify()` functions aren't replicated, notifications aren't reliable in case of failover. This means that notifications can easily be lost at failover if a transaction is committed just when the server crashes. Applications running LISTEN might miss notifications in case of failover. This is true in standard PostgreSQL replication, and BDR doesn't yet improve on this. CAMO and Eager Replication options don't allow the NOTIFY SQL command or the `pg_notify()` function.

2.3.3 DML and DDL replication

BDR doesn't replicate the DML statement. It replicates the changes caused by the DML statement. For example, an UPDATE that changed two rows replicates two changes, whereas a DELETE that didn't remove any rows doesn't replicate anything. This means that the results of executing volatile statements are replicated, ensuring there's no divergence between nodes as might occur with statement-based replication.

DDL replication works differently to DML. For DDL, BDR replicates the statement, which then executes 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 [DML and DDL replication](#).

BDR works to ensure that intermixed DML and DDL statements work correctly, even in the same transaction.

2.3.4 Replicating between different release levels

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

BDR is also designed to replicate between nodes that have different versions of BDR software. This feature is 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't add a node with a minor version if the cluster uses a newer protocol version. This returns an error.

Both of these features might be affected by specific restrictions. See [Release Notes](#) for any known incompatibilities.

2.3.5 Replicating between nodes with differences

By default, DDL is automatically sent to all nodes. You can control this manually, as described in [DML and DDL replication](#), and you could use it to create differences between database schemas across nodes. BDR is designed to allow replication to continue even with minor differences 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 might allow a mapping between different table names.

It is possible to replicate between tables with dissimilar partitioning definitions, such as a source that 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 doesn't need to execute at apply time. For details, see [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 were defined that allow that.

BDR supports replicating between tables that have a different number of columns.

If the target has missing columns from the source, then BDR raises a `target_column_missing` conflict, for which the default conflict resolver is `ignore_if_null`. This throws an error if a non-NULL value arrives. Alternatively, you can also configure a node with a conflict resolver of `ignore`. This setting doesn't throw an error but silently ignores any additional columns.

If the target has additional columns not seen in the source record, then BDR raises a `source_column_missing` conflict, for which the default conflict resolver is `use_default_value`. Replication proceeds if the additional columns have a default, either NULL (if nullable) or a default expression, but throws an error and halts 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 is attempted, but it might fail if the rows from source can't be applied to the target. Row filters can 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 can 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 can't be entered. That way, no bad data ever arrives by replication, so it never fails 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 to enforce that the text is XML.

You can define a table with different indexes on each node. By default, the index definitions are replicated. See [DML and 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`, can 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 must be owned by the same user/role on each node. See [Security and Roles](#) for further discussion.

Roles can have different passwords for connection on each node, although by default changes to roles are replicated to each node. See

[DML and DDL replication](#) to specify how to alter a role password only on a subset of nodes or locally.

2.3.6 Comparison between nodes with differences

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

Since LiveCompare 1.3, you can configure with `all_bdr_nodes` set. This saves you from clarifying all the relevant DSNs for each separate node in the cluster. A BDR cluster has N amount of nodes with connection information, but it's only the initial and output connection that LiveCompare 1.3+ needs to complete its job. Setting `logical_replication_mode` states how all the nodes are communicating.

All the configuration is done in a `.ini` file, named `bdrLC.ini`, for example. Find templates for this configuration file in `/etc/2ndq-livecompare/`.

While LiveCompare executes, you see N+1 progress bars, N being the number of processes. Once all the tables are sourced, a time displays, as the transactions per second (tps) was measured. This continues to count the time, giving you an estimate and then a total execution time at the end.

This tool offers a lot of customization and filters, such as tables, schemas, and `replication_sets`. LiveCompare can use stop-start without losing context information, so it can run at convenient times. After the comparison, a summary and a DML script are generated so you can review it. Apply the DML to fix the any differences found.

2.3.7 General rules for applications

BDR uses replica identity values to identify the rows to change. Applications can cause difficulties if they insert, delete, and then later reuse the same unique identifiers. This is known as the

ABA problem. BDR can't know whether the rows are the current row, the last row, or much older rows.

Similarly, since BDR uses table names to identify the table against which changes are replayed, a similar ABA problem exists with applications that create, drop, and then later reuse the same object names.

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

- Use unique identifiers for rows (INSERT).
- Avoid modifying unique identifiers (UPDATE).
- Avoid reusing deleted unique identifiers.
- Avoid reusing 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: while anomalies are unlikely, they aren't impossible. For example, a row value can be reused as long as the DELETE was replayed on all nodes, including down nodes. This might normally occur in less than a second but can take days if a severe issue occurred on one node that prevented it from restarting correctly.

2.3.8 Timing considerations and synchronous replication

Being asynchronous by default, peer nodes might lag behind, making it 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. See *Durability and performance options* for an overview and comparison of all variants available and its different modes.

2.3.9 Application testing

You can test BDR applications 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 Postgres-BDR.

TPAexec includes test suites for each reference architecture. It also simplifies creating and managing a local collection of tests to run against a TPA cluster, using a syntax like the following:

```
tpaexec test mycluster mytest
```

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

pgbench with CAMO/Failover options

In EDB Postgres Extended, the pgbench was extended to allow users to run failover tests while using CAMO or regular BDR deployments. The following options were added:

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

- --retry
retry transactions on failover
```

In addition to these 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. You can use The `-m failover` specification to test failover in regular BDR deployments.
- Use `--retry` to specify whether to retry transactions when failover happens with `-m failover` mode. This option is enabled by default for `-m camo` mode.

Here's an example in a CAMO environment:

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

This command runs in camo mode. It connects to node1 and runs the tests. If the connection to node1 is lost, then pgbench connects to node2. It queries node2 to get the status of in-flight transactions. Aborted and in-flight transactions are retried in camo mode.

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

isolationtester with multi-node access

isolationtester was extended to allow users to run tests on multiple sessions and on multiple nodes. This is used for internal BDR testing, although it's 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 for examining concurrent behaviors in PostgreSQL. These tests require running multiple interacting transactions, which requires managing 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. Tests for other sorts of concurrent behaviors were added as well.

It's built using PGXS as an external module. On installation, it creates the `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 was modified to accept multiple hosts as parameters. It then passes the host conninfo along with server names to the `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. 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 for the sessions to run on. There can be zero or more server "<name>" specifications. The conninfo corresponding to the names is provided by 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` can't 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, such as 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, but they are executed in each session. The setup part typically contains a `BEGIN` command to begin a transaction.

A session part also consists of `connect_to` specification. This points to a server name specified in the beginning that 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's 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). The list of steps in a manually specified “permutation” line doesn’t actually have to be a permutation of the available steps. It can, for instance, repeat some steps more than once or leave others out.

Lines beginning with a # are comments.

For each permutation of the session steps (whether these are manually specified in the spec file or automatically generated), the isolation tester runs:

1. The main setup part
2. Per-session setup parts
3. The selected session steps
4. Per-session teardown
5. The main teardown script

Each selected step is sent to the connection associated with its session.

To run isolation tests in a BDR environment that ran all prerequisite make commands:

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
```

To run `isolationcheck-installcheck`, you need to have two or more postgresql servers running. Pass the conninfo of each server to `pg_isolation_regress` in the BDR makefile. Ex: `pg_isolation_regress --server 'd1=host=myhost dbname=mydb port=5434' --server 'd2=host=myhost1 dbname=mydb port=5432'`

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

Then run `make isolationcheck-installcheck`

`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 and the conninfo of bdr instances to `pg_isolation_regress` in the 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 can 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, that is, those that don’t expect a blocked session to execute a command. If a test doesn’t follow that rule, isolationtester cancels it after 300 seconds. If the cancel doesn’t work, isolationtester exits uncleanly after 375 seconds of wait time. Avoid testing invalid permutations because they can make the isolation tests take a very long time to run, and they serve no useful testing purpose.

isolationtester recognizes that a command has blocked by checking whether it is shown as waiting in the `pg_locks` view. Therefore, only blocks on heavyweight locks are detected.

2.3.10 Performance testing and 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.

First, replaying changes from another node has a CPU cost, an I/O cost, and it generates WAL records. The resource use is usually less than in the original transaction since CPU overheads are lower as a result of not needing to reexecute SQL. In the case of UPDATE and DELETE transactions, there might 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 conflict-free replicated data types (CRDT) and column-level conflict detection (CLCD) features ensure you get the correct answers even for concurrent updates, but they don't remove the normal locking overheads. If you get locking contention, try to avoid conflicting updates or keep transactions as short as possible. A heavily updated row in a larger transaction causes a bottleneck on performance for that transaction. Complex applications require some thought to maintain scalability.

If you think you're having performance problems, develop performance tests using the benchmarking tools. `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.

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 shows you the effect of moving to multiple nodes.
4. Increase the number of sessions for these two 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.

2.3.11 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 easily be modified to become BDR compliant. You can undertake an assessment activity in which you can point your application to a BDR-enabled setup. BDR provides a few knobs that can be set during the assessment period. These aid in the process of deciding suitability of your application in a BDR-enabled environment.

Assessing updates of primary key/replica identity

BDR can't currently perform conflict resolution where the PRIMARY KEY is changed by an UPDATE operation. You can update the primary key, but you must ensure that no conflict with existing values is possible.

BDR provides the following configuration parameter to assess how frequently the primary key/replica identity of any table is being subjected to update operations.

Use these configuration parameters only for assessment. You can use them on a single node BDR instance, but don't use them on a production BDR cluster with two or more nodes replicating to each other. In fact, a node might fail to start or a new node 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, you can log updates to the key/replica identity values of a row. You can also potentially block such updates, if desired. For example:

```
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 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're 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.

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 you can track 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, you can track (or block) such explicit locking activity. For example:

```
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

Several PostgreSQL configuration parameters affect BDR nodes. You can set these parameters differently on each node, although that isn't generally recommended.

2.4.1 PostgreSQL settings for BDR

BDR requires these PostgreSQL settings to run correctly:

- `wal_level` — Must be set to `logical`, since BDR relies on logical decoding.
- `shared_preload_libraries` — Must contain `bdr`, although it can contain other entries before or after, as needed. However, don't include `pglogical`.
- `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 overflow the buffer and are stored temporarily on local disk. Default is 64 MB, but you can set it much higher.
- `max_worker_processes` — BDR uses background workers for replication and maintenance tasks, so you need enough worker slots for it to work correctly. The formula for the correct minimal number of workers, for each database, is: - One per PostgreSQL instance plus - One per database on that instance plus - Four per BDR-enabled database plus - One per peer node in the BDR group plus - One for each writer-enabled per peer node in the BDR group. You might need more worker processes temporarily when a 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` — Determines how quickly a node considers its CAMO partner as disconnected or reconnected. See CAMO failure scenarios for details.

In normal running for a group with N peer nodes, BDR requires N slots and WAL senders. During synchronization, BDR temporarily uses another $N - 1$ slots and WAL senders, so be careful to set the parameters high enough for this occasional peak demand.

With parallel apply turned on, the number of slots must be increased to N slots from the formula $\times$ writers. This is because the `max_replication_slots` also sets the maximum number of replication origins, and some of the functionality of parallel apply uses extra origin per writer.

When the *Decoding Worker* is enabled, this process requires one extra replication slot per BDR group.

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

Changing these parameters requires restarting the local node: `max_worker_processes`, `max_wal_senders`, `max_replication_slots`.

You might also want your applications to set these parameters. See *Durability and performance options* for details.

- `synchronous_commit` — Affects the durability and performance of BDR replication. in a similar way to *physical replication*.
- `synchronous_standby_names` — Same as above.

2.4.2 BDR-specific settings

You can also set BDR-specific configuration settings. Unless noted otherwise, you can set the values 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* . The default value is `distributed` , which means `snowflakeid` is used for `int8` sequences (i.e., `bigserial`) and `galloc` sequence for `int4` (i.e., `serial`) and `int2` sequences.

DDL handling

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

The accepted values are:

```
- `DEFAULT` &mdash; Records the old values of the columns of the primary key, if any
↪(this is the default PostgreSQL behavior). - `FULL` &mdash; Records the old values
↪of all columns in the row. - `NOTHING` &mdash; Records no information about the old
↪row.
```

See [PostgreSQL documentation](#) for more details.

BDR can't replicate `UPDATE` and `DELETE` operations on tables without a `PRIMARY KEY` or `UNIQUE` constraint. The exception is when the replica identity for the table is `FULL` , either by table-specific configuration or by `bdr.default_replica_identity` .

If `bdr.default_replica_identity` is `DEFAULT` and there is a `UNIQUE` constraint on the table, it isn't automatically picked up as `REPLICA IDENTITY` . You need to set it explicitly when creating the table or after, as described above.

Setting the replica identity of tables 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 is on).

This parameter can be set only 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 [DML and DDL replication](#) 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 is on). Only a superuser can set this parameter. This setting works only 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 might cause replicated DDL to interrupt replication until the administrator intervenes.

See *Role manipulation statements* for details.

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

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

Possible options are:

```
- off &mdash; Don't use global locking for DDL operations. - on &mdash; Use global
locking for all DDL operations. - dml &mdash; Use global locking only 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's normal for some statements to result in two `WARNING` messages: 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). Can be set only at Postgres server start.
- `bdr.global_lock_timeout` — Sets the maximum allowed duration of any wait for a global lock (default 10 minutes). 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 60 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.
- `bdr.predictive_checks` — Log level for predictive checks (currently used only by global locks). Can be `DEBUG`, `LOG`, `WARNING` (default), or `ERROR`. Predictive checks are early validations for expected cluster state when doing certain operations. You can use them for those operations for fail early rather than wait for timeouts. In global lock terms, BDR checks that there are enough nodes connected and withing reasonable lag limit for getting quorum needed by the global lock.

Node management

- `bdr.replay_progress_frequency` — Interval for sending replication position info to the rest of the cluster (default 1 minute).
- `bdr.standby_slot_names` — Require these slots to receive and confirm replication changes before any other ones. This setting is useful primarily when using physical standbys for failover or when using subscribe-only nodes.

Generic replication

- `bdr.writers_per_subscription` — Default number of writers per subscription (in BDR, you can also change this with `bdr.alter_node_group_config` for a group).
- `bdr.max_writers_per_subscription` — Maximum number of writers per subscription (sets upper limit for the setting above).
- `bdr.xact_replication` — Replicate current transaction (default is `on`).

Turning this off makes the whole transaction local only, which means the transaction isn't visible to logical decoding by BDR and all other downstream targets of logical decoding. Data isn't transferred to any other node, including logical standby nodes.

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

This parameter can be set only inside the current transaction using the `SET LOCAL` command unless `bdr.permit_unsafe_commands = on`.

Note: Even with transaction replication disabled, WAL is generated, but those changes are filtered away on the origin.

Warning: Turning off `bdr.xact_replication` leads to data inconsistency between nodes. Use it only 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 can either produce inconsistent results or break replication altogether. Use at your own risk.

- `bdr.batch_inserts` — Number of consecutive inserts to one table in 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 is also how the initial data during node join is copied.

- `bdr.maximum_clock_skew`

This option specifies the maximum difference between the incoming transaction commit timestamp and the current time on the subscriber before triggering `bdr.maximum_clock_skew_action`.

It checks if the timestamp of the currently replayed transaction is in the future compared to the current time on the subscriber. If it is, and the difference is larger than `bdr.maximum_clock_skew`, it performs 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's 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` &mdash; 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` &
↪mdash; Wait until the current local timestamp is no longer older than remote commit
↪timestamp minus the `bdr.maximum_clock_skew` .
```

- `bdr.accept_connections` — Option to enable or disable connections to BDR. Defaults to on .

Requires `bdr_superuser` or PostgreSQL superuser.

`bdr.standby_slot_names`

This option is typically used in failover configurations to ensure that the failover-candidate streaming physical replicas for this BDR node have received and flushed all changes before they ever become visible to subscribers. That guarantees that a commit can't vanish on failover to a standby for the provider.

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

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

Any replication slot can be listed in `bdr.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 and then vanish from the provider on failover because the failover candidate hadn't received it yet:

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

Setting `bdr.standby_slot_names` by design causes subscribers to lag behind the provider if the provider's failover-candidate replicas aren't keeping up. Monitoring is thus essential.

Another use case where the `bdr.standby_slot_names` is useful is when using subscriber-only, to ensure that the subscriber-only node doesn't move ahead of any of the other BDR nodes.

`bdr.standby_slots_min_confirmed`

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

`bdr.writer_input_queue_size`

This option specifies 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 1 MB, and the maximum allowed size is 1 GB. While any storage size specifier can be used to set the GUC, the default is KB.

bdr.writer_output_queue_size

This option specifies the size of the shared memory queue used by the receiver to receive data from the writer process. Since the writer isn't expected to send a large amount of data, a relatively smaller sized queue is enough. The default is 32 KB, and the maximum allowed size is 1 MB. While any storage size specifier can be used to set the GUC, the default is KB.

bdr.min_worker_backoff_delay

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

Time-unit suffixes are supported.

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

The default for `bdr.min_worker_backoff_delay` is 1 second. For `bdr.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 for reset. Additionally, the `bdr.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 isn't persistent. It is cleared by PostgreSQL restarts, including soft restarts during crash recovery after an unclean backend exit.

You can use the view `:ref:`bdr.worker_tasks` <bdr.worker_tasks>` 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 doesn't apply, for example, manager workers don't have a subscription ID and receivers don't have a writer ID.

CRDTs

- `bdr.crdt_raw_value` — Sets the output format of *Handling column conflicts using 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 prevents all Eager transactions on the cluster. You can set this only at Postgres server start.

Eager Replication

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

Commit At Most Once

- `bdr.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 backward-compatibility, the values `on`, `true`, and `1` set the safest `remote_commit_flush` mode, while `false` or `0` also disable CAMO.
- `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. Is usually unset. You can set it only 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. Set to `0` to disable this feature.
- `bdr.camo_enable_client_warnings` — Emit warnings if an activity is carried out in the database for which CAMO properties can't 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.
- `synchronous_replication_availability` — Can optionally be `async` for increased availability by allowing a node to continue and commit after its CAMO partner got disconnected. Under the default value of `wait`, the node waits indefinitely and proceeds to commit only after the CAMO partner reconnects and sends confirmation.

Transaction streaming

- `bdr.default_streaming_mode` — Used to control transaction streaming by the subscriber node. Permissible values are: `off`, `writer`, `file`, and `auto`. Defaults to `auto`. If set to `off`, the subscriber doesn't request transaction streaming. If set to one of the other values, the subscriber requests transaction streaming and the publisher provides it if it supports them and if configured at group level. For more details, see [Transaction streaming](#).

Lag control

- `bdr.lag_control_max_commit_delay` — Maximum acceptable post commit delay that can be tolerated, in fractional milliseconds.
- `bdr.lag_control_max_lag_size` — Maximum acceptable lag size that can be tolerated, in kilobytes.
- `bdr.lag_control_max_lag_time` — Maximum acceptable lag time that can be tolerated, in milliseconds.
- `bdr.lag_control_min_conforming_nodes` — Minimum number of nodes required to stay below acceptable lag measures.
- `bdr.lag_control_commit_delay_adjust` — Commit delay micro adjustment measured as a fraction of the maximum commit delay time. At a default value of 0.01%, it takes 100 net increments to reach the maximum commit delay.
- `bdr.lag_control_sample_interval` — Minimum time between lag samples and commit delay micro adjustments, in milliseconds.
- `bdr.lag_control_commit_delay_start` — The lag threshold at which commit delay increments start to be applied, expressed as a fraction of acceptable lag measures. At a default value of 1.0%, commit delay increments don't begin until acceptable lag measures are breached.

By setting a smaller fraction, it might be possible to prevent a breach by “bending the lag curve” earlier so that it's asymptotic with the acceptable lag measure.

Timestamp-based snapshots

- `snapshot_timestamp` — Turns on the use of *Timestamp-based snapshots* and sets the timestamp to use.
- `bdr.timestamp_snapshot_keep` — Time to keep valid snapshots for the timestamp-based snapshot use (default is 0, meaning don't keep past snapshots).

Monitoring and logging

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

Warning: Setting `bdr.debug_level` or `bdr.trace_level` to a value $\geq \text{log_min_messages}$ can produce a very large volume of log output, so don't enable it long term in production unless plans are in place for log filtering, archival, and rotation to prevent disk space exhaustion.

- `bdr.track_subscription_apply` — Track apply statistics for each subscription.
- `bdr.track_relation_apply` — Track apply statistics for each relation.
- `bdr.track_apply_lock_timing` — Track lock timing when tracking statistics for relations.

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 disables log compaction. You can set this only at Postgres server start.
.. Warning

If log compaction is disabled, the log grows in size forever.

- `bdr.raft_response_timeout` — To account for network failures, the Raft consensus protocol implemented times 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 can indicate a problem. Default value of this parameter is 3000 ms.
- `bdr.raft_log_min_message_duration` — When to log a consensus request. Measure roundtrip 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.raft_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. You can set it only at Postgres server start.
- `bdr.backwards_compatibility` — Specifies the version to be backward 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 in the configuration file unless the value is different from the current version.
- `bdr.track_replication_estimates` — Track replication estimates in terms of apply rates and catchup intervals for peer nodes. Protocols like CAMO can use this information to estimate the readiness of a peer node. This parameter is enabled by default.
- `bdr.lag_tracker_apply_rate_weight` — We monitor how far behind peer nodes are in terms of applying WAL from the local node and calculate a moving average of the apply rates for the lag tracking. This parameter specifies how much contribution newer calculated values have in this moving average calculation. Default value is 0.1.

2.5 Node Management

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

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

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

2.5.1 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.

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 extension. However, it's important to that all required database users and roles are created.

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 4.0 node can join a BDR cluster running 3.7.x at a specific minimum maintenance release (e.g. 3.7.6) or a mix of 3.7 and 4.0 nodes. This procedure is useful when user wants to upgrade not just the BDR major version but also the underlying PostgreSQL major version. This can be achieved by joining a 3.7 node running on PostgreSQL 12 or 13 to a BDR cluster running 3.6.x on PostgreSQL 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.

2.5.2 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.

2.5.3 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.

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

DML and 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.

2.5.4 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 = 'bdr'`, there can be other plugins in the list as well, but `pglogical` must not be included - `hot_standby = on` - `hot_standby_feedback = on`
- The Primary has: - `postgresql.conf`: - `bdr.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 `bdr.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.

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

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

Take special note of the `bdr.standby_slot_names` parameter as well. It is important to set in a BDR cluster where there is a Primary -> Physical Standby relationship or when using subscriber-only groups.

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.

2.5.5 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

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.

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 `bdr.standby_slot_names` and `bdr.standby_slots_min_confirmed` appropriately so that there is always a fully active BDR node that is ahead of the `subscriber-only` nodes.

This may 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.

2.5.6 Decoding Worker

BDR4 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.

- `bdr.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 `bdr.max_lcr_segment_file_size`, which defaults to 1GB.

2.5.7 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 Slots created by BDR* 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 Server and EDB Postgres Advanced Server, 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 logging*) 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 **mustnot** 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 logical replication 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
- keep the historical snapshot for timestamp based snapshots

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 BDR 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 `group20xxxxxxxxxxxxxxxxxx` (20 bytes long); the logical replication slot associated to node `a30xxxxxxxxxxxxxxxxxxxxxxxxxx` (30 bytes long) will be called:

`bdr_db20xxxx3597186_group20xbe9cbd0_a30xxxxxxxxxxxxx7f304a2`

...since `3597186` , `be9cbd0` and `7f304a2` are respectively the hashes of `db20xxxxxxxxxxxxxxxxxx` , `group20xxxxxxxxxxxxxxxxxx` and `a30xxxxxxxxxxxxxxxxxxxxxxxxxx` .

2.5.8 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 `SnowflakeId` or `gallocc`
- 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 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.

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

2.5.9 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 BDR cluster yet.
- **JOIN_START** : `bdr.join_node_group()` begins to join the local node to an existing BDR 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 BDR cluster.
- **PARTING** : Node disconnects from other nodes and plays no further part in consensus or replication.
- **PART_CATCHUP** : Non-parting nodes synchronize any missing data from the recently parted node.
- **PARTED** : Node parting operation is now complete on all nodes.

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

2.5.10 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` - Deprecated, will be removed in the future
- `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: BDR4 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 snowflakeid and 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,
```

(continues on next page)

(continued from previous page)

```
streaming_mode text DEFAULT NULL,
default_commit_scope text 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. **This option is deprecated and may be disabled or removed in future versions of BDR.**
- `num_writers` - number of parallel writers for subscription backing this node group, -1 means the default (as specified by the GUC `bdr.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 the Decoding Worker process cannot be enabled if `streaming_mode` is already enabled.
- `streaming_mode` - Enables/disables streaming of large transactions. When set to `off`, streaming is disabled. When set to any other value, large transactions are decoded while they are still in progress and the changes are sent to the downstream. If the value is set to `file`, then the incoming changes of streaming transactions are stored in a file and applied only after the transaction is committed on upstream. If the value is set to `writer`, then the incoming changes are directly sent to one of the writers, if available. If parallel apply is disabled or no writer is free to handle streaming transaction then the changes are written to a file and applied after the transaction is committed. If the value is set to `auto`, BDR will try to intelligently pick between `file` and `writer` depending on the transaction property and available resources. Note that `streaming_mode` cannot be enabled if the WAL decoder is already enabled.

For more details, see [Transaction streaming](#).

- `default_commit_scope` - the commit scope to use by default, initially the 'local' commit scope. This is only applicable to the top-level node group. Individual rules can be used for different origin groups of the same commit scope. See the section about

[Origin Groups](#) for Group Commit for more details.

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,
    fast boolean DEFAULT true
)
```

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.
- **fast** - This argument only influences the behavior of **immediate**. If set to **true** (the default) it stops all the workers associated with the disabled subscription without waiting for them to finish current work.

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 receive the stop without waiting for the COMMIT and if the **fast** argument is set to **true** the interruption of the workers will not wait for current work to finish.

2.5.11 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.

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 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.

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). This is feature of EDB Postgres Extended Server only.
- `-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.

2.6.1 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.

2.6.2 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.

2.6.3 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 and logging* chapter.

2.6.4 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 3.6, but not with BDR 3.7 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+.

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.

2.6.5 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.

2.6.6 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` (SQL-STATE `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.

2.6.7 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	Repl- cated	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

continues on next page

Order	Amount	Days	Level
-------	--------	------	-------

Command	Allowed	Replicated	Lock
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	:ref:`bdr.global_consensus_journal_details`<bdr.global_c	Y	DDL
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	:ref:`bdr.global_consensus_journal_details`<bdr.global_c	Y	DDL
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	:ref:`bdr.global_consensus_journal_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

Command	Allowed	Repl-icated	Lock
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
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	:ref:~bdr.global_Y_consensus_journal_details`<bdr.global_	Y	DDL
CREATE SERVER	Y	Y	DDL
CREATE STATISTICS	Y	Y	DDL
CREATE SUBSCRIPTION	Y	Y	DDL
CREATE SYNONYM	Y	Y	DDL
CREATE TABLE	:ref:~bdr.global_Y_consensus_journal_details`<bdr.global_	Y	DDL
CREATE TABLE AS	:ref:~bdr.global_Y_consensus_journal_details`<bdr.global_	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

66

Table 6 – continued from previous page

Command	Allowed	Repl- icated	Lock
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
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

continues on next page

Table 6 – continued from previous page

Command	Allowed	Repl- icated	Lock
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
DROP USER MAPPING	Y	Y	DDL
DROP VIEW	Y	Y	DDL
EXECUTE	Y	N	N
EXPLAIN	Y	:ref:``bdr.global_consensus_details`<bdr.global_	:ref:``bdr.global_consensus_details`<bdr.global_
FETCH	Y	N	N
GRANT	Y	:ref:``bdr.global_consensus_details`<bdr.global_	DDLbdr.global_consensus_details`<bdr.global_
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	:ref:``bdr.global_consensus_details`<bdr.global_
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	:ref:``bdr.global_consensus_details`<bdr.global_	DDLbdr.global_consensus_details`<bdr.global_
REVOKE ROLE	Y	Y	DDL
ROLLBACK	Y	N	N
ROLLBACK PREPARED	Y	N	N
SAVEPOINT	Y	N	N
SECURITY LABEL	Y	:ref:``bdr.global_consensus_details`<bdr.global_	DDLbdr.global_consensus_details`<bdr.global_
SELECT INTO	:ref:``bdr.global_consensus_details`<bdr.global_	Y	DDLbdr.global_consensus_details`<bdr.global_
SET	Y	N	N
SET CONSTRAINTS	Y	N	N
SHOW	Y	N	N
START TRANSACTION	Y	N	N
TRUNCATE TABLE	Y	:ref:``bdr.global_consensus_details`<bdr.global_	:ref:``bdr.global_consensus_details`<bdr.global_
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 gallo sequences (only). ALTER SEQUENCE ... SET SCHEMA is not supported on gallo 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 timestamptz. The cast between timestamp and timestamptz relies upon the time zone of the session and so is not immutable.

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

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
- ALTER TABLE ... ALTER COLUMN ... SET STATISTICS
- ALTER TABLE ... VALIDATE CONSTRAINT
- ALTER TABLE ... ATTACH PARTITION
- ALTER TABLE ... DETACH PARTITION
- ALTER TABLE ... ENABLE TRIGGER (ENABLE REPLICA TRIGGER will still take a DML lock)
- ALTER TABLE ... CLUSTER ON
- ALTER TABLE ... SET WITHOUT CLUSTER
- ALTER TABLE ... SET (storage_parameter = value [, ...])
- ALTER TABLE ... RESET (storage_parameter = [, ...])
- 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 BDR cluster. Note that because this involves a rewrite, the activity will still take the DML lock for a brief period, and thus requires that the whole cluster is available. With the above specifics in place, the rolling upgrade of the non-replicated alter activity can be carried out as below:

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

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

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

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

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

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 if all sub-commands are also allowed.

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 not replicated, but it may acquire the global DML lock when `bdr.lock_table_locking` is set on . The `bdr.global_lock_table()` function can also be used to request a global DML lock explicitly.

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

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.

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

Configuring and managing BDR does not require superuser access, nor is that recommended. The privileges required by BDR 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 BDR extension is installed. See BDR Default Roles 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.

2.7.1 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.

2.7.2 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.

2.7.3 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.

2.7.4 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.

2.7.5 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.

2.7.6 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.

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 `bdr_superuser` role
- 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 `bdr_superuser` role.

2.7.7 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 use 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.

2.7.8 BDR Default/Predefined Roles

BDR predefined roles are created when the BDR extension is installed. Note that after BDR 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_version`
- `bdr.bdr_version_num`
- `bdr.conflict_resolution_to_string`
- `bdr.conflict_type_to_string`
- `bdr.decode_message_payload`
- `bdr.get_global_locks`
- `bdr.get_raft_status`
- `bdr.get_relation_stats`
- `bdr.get_slot_flush_timestamp`
- `bdr.get_sub_progress_timestamp`
- `bdr.get_subscription_stats`
- `bdr.peer_state_name`
- `bdr.show_subscription_status`

bdr_monitor

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`
- `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.

2.7.9 Verification

BDR has been verified using the following tools and approaches.

Coverity

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

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

CIS Benchmark

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

	Result	Description
1.4	PASS	Ensure systemd Service Files Are Enabled
1.5	PASS	Ensure Data Cluster Initialized Successfully
2.1	PASS	Ensure the file permissions mask is correct
2.2	PASS	Ensure the PostgreSQL pg_wheel group membership is correct
3.1.2	PASS	Ensure the log destinations are set correctly
3.1.3	PASS	Ensure the logging collector is enabled
3.1.4	PASS	Ensure the log file destination directory is set correctly
3.1.5	PASS	Ensure the filename pattern for log files is set correctly
3.1.6	PASS	Ensure the log file permissions are set correctly
3.1.7	PASS	Ensure 'log_truncate_on_rotation' is enabled
3.1.8	PASS	Ensure the maximum log file lifetime is set correctly
3.1.9	PASS	Ensure the maximum log file size is set correctly
3.1.10	PASS	Ensure the correct syslog facility is selected
3.1.11	PASS	Ensure the program name for PostgreSQL syslog messages is correct
3.1.14	PASS	Ensure 'debug_print_parse' is disabled
3.1.15	PASS	Ensure 'debug_print_rewritten' is disabled
3.1.16	PASS	Ensure 'debug_print_plan' is disabled
3.1.17	PASS	Ensure 'debug_pretty_print' is enabled
3.1.18	PASS	Ensure 'log_connections' is enabled
3.1.19	PASS	Ensure 'log_disconnections' is enabled
3.1.21	PASS	Ensure 'log_hostname' is set correctly
3.1.23	PASS	Ensure 'log_statement' is set correctly
3.1.24	PASS	Ensure 'log_timezone' is set correctly
3.2	PASS	Ensure the PostgreSQL Audit Extension (pgAudit) is enabled
4.1	PASS	Ensure sudo is configured correctly
4.2	PASS	Ensure excessive administrative privileges are revoked
4.3	PASS	Ensure excessive function privileges are revoked
4.4	PASS	Tested Ensure excessive DML privileges are revoked
5.2	Not Tested	Ensure login via 'host' TCP/IP Socket is configured correctly
6.2	PASS	Ensure 'backend' runtime parameters are configured correctly
6.7	Not Tested	Ensure FIPS 140-2 OpenSSL Cryptography Is Used
6.8	PASS	Ensure SSL is enabled and configured correctly
7.3	PASS	Ensure WAL archiving is configured and functional

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

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

2.8 Conflicts

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

Conflicts aren't errors. In most cases, they are events that BDR can detect and resolve as they occur. Resolution depends on the nature of the application and the meaning of the data, so it's important that BDR provides the application a range of choices as to how to resolve conflicts.

By default, conflicts are resolved at the row level. When changes from two nodes conflict, either the local or remote tuple is picked and the other is discarded. For example, the commit timestamps might be compared for the two conflicting changes and the newer one kept. This approach ensures that all nodes converge to the same result and establishes commit-order-like semantics on the whole cluster.

Conflict handling is configurable, as described in *Enabling and disabling column-level conflict resolution*. Conflicts can be detected and handled differently for each table using conflict triggers, described in *Stream Triggers*.

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

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

- Conflict-free data types (CRDTs), described in `:ref:`bdr.crdt_handlers``<bdr.crdt\_handlers>``.
- Eager Replication, described in *Eager Replication*.

By default, all conflicts are logged to `bdr.conflict_history`. If conflicts are possible, then table owners must monitor for them and analyze how to avoid them or make plans to handle them regularly as an application task. The *Data verification with 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 to each other but can't support geographically distributed applications where very low latency is critical for acceptable performance.

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

2.8.1 How conflicts happen

Inter-node conflicts arise as a result of sequences of events that can't happen if all the involved transactions happen concurrently on the same node. Because the nodes exchange changes only after the transactions commit, each transaction is individually valid on the node it committed on. It isn't 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's a conflict between a transaction being applied and a transaction that was committed on the receiving node.

Most conflicts can't happen when all transactions run on a single node because Postgres has inter-transaction communication mechanisms to prevent it such as UNIQUE indexes, SEQUENCE operations, row and relation locking, and SERIALIZABLE dependency tracking. All of these mechanisms are ways to communicate between ongoing transactions to prevent undesirable concurrency issues.

BDR doesn't 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.

2.8.2 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 can't on a single node. BDR can detect most of those and applies the `update_if_newer` conflict resolver.

Row conflicts include:

- INSERT versus INSERT
- UPDATE versus UPDATE
- UPDATE versus DELETE
- INSERT versus UPDATE
- INSERT versus DELETE
- DELETE versus 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 operations 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 situation by retaining the most recently inserted tuple of the two according to the originating node's timestamps, unless this behavior is overridden by a user-defined conflict handler.

This conflict generates 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). You can configure other resolvers. See [Enabling and disabling column-level conflict resolution](#) for details.

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

You can effectively eliminate this type of conflict by using [BDR Global Sequences](#) .

INSERT operations 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 produces a `multiple_unique_conflicts` conflict.

In case of such a conflict, you must remove some rows for replication to continue. Depending on the resolver setting for `multiple_unique_conflicts` , the apply process either exits with error, skips the incoming row, or deletes some of the rows. The deletion tries 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 is always deleted.

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

UPDATE/UPDATE conflicts

Where two concurrent UPDATE operations 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're 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 changed, the `update_origin_change` conflict is generated. In all other cases, the UPDATE is normally applied without generating a conflict.

Both of these conflicts are resolved the same way as `insert_exists` , described in *INSERT/INSERT conflicts* .

UPDATE conflicts on the PRIMARY KEY

BDR can't currently perform conflict resolution where the PRIMARY KEY is changed by an UPDATE operation. You can 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 intervention.

Updating a primary key is possible in Postgres, but there are issues in both Postgres and BDR.

A simple schema provides an example that explains:

```
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;
```

However, suppose there are multiple rows in the table:

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

Some UPDATES succeed:

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

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

Other UPDATES fail with constraint errors:

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

So for Postgres applications that update primary keys, be 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)
```

Executing these next two changes concurrently causes a divergent error, since both changes are accepted. But applying the changes on the other node results in `update_missing` conflicts.

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

This scenario leaves 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)
```

You can identify and resolve this situation using *Data verification with LiveCompare* .

Concurrent conflicts present problems. Executing these two changes concurrently isn't easy to resolve:

```
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 requires manual resolution.

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

You can avoid divergence in cases 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 occurs even with `update_if_newer` , namely when two different rows both are updated concurrently to the same new primary key.

As a result, we recommend strongly against allowing primary key UPDATE operations in your applications, especially with BDR. If parts of your application 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 is always deleted.

UPDATE operations that violate multiple UNIQUE constraints

Like *INSERT operations that violate multiple UNIQUE constraints*, where an incoming UPDATE violates more than one UNIQUE index (or the PRIMARY KEY), BDR raises a `multiple_unique_conflicts` conflict.

BDR supports deferred unique constraints. If a transaction can commit on the source, then it applies cleanly on target, unless it sees conflicts. However, a deferred primary key can't be used as a REPLICA IDENTITY, so the use cases are already limited by that and the warning about using multiple unique constraints.

UPDATE/DELETE conflicts

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

If the deleted row is still detectable (the deleted row wasn't removed by VACUUM), the `update_recently_deleted` conflict is generated. By default the UPDATE is skipped, but you can configure the resolution for this. See *Enabling and disabling column-level 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 can't differentiate between UPDATE /DELETE conflicts and *INSERT/UPDATE conflicts* and generates the `update_missing` conflict.

Another type of conflicting DELETE and UPDATE is a DELETE that comes after the row was updated locally. In this situation, the outcome depends on the type of conflict detection used. When using the default, *Origin conflict detection*, no conflict is detected at all, leading to the DELETE being applied and the row removed. If you enable

Row version conflict detection, a `delete_recently_updated` conflict is generated.

The default resolution for this conflict type is to apply the DELETE and remove the row, but you can configure this or this can be handled by a conflict trigger.

INSERT/UPDATE conflicts

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

When this happens, the `update_missing` conflict is generated. The default conflict resolver is `insert_or_skip`, though you can use `insert_or_error` or `skip` instead. Resolvers that do insert-or-action 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

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

BDR can't currently detect this conflict type. The INSERT operation doesn't generate any conflict type and the INSERT is applied.

The DELETE operation always generates a `delete_missing` conflict, which is by default resolved by skipping the operation.

DELETE/DELETE conflicts

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

This always generates a `delete_missing` conflict, which is by default resolved by skipping the operation.

This conflict is harmless since both DELETE operations have the same effect. One of them can be safely ignored.

Conflicts with three or more nodes

If one node inserts a row that is then replayed to a second node and updated there, a third node can receive the UPDATE from the second node before it receives the INSERT from the first node. This scenario is an INSERT /UPDATE conflict.

These conflicts are handled by discarding the UPDATE . This can lead to different data on different nodes. These are *Divergent conflicts* .

This conflict type can happen only with three or more masters, of which at least two 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. However, enabling this option opens the door for INSERT /DELETE conflicts:

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, we recommend tuning 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 three 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 it simply logs `delete_missing` .

Later releases will automatically resolve INSERT /DELETE anomalies via rechecks using *Data verification with Live-Compare* when `delete_missing` conflicts occur. These can be performed manually by applications by checking the `bdr.conflict_history_summary` view.

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 primary key identifier of a table is reused

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

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

Applications that insert, delete, and then later reuse 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.

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

Any identifier reuse that occurs in the time interval it takes for changes to pass across the system causes difficulties. Although that time might be short in normal operation, down nodes can extend that interval to hours or days.

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

This problem doesn't occur in applications that use sequences or UUIDs.

Foreign key constraint conflicts

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

BDR applies changes with `session_replication_role = 'replica'` , so foreign keys aren't rechecked 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.

In single-master Postgres, any INSERT /UPDATE that refers to a value in the referenced table must wait for DELETE operations to finish before they can gain a row-level lock. If a DELETE removes a referenced value, then the INSERT /UPDATE fails the FK check.

In multi-master BDR, there are no inter-node row-level locks. An INSERT on the referencing table doesn't 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 use 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's no longer present in the referenced table.

In practice, this occurs if the DELETE operations occurs on referenced tables in separate transactions from DELETE operations on referencing tables. This isn't a common operation.

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

While there's a possibility of dangling FKs, the risk of this in general is very low and so BDR doesn't impose a generic solution to cover this case. Once you 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 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 use `FOREIGN KEY` information to avoid FK anomalies by setting referencing columns to `NULL`, much as if you had a `SET NULL` constraint. This rechecks all FKs in one trigger, so you need to add only one trigger per table to prevent FK violation.

As an example, suppose you have two tables: `Fact` and `RefData`. `Fact` has an FK that references `RefData`. `Fact` is the referencing table and `RefData` is the referenced table. You need to add one trigger to each table.

Add a trigger that sets columns to `NULL` in `Fact` if the referenced row in `RefData` was already 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 sets 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 avoids dangling foreign keys.

TRUNCATE conflicts

`TRUNCATE` behaves similarly to a `DELETE` of all rows but performs this action by physically removing the table data rather than row-by-row deletion. As a result, row-level conflict handling isn't available, so `TRUNCATE` commands don't generate conflicts with other DML actions, even when there's a clear conflict.

As a result, the ordering of replay can cause divergent changes if another DML is executed concurrently on other nodes to the `TRUNCATE`.

You can take one of the following actions:

- Ensure `TRUNCATE` isn't executed alongside other concurrent DML. Rely on *Data verification with LiveCompare* to highlight any such inconsistency.
- Replace `TRUNCATE` with a `DELETE` statement with no `WHERE` clause. This approach is likely to have very poor performance on larger tables.
- Set `bdr.truncate_locking = 'on'` to set the `TRUNCATE` command's locking behavior. This setting determines whether `TRUNCATE` obeys the `bdr.ddl_locking` setting. This isn't the default behavior for `TRUNCATE` since it requires all nodes to be up. This configuration might not be possible or wanted in all cases.

Exclusion constraint conflicts

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

If an existing standalone database is converted to a BDR database, then drop all exclusion constraints manually.

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

If you force BDR to create an exclusion constraint, or you don't drop existing ones when converting a standalone database to BDR, expect replication to break. To get it to progress again, remove or alter the local tuples 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 (Postgres-system-wide) data, like roles, that differ. This can result in operations—mainly DDL—that can run successfully and commit on one node but then fail to apply to other nodes.

For example, node1 might have a user named fred, and that user wasn't created on node2. If fred on node1 creates a table, the table is replicated with its owner set to fred. When the DDL command is applied to node2, the DDL fails because there's no user named fred. This failure emits an error in the Postgres logs.

Administrator intervention is required to resolve this conflict by creating the user fred in the database where BDR is running. You can 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're 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`, `INSERT`, or `DELETE` operations, either from local activity or from replication from other nodes

A BDR writer process can 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 can also deadlock with each other. Postgres's deadlock detector steps in and terminates one of the problem transactions. If the BDR writer process is terminated, it retries and generally succeeds.

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 can terminate the user session to get replication flowing again, but this is no different from a user holding a long lock that impacts another user session.

Use of the `log_lock_waits` facility in Postgres 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 don't arise so long as you use BDR as documented and avoid settings or functions marked as unsafe.

The administrator must manually resolve such conflicts. You might need to use the advanced options such as `bdr.ddl_replication` and `bdr.ddl_locking` depending on the nature of the conflict. However, careless use of these options can make things much worse and create a conflict that generic instructions can't address.

TOAST support details

Postgres uses out-of-line storage for larger columns called *TOAST support details*.

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

The TOAST value is 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 didn't change a value of a toasted column produces a row without that column. As mentioned in *INSERT/UPDATE conflicts*, BDR reports 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 this one in case of concurrent workloads with asynchronous replication (Eager transactions aren't affected). Imagine, for example, the following workload on a BDR cluster with three 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 conflicts (on `update_origin_change`) and is skipped.
6. Node C misses the toasted data from A1 forever.

This scenario isn't usually a problem when using BDR. (It is when using either built-in logical replication or plain pglogical for multi-master.) BDR adds its own logging of TOAST columns when it detects a local UPDATE to a row that recently replicated a TOAST column modification and the local UPDATE isn't modifying the TOAST. Thus BDR prevents any inconsistency for toasted data across different nodes. This situation causes increased WAL logging when updates occur on multiple nodes (that is, when origin changes for a tuple). Additional WAL overhead is 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 removes metadata needed for the extra logging to work. This means that, for a short period of time after such a statement, the protection against these concurrency issues isn't be present.

Warning: The additional WAL logging of TOAST is done using the *BEFORE UPDATE* trigger on standard Postgres. 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 a name that sorts after it. Otherwise you won't have the protection against the concurrency issues.

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`. This setting always logs a toasted value as part of the key since the whole row is considered to be part of the key. BDR can reconstruct the new row, filling the missing data from the key row. As a result, using `REPLICA IDENTITY FULL` can increase WAL size significantly.

2.8.3 Avoiding or tolerating conflicts

In most cases, you can design the application to avoid or tolerate conflicts.

Conflicts can happen only if things are happening at the same time on multiple nodes. 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 allow data to be changed only by the owning user, such as changing the default billing address on your account. Such data changes seldom have update conflicts.

You might make a change just before a node goes down, so the change seems to be lost. You might then make the same change again, leading to two updates on different nodes. When the down node comes back up, it tries to send the older change to other nodes, but it's rejected because the last update of the data is kept.

For `INSERT /INSERT` conflicts, use *BDR Global Sequences* to prevent this type of conflict.

For applications that assign relationships between objects, such as a room booking application, applying `update_if_newer` might not give an acceptable business outcome. That is, 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 on the application. For example, you can 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 are reserved.

Another technique for ensuring certain types of updates occur only from one specific node is to route different types of transactions through different nodes. For example:

- Receiving parcels on one node but delivering parcels using 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

Frequently, the best course is to allow conflicts to occur and design the application to work with BDR's conflict resolution mechanisms to cope with the conflict.

2.8.4 Conflict detection

BDR provides these mechanisms for conflict detection:

- *Origin conflict detection* (default)
- *Row version conflict detection*
- *Column-level conflict detection*

Origin conflict detection

Origin conflict detection uses and relies on commit timestamps as recorded on the node 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 isn't the case, conflict resolution tends to favor the node that's further ahead. You can manage clock skew between nodes using the parameters `bdr.maximum_clock_skew` and `bdr.maximum_clock_skew_action`.

Row origins are available only if `track_commit_timestamp = on`.

Conflicts are initially detected based on whether the replication origin changed, so conflict triggers are called in situations that might turn out not to be conflicts. Hence, this mechanism isn't 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 was frozen don't raise a conflict so are applied in all cases. This is the normal case when adding a new node by `bdr_init_physical`, so raising conflicts causes many false-positive results in that case.

When a node that was offline reconnects and begins sending data changes, this can cause divergent errors if the newly arrived updates are older than the frozen rows that they update. Inserts and deletes aren't affected by this situation.

We suggest that you don't leave down nodes for extended outages, as discussed in *Node Restart and Down Node Recovery*.

On EDB Postgres Extended Server and EDB Postgres Advanced Server, BDR holds back the freezing of rows while a node is down. This mechanism handles this situation gracefully so you don't need to change parameter settings.

On other variants of Postgres, you might 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`

Choose values based on the transaction rate, giving a grace period of downtime before removing any conflict data from the database node. For example, when `vacuum_freeze_min_age` is set to 500 million, a node performing 1000 TPS can be down for just over 5.5 days before conflict data is removed. The CommitTS data structure takes on-disk space of 5 GB with that setting, so lower transaction rate systems can benefit from lower settings.

Initially recommended settings are:

```
# 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:

- You can set `autovacuum_freeze_max_age` only at node start.
- You can set `vacuum_freeze_min_age`, so using a low value freezes rows early and can result in conflicts being ignored. You can also set `autovacuum_freeze_min_age` and `toast.autovacuum_freeze_min_age` for individual tables.
- Running the `CLUSTER` or `VACUUM FREEZE` commands also freezes rows early and can 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 that you enable three things. If any of these steps aren't performed correctly then *Origin conflict detection* is used.

1. `check_full_tuple` must be enabled for the BDR node group.
2. `REPLICA IDENTITY FULL` must be enabled on all tables that use row version conflict detection.
3. Row Version Tracking must be enabled on the table by using `bdr.alter_table_conflict_detection`. This function adds a column (with a name you specify) and an `UPDATE` trigger that manages the new column value. The column is 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 *Enabling and disabling column-level conflict resolution* configuration even with row versioning. The way the row version is generated is useful only for detecting conflicts. Don't rely on it 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** — The conflict detection method to use.
- **column_name** — The column to use for storing the column detection data. This can be skipped, in which case the column name is chosen based on the conflict detection method. The `row_origin` method doesn't require an 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 that doesn't require an extra column in the table.
- **row_version** — Row version column (see *Row version conflict detection*).
- **column_commit_timestamp** — Per-column commit timestamps (described in CLCD).
- **column_modify_timestamp** — Per-column modification timestamp (described in CLCD).

Notes

For more information about the difference between `column_commit_timestamp` and `column_modify_timestamp` conflict detection methods, see *Current versus commit timestamp*.

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

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

This function is transactional. You can roll back the effects 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 executed only by the owner of the `relation`, unless `bdr.backwards_compatibility` is set to 30618 or below.

Warning: When changing the conflict detection method from one that uses an extra column to store metadata, that column is dropped.

Warning: This function disables CAMO (together with a warning, as long as these aren't 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 a unique key/index.
- **update_differing** — An incoming update's key row differs from a local row. This can happen only 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 doesn't 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's 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 doesn't 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 Postgres when applying a replicated DDL command.

2.8.5 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 retains the most recently inserted or changed row of the two conflicting ones based on the same commit timestamps used for conflict detection. The behavior in certain corner-case scenarios depends on the settings used for `bdr.create_node_group` and alternatively for `bdr.alter_node_group`.

BDR lets you override the default behavior of conflict resolution by using the following function:

`bdr.alter_node_set_conflict_resolver`

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

Synopsis

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

Parameters

- `node_name` — Name of the node that's being changed.
- `conflict_type` — Conflict type for which to apply the setting (see *List of conflict types*).
- `conflict_resolver` — Resolver to use for the given conflict type (see *List of conflict resolvers*).

Notes

Currently you can change only the local node. The function call isn't replicated. If you want to change settings on multiple nodes, you must run the function on each of them.

The configuration change made by this function overrides any default behavior of conflict resolutions specified by `bdr.create_node_group` or `bdr.alter_node_group`.

This function is transactional. You can roll back the changes, and they are visible to the current transaction.

List of conflict resolvers

Several conflict resolvers are 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 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 doesn't exist downstream because it was recently (within one 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 can pose challenges if a table with the same name is re-created shortly after it's dropped. In that case, one of the nodes might see the DMLs on the re-created table before it sees the DDL to re-create the table. It then incorrectly skips the remote data, assuming that the table is recently dropped, and causes data loss. We hence recommend that you don't reuse the object names immediately after they are dropped along with this conflict resolver.
- **skip_transaction** — Skips the whole transaction that 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 node) 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` (turns 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 isn't 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 isn't enough information available to build full row, throw an 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 an 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) stops 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* .

This matrix helps you individuate the conflict types the conflict resolvers can handle.

	in- sert_exists	up- date_differing	up- date_origin_change	up- date_missing	up- update_recently_deleted	up- update_pkey_exists	delete_recently_updated	delete_missing`	insert_or_skip	insert_or_error	ignore	ignore_if_null	use_default_value	conflict_trigger
error	X	X	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	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
<code>insert_exists</code>	<code>update_if_newer</code>
<code>update_differing</code>	<code>update_if_newer</code>
<code>update_origin_change</code>	<code>update_if_newer</code>
<code>update_missing</code>	<code>insert_or_skip</code>
<code>update_recently_deleted</code>	<code>skip</code>
<code>update_pkey_exists</code>	<code>update_if_newer</code>
<code>multiple_unique_conflicts</code>	<code>error</code>
<code>delete_recently_updated</code>	<code>skip</code>
<code>delete_missing</code>	<code>skip</code>
<code>target_column_missing</code>	<code>ignore_if_null</code>
<code>source_column_missing</code>	<code>use_default_value</code>
<code>target_table_missing</code>	<code>skip_if_recently_dropped</code>
<code>apply_error_ddl</code>	<code>error</code>

List of conflict resolutions

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

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

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

2.8.6 Conflict logging

To ease the diagnosis and handling of multi-master conflicts, BDR, by default, logs every conflict into the `bdr.conflict_history` table. You can change this behavior 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,  
                             conflict_type text[] DEFAULT NULL,  
                             conflict_resolution text[] DEFAULT NULL)
```

Parameters

- `node_name` — Name of the node that's being changed.
- `log_to_file` — Whether to log to the node log file.
- `log_to_table` — Whether to log to the `bdr.conflict_history` table.
- `conflict_type` — Conflict types to log. NULL (the default) means all.
- `conflict_resolution` — Conflict resolutions to log. NULL (the default) means all.

Notes

Only the local node can be changed. The function call isn't replicated. If you want to change settings on multiple nodes, you must run the function on each of them.

This function is transactional. You can roll back the changes, and they 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 the conflict type and resolution it logs.

Logging conflicts to a table

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

This table is range partitioned on the column `local_time`. The table is managed by Autopartition. By default, a new partition is created for every day, and conflicts of the last one month are maintained. After that, the old partitions are dropped automatically. Autopartition creates between 7 and 14 partitions in advance. `bdr_superuser` can 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. BDR does 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 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 might not be readable even to the owner. So BDR also enforces 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 doesn't contain user data, allowing monitoring of any conflicts logged.

Conflict reporting

Conflicts logged to tables can be summarized in reports. Reports allow application owners to identify, understand, and resolve conflicts and 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)

2.8.7 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. By default, it compares all replicated tables and reports differences. LiveCompare also works with non-BDR data sources such as Postgres and Oracle.

You can also use LiveCompare to continuously monitor incoming rows. You can stop and start it without losing context information, so you can run it at convenient times.

LiveCompare allows concurrent checking of multiple tables. You can configure it 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. If different, LiveCompare then compares whole rows. LiveCompare avoids overheads by comparing rows in useful-sized batches.

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

Refer to the [Data verification with LiveCompare](#) documentation 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.

2.9.1 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 **globalsequences**.

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:

- `SnowflakeId` sequences
- Globally-allocated range sequences

`SnowflakeId` 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.

`SnowflakeId` 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.
- `snowflakeid` which creates global sequences for `BIGINT` sequences which consist of time, nodeid and counter components, cannot be used with `INTEGER` sequences (so it can be used for `bigserial` but not for `serial`).
- `timeshard` older version of `SnowflakeId` sequence which is provided for backwards compatibility only, the `SnowflakeId` is preferred

- distributed special value which can only be used for `bdr.default_sequence_kind` and will select `snowflakeid` for `int8` sequences (i.e. `bigserial`) and `galloc` sequence for `int4` (i.e. `serial`) and `int2` sequences.

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

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

SnowflakeId Sequences

The ids generated by `SnowflakeId` 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.

`SnowflakeId` 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.

`SnowflakeId` 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 `SnowflakeId` sequences.

`SnowflakeId` 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.

For `SnowflakeId` there is a limit of 4096 sequence values generated per millisecond on any given node (this means about 4 million sequence values per second). In case the sequence value generation wraps around within given millisecond, the `SnowflakeId` sequence will wait until next millisecond and get fresh value for that millisecond.

Since `SnowflakeId` sequences encode timestamp into sequence value, new sequence values can only be generated within given time frame (depending on system clock). The oldest timestamp which can be used 2016-10-07 which is the epoch time for the `SnowflakeId`. The values will wrap to negative values in year 2086 and completely run out of numbers by 2156.

Since timestamp is important part of `SnowflakeId` sequence, there is additional protection from generating sequences with older timestamp than the latest one used within the lifetime of postgres process (but not between postgres restarts).

The `INCREMENT` option on a sequence used as input for `SnowflakeId` 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 `SnowflakeId` sequences.

Timeshard sequences

Timeshard sequences are provided for backwards compatibility with existing installations but are not recommended for new application use. It's recommended to use the SnowflakeId sequence instead.

Timeshard is very similar to SnowflakeId, but has different limits and fewer protections and worse performance.

The differences between timeshard and SnowflakeId are as following:

- Timeshard can generate up to 16384 per millisecond (about 16 million per second) which is more than SnowflakeId, however there is no protection against wraparound within given millisecond so schemas using the timeshard sequence should protect use UNIQUE constraint when using timeshard values for given column. - The timestamp component of timeshard sequence 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. This is considerably shorter time span than SnowflakeId and is the main reason why SnowflakeId is preferred. - Timeshard sequences require occasional disk writes (similar to standard local sequences), while SnowflakeId are calculated in memory so the SnowflakeId sequences are in general a little faster than 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 SnowflakeId 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 `:ref:~bdr.sequence_alloc`<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.

Converting a local sequence to a galloc sequence

Before transforming a local sequence to galloc, you need to take care of several prerequisites.

1. Verify that sequence and column data type match

Check that the sequence's data type matches the data type of the column with which it will be used. For example, it is possible to create a `bigint` sequence and assign an `integer` column's default to the `nextval()` returned by that sequence. With galloc sequences, which for `bigint` are allocated in blocks of 1 000 000 000, this will quickly result in the values returned by `nextval()` exceeding the `int4` range if more than two nodes are in use.

The following example demonstrates what can happen:

```
CREATE SEQUENCE int8_seq;

SELECT sequencename, data_type FROM pg_sequences;
 sequencename | data_type
-----+-----
 int8_seq     | bigint
(1 row)

CREATE TABLE seqtest (id INT NOT NULL PRIMARY KEY);

ALTER SEQUENCE int8_seq OWNED BY seqtest.id;

SELECT bdr.alter_sequence_set_kind(public.int8_seq::regclass, galloc, 1);
 alter_sequence_set_kind
-----
(1 row)

ALTER TABLE seqtest ALTER COLUMN id SET DEFAULT nextval(int8_seq::regclass);
```

After executing `INSERT INTO seqtest VALUES (DEFAULT)` on two nodes, the table will contain the following values:

```
SELECT * FROM seqtest;
 id
-----
      2
20000000002
(2 rows)
```

However, attempting the same operation on a third node will fail with an `integer out of range` error, as the sequence will have generated the value `40000000002`. The next section contains more details on how chunks of sequences are allocated.

Tip: The current data type of a sequence can be retrieved from the PostgreSQL `pg_sequences` view. The data type of a sequence can be modified with `ALTER SEQUENCE ... AS ...`, e.g.: `ALTER SEQUENCE public.sequence AS integer`, as long as its current value has not exceeded the maximum value of the new data type.

2. Set a new start value for the sequence

When the 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(
      ESELECT 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 20000004 to 40000003 .

```

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.

2.9.2 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 pseudo-random 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, KSUUID values do not include the MAC of the computer on which they were generated, so there should be no security concerns from using KSUUIDs.

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

There are two versions of KSUUID in BDR, v1 and v2. The legacy KSUUID 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 KSUUID also no longer stores the UUID version number.

Step & Offset Sequences

In offset-step sequences, a normal PostgreSQL sequence is used on each node. Each sequence increments by the same amount and starts at differing offsets. For example with step 1000, node1's sequence generates 1001, 2001, 3001, and so on, node2's generates 1002, 2002, 3002, etc. This scheme works well even if the nodes cannot communicate for extended periods, but 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.

2.9.3 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 [DML and DDL replication](#).

Synopsis `sql 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, `snowflakeid` or `galloc` for globally unique BDR sequences, or `timeshard` for legacy globally unique sequence

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_snowflakeid

This function extracts the timestamp component of the `snowflakeid` sequence. The return value is of type “timestamp”.

Synopsis sql `bdr.extract_timestamp_from_snowflakeid(snowflakeid bigint)`

Parameters - `snowflakeid` - value of a `snowflakeid` sequence

Notes

This function is only executed on the local node.

bdr.extract_nodeid_from_snowflakeid

This function extracts the nodeid component of the `snowflakeid` sequence.

Synopsis sql `bdr.extract_nodeid_from_snowflakeid(snowflakeid bigint)`

Parameters - `snowflakeid` - value of a `snowflakeid` sequence

Notes

This function is only executed on the local node.

bdr.extract_localseqid_from_snowflakeid

This function extracts the local sequence value component of the `snowflakeid` sequence.

Synopsis sql `bdr.extract_localseqid_from_snowflakeid(snowflakeid bigint)`

Parameters - `snowflakeid` - value of a `snowflakeid` sequence

Notes

This function is only executed on the local node.

bdr.timestamp_to_snowflakeid

This function converts a timestamp value to a dummy `snowflakeid` sequence value.

This is useful for doing indexed searches or comparisons of values in the `snowflakeid` column and for a specific timestamp.

For example, given a table `foo` with a column `id` which is using a `snowflakeid` sequence, we can get the number of changes since yesterday midnight like this:

```
SELECT count(1) FROM foo WHERE id > bdr.timestamp_to_snowflakeid(yesterday)
```

A query formulated this way will use an index scan on the column `id`.

Synopsis `sql bdr.timestamp_to_snowflakeid(ts timestamptz)`

Parameters - `ts` - timestamp to be used for the snowflakeid sequence generation

Notes

This function is only executed on local node.

`bdr.extract_timestamp_from_timeshard`

This function extracts the timestamp component of the `timeshard` sequence. The return value is of type “timestamptz”.

Synopsis

```
bdr.extract_timestamp_from_timeshard(timeshard_seq bigint)
```

Parameters

- `timeshard_seq` - value of a timeshard sequence

Notes

This function is only executed on the local node.

`bdr.extract_nodeid_from_timeshard`

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.

2.9.4 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 - KSUUID 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 KSUUID v2. The return value is of type “timestamptz”.

Synopsis

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

Parameters

- UUID - KSUUID v2 value to extract timestamp from

Notes

This function is only executed on the local node.

2.9.5 KSUUID v1 Functions

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

bdr.gen_ksuuid

This function generates a new KSUUID v1 value, using the current system time. The return value is of type “UUID”.

Synopsis

```
bdr.gen_ksuuid()
```

Notes

This function is only executed on the local node.

bdr.uuid_v1_cmp

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 might 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 might 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
```

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 behavior, but for some this might 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 might step on each other's toes, overwriting their changes.

For such use cases, it might be more appropriate to resolve conflicts on a given table at the column level. To achieve that, BDR tracks the timestamp of the last change for each column separately and uses 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 can be useful to see tables as vertically partitioned, so that each update affects data in only one slice. This approach eliminates conflicts between changes to different subsets of columns. In fact, vertical partitioning can even be a practical alternative to column-level conflict resolution.

Column-level conflict resolution requires the table to have `REPLICA IDENTITY FULL` . The `bdr.alter_table_conflict_detection` function does check that and fails with an error otherwise.

2.10.1 Enabling and disabling column-level conflict resolution

The column-level conflict resolution is managed by the `bdr.alter_table_conflict_detection()` function.

Example

To see how the `bdr.alter_table_conflict_detection()` is used, consider this example that creates a trivial table `test_table` and then enables column-level conflict resolution on it:

```
db=# CREATE TABLE my_app.test_table (id SERIAL PRIMARY KEY, val INT);
CREATE TABLE

db=# ALTER TABLE my_app.test_table REPLICA IDENTITY FULL;
ALTER TABLE

db=# SELECT bdr.alter_table_conflict_detection(
db(# my_app.test_table::regclass, column_modify_timestamp, cts);
alter_table_conflict_detection
- -----
t

db=# \d my_app.test_table
```

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, the new column specifies `NOT NULL` with a default value, which means that `ALTER TABLE ... ADD COLUMN` doesn't perform a table rewrite.

Note: We discourage using columns with the `bdr.column_timestamps` data type for other purposes as it can have negative effects. For example, it switches the table to column-level conflict resolution, which doesn't work correctly without the triggers.

Listing table with column-level conflict resolution

You can list tables having column-level conflict resolution enabled with the following query. This query 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. It's called within `column_timestamp_enable`.

Synopsis

```
bdr.column_timestamps_create(p_source cstring, p_timestamp timestampz)
```

Parameters

- `p_source` — The two options are `current` or `commit`.
- `p_timestamp` — Timestamp depends on the source chosen. If `commit`, then `TIMESTAMP_SOURCE_COMMIT`. If `current`, then `TIMESTAMP_SOURCE_CURRENT`.

2.10.2 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. This approach ensures we see only conflicts with timestamps in both tuples or in 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.

2.10.3 Current versus commit timestamp

An important decision is the timestamp to assign to modified columns.

By default, the timestamp assigned to modified columns is the current timestamp, as if obtained from `clock_timestamp`. This is simple, and for many cases it is perfectly correct (for example, when the conflicting rows modify non-overlapping subsets of columns).

It can, however, have various unexpected effects:

- The timestamp changes during statement execution, so if an UPDATE affects multiple rows, each gets a slightly different timestamp. This means that the effects of concurrent changes might 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 isn’t equivalent to the commit order, which means it likely can’t be serialized.

Note: We might add statement and transaction timestamps in the future, which would address issues with mixing effects of concurrent statements or transactions. Still, neither of these options can ever produce results equivalent to commit order.

It’s possible to also use the actual commit timestamp, although this feature is currently considered experimental. To use the commit timestamp, set the last parameter to `true` when enabling column-level conflict resolution:

```
SELECT bdr.column_timestamps_enable(test_table::regclass, cts, true);
```

You can disable it using `bdr.column_timestamps_disable`.

Commit timestamps currently have restrictions that are explained in [Notes](#).

2.10.4 Inspecting column timestamps

The column storing timestamps for modified columns is maintained automatically by triggers. Don’t modify it directly. It can be useful to inspect the current timestamps value, for example, while investigating how a conflict was resolved.

Three functions are useful 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 matters only 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)
```

2.10.5 Handling column conflicts using CRDT data types

By default, column-level conflict resolution picks the value with a higher timestamp and discards the other one. You can, however, reconcile the conflict in different, more elaborate ways. For example, you can use CRDT types that allow merging the conflicting values without discarding any information.

2.10.6 Notes

- The attributes modified by an `UPDATE` are determined by comparing the old and new row in a trigger. This means that if the attribute doesn't change a value, it isn't detected as modified even if it's explicitly set. For example, `UPDATE t SET a = a` doesn't mark `a` as modified for any row. Similarly, `UPDATE t SET a = 1` doesn't mark `a` as modified for rows that are already set to 1.
- For `INSERT` statements, we don't 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 weren't included in the `INSERT` statement and received default values. We can detect which attributes have a default value but can't know if it was included automatically or specified explicitly.

This effectively means column-level conflict resolution doesn't work for `INSERT-INSERT` conflicts even if the `INSERT` statements specify different subsets of columns. The newer row has timestamps that are all newer than the older row.

- By treating the columns independently, it's easy to violate constraints in a way that isn't 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);
```

Assume one node does:

```
UPDATE t SET a = 100;
```

Another node concurrently does:

```
UPDATE t SET b = 500;
```

Each of those updates is valid when executed on the initial row and so passes on each node. But when replicating to the other node, the resulting row violates the CHECK ($A > b$) constraint, and the replication stops until the issue is resolved manually.

- The column storing timestamp mapping is managed automatically. Don't specify or override the value in your queries, as it can 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 matters. So if you have custom triggers that modify tuples and are executed after the `pgl_clcd_` triggers, the modified columns aren't detected correctly.
- When using regular timestamps to order changes/commits, it's possible that the conflicting changes have exactly the same timestamp (because two or more nodes happened to generate the same timestamp). This risk isn't unique to column-level conflict resolution, as it can happen even for regular row-level conflict resolution. We use node id as a tie-breaker in this situation (the higher node id wins), which ensures that the same changes are applied on all nodes.
- It is possible that there is a clock skew between different nodes. While it can induce somewhat unexpected behavior (discarding seemingly newer changes because the timestamps are inverted), you can manage clock skew between nodes using the parameters `bdr.maximum_clock_skew` and `bdr.maximum_clock_skew_action`.

```
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 inside the BDR writer 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.

2.11.1 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.

2.11.2 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.

2.11.3 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.).

2.11.4 CRDT Types vs. Conflict Reporting

By default, detected conflicts are individually reported. 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.

2.11.5 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.

2.11.6 Implemented CRDT data types

Currently there are six CRDT data types implemented - grow-only counter and sum, positive-negative counter and sum, and delta counter and sum. The counters and sums behave mostly the same, except that the “counter” types are integer-based (`bigint`), while the “sum” types are decimal-based (`numeric`).

Additional CRDT types, described at [1], may be implemented later.

The currently implemented CRDT data types can be listed with the following query:

```
SELECT n.nspname, t.typname
FROM bdr.crdt_handlers c
JOIN (pg_type t JOIN pg_namespace n ON t.typnamespace = n.oid)
ON t.oid = c.crdt_type_id;
```

grow-only counter (crdt_gcounter)

- supports only increments with non-negative values (value + int and counter + bigint operators)
- current value of the counter can be obtained either using # operator or by casting it to bigint
- is not compatible with simple assignments like counter = value (which is common pattern when the new value is computed somewhere in the application)
- allows simple reset of the counter, using the ! operator (counter = !counter)
- internal state can be inspected using crdt_gcounter_to_text

```
CREATE TABLE crdt_test (
    id          INT PRIMARY KEY,
    cnt         bdr.crdt_gcounter NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);      -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129824); -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4531);  -- error: negative value

-- enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

-- increment counters
UPDATE crdt_test SET cnt = cnt + 1 WHERE id = 1;
UPDATE crdt_test SET cnt = cnt + 120 WHERE id = 2;

-- error: minus operator not defined
UPDATE crdt_test SET cnt = cnt - 1 WHERE id = 1;

-- error: increment has to be non-negative
UPDATE crdt_test SET cnt = cnt + (-1) WHERE id = 1;

-- reset counter
UPDATE crdt_test SET cnt = !cnt WHERE id = 1;

-- get current counter value
SELECT id, cnt::bigint, cnt FROM crdt_test;

-- show internal structure of counters
SELECT id, bdr.crdt_gcounter_to_text(cnt) FROM crdt_test;
```

grow-only sum (crdt_gsum)

- supports only increments with non-negative values (`sum + numeric`)
- current value of the sum can be obtained either by using `#` operator or by casting it to `numeric`
- is not compatible with simple assignments like `sum = value` (which is the common pattern when the new value is computed somewhere in the application)
- allows simple reset of the sum, using the `!` operator (`sum = !sum`)
- internal state can be inspected using `crdt_gsum_to_text`

```
CREATE TABLE crdt_test (
  id      INT PRIMARY KEY,
  gsum    bdr.crdt_gsum NOT NULL DEFAULT 0.0
);

INSERT INTO crdt_test VALUES (1, 0.0);      -- initialized to 0
INSERT INTO crdt_test VALUES (2, 1298.24); -- initialized to 1298.24
INSERT INTO crdt_test VALUES (3, -45.31);  -- error: negative value

-- enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

-- increment sum
UPDATE crdt_test SET gsum = gsum + 11.5 WHERE id = 1;
UPDATE crdt_test SET gsum = gsum + 120.33 WHERE id = 2;

-- error: minus operator not defined
UPDATE crdt_test SET gsum = gsum - 15.2 WHERE id = 1;

-- error: increment has to be non-negative
UPDATE crdt_test SET gsum = gsum + (-1.56) WHERE id = 1;

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

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

-- show internal structure of sums
SELECT id, bdr.crdt_gsum_to_text(gsum) FROM crdt_test;
```

positive-negative counter (crdt_pncounter)

- supports increments with both positive and negative values (through `counter + int` and `counter + bigint` operators)
- current value of the counter can be obtained either by using `#` operator or by casting to `bigint`
- is not compatible with simple assignments like `counter = value` (which is the common pattern when the new value is computed somewhere in the application)
- allows simple reset of the counter, using the `!` operator (`counter = !counter`)

- internal state can be inspected using `crdt_pncounter_to_text`

```
CREATE TABLE crdt_test (  
    id          INT PRIMARY KEY,  
    cnt         bdr.crdt_pncounter NOT NULL DEFAULT 0  
);  
  
INSERT INTO crdt_test VALUES (1, 0);      -- initialized to 0  
INSERT INTO crdt_test VALUES (2, 129824); -- initialized to 129824  
INSERT INTO crdt_test VALUES (3, -4531);  -- initialized to -4531  
  
-- enable CLCD on the table  
ALTER TABLE crdt_test REPLICA IDENTITY FULL;  
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);  
  
-- increment counters  
UPDATE crdt_test SET cnt = cnt + 1      WHERE id = 1;  
UPDATE crdt_test SET cnt = cnt + 120    WHERE id = 2;  
UPDATE crdt_test SET cnt = cnt + (-244) WHERE id = 3;  
  
-- decrement counters  
UPDATE crdt_test SET cnt = cnt - 73     WHERE id = 1;  
UPDATE crdt_test SET cnt = cnt - 19283 WHERE id = 2;  
UPDATE crdt_test SET cnt = cnt - (-12)  WHERE id = 3;  
  
-- get current counter value  
SELECT id, cnt::bigint, cnt FROM crdt_test;  
  
-- show internal structure of counters  
SELECT id, bdr.crdt_pncounter_to_text(cnt) FROM crdt_test;  
  
-- reset counter  
UPDATE crdt_test SET cnt = !cnt WHERE id = 1;  
  
-- get current counter value after the reset  
SELECT id, cnt::bigint, cnt FROM crdt_test;
```

positive-negative sum (`crdt_pnsum`)

- supports increments with both positive and negative values (through `sum + numeric`)
- current value of the sum can be obtained either by using `#` operator or by casting to `numeric`
- is not compatible with simple assignments like `sum = value` (which is the common pattern when the new value is computed somewhere in the application)
- allows simple reset of the sum, using the `!` operator (`sum = !sum`)
- internal state can be inspected using `crdt_pnsum_to_text`

```
CREATE TABLE crdt_test (  
    id          INT PRIMARY KEY,  
    pnsum       bdr.crdt_pnsum NOT NULL DEFAULT 0  
);
```

(continues on next page)

(continued from previous page)

```

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

```

(continues on next page)

(continued from previous page)

```

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 Transaction streaming

With logical replication, transactions are decoded concurrently on the publisher but are not sent to subscribers until the transaction is committed. If the changes exceed `logical_decoding_work_mem` (PostgreSQL 13 and later), they are spilled to disk. This means that particularly with large transactions, there will be some delay before they reach subscribers, and may entail additional I/O on the publisher.

Beginning with PostgreSQL 14, transactions can optionally be decoded and sent to subscribers *before* they are committed on the publisher. The subscribers save the incoming changes to a staging file (or set of files) and apply them when the transaction commits (or discard them if the transaction aborts). This makes it possible to apply transactions on on subscribers as soon as the transaction commits.

2.12.1 BDR enhancements

PostgreSQL's built-in transaction streaming, as described above, has the following limitations:

- while it's no longer necessary to spill changes to disk on the publisher, changes must be written to disk on each subscriber
- if the transaction aborts, the work (changes received by each subscriber and the associated storage I/O) will have been wasted

However, from version 3.7, BDR supports Parallel Apply, enabling multiple writer processes on each subscriber, which is leveraged to provide the following enhancements:

- decoded transactions can be streamed directly to a writer on the subscriber
- decoded transactions do not need to be stored on-disk on subscribers
- no need to wait for the transaction to commit before starting to apply the transaction on the subscriber

Caveats

- parallel apply must be enabled
- if the workload consists of many small and conflicting transactions, this can lead to frequent deadlocks between writers

Note: Direct streaming to writer is still an experimental feature and must be used with caution. For specifically, it may not work well with conflict resolutions since the commit timestamp of the streaming may not be available (as the transaction may not have yet committed on the origin).

2.12.2 Configuration

Transaction streaming is configured in two locations:

- at node level, via the GUC *Node configuration using `bdr.default_streaming_mode`*
- at group level, via the function *Group configuration using `bdr.alter_node_group_config()`*

Node configuration using `bdr.default_streaming_mode`

Permitted values are:

- `off`
- `writer`
- `file`
- `auto`

Default value is `auto`.

Note that to become effective, a change to this setting requires a restart of the pglogical receiver process for each subscription, which can be achieved with a server restart.

If `bdr.default_streaming_mode` is set any other value than `off`, the subscriber will request transaction streaming from the publisher. How this will be provided may also depend on the group configuration setting; see section XXX below for details.

Group configuration using `bdr.alter_node_group_config()`

The parameter `streaming_mode` in function *Group configuration using `bdr.alter_node_group_config()`* can be used to set the group transaction streaming configuration.

Permitted values are:

- `off`
- `writer`
- `file`
- `auto`
- `default`

Default value is `default`.

The value of the current setting is contained in the column `node_group_streaming_mode` from the view `:ref:bdr.node_group`<`bdr.node_group`>``. The value returned is a single char type and the possible values are ``D`` (`default`), ``W`` (`writer`), ``F`` (`file`), ``A`` (`auto`) and ``O`` (`off`).

Configuration setting effects

Transaction streaming is controlled at subscriber level by the GUC `bdr.default_streaming_mode`. Unless set to `off` (which disables transaction streaming), the subscriber will request transaction streaming.

If the publisher is capable of providing transaction streaming, it will stream transactions whenever the transaction size exceeds the threshold set in `logical_decoding_work_mem`. The publisher usually has no control over whether the transactions will be streamed to a file or to a writer. Except for some situations (such as `COPY`), it might hint the subscriber to stream the transaction to a writer (if possible).

The subscriber can stream transactions received from the publisher to either a writer or a file. The decision is based on several factors:

- if parallel apply is off (`num_writers = 1`), then it will be streamed to a file (writer 0 is always reserved for non-streamed transactions)
- if parallel apply is on, but all writers are already busy handling streamed transactions, then the new transaction will be streamed to a file. See `[bdr.writers](monitoring#monitoring-bdr-writers)` to check BDR writer status.

If streaming to a writer is possible (i.e. a free writer is available), then the decision whether to stream the transaction to a writer or a file is based upon the combination of group and node settings as per the following table:

Group	Node	Streamed to
off	(any)	(none)
(any)	off	(none)
writer	file	file
file	writer	file
default	writer	writer
default	file	file
default	auto	writer
auto	(any)	writer

Note that if the group configuration is set to `auto`, or the group configuration is `default` and the node configuration is `auto`, then the transaction is streamed to a writer if and only if the publisher has hinted that this should be done.

Currently the publisher will hint for the subscriber to stream to writer for the following transaction types, which are known to be conflict free and can be safely handled by the writer in any case:

- `COPY`
- `CREATE INDEX CONCURRENTLY`

2.12.3 Monitoring

Usage of transaction streaming can be monitored via the `:ref:`bdr.stat_subscription`bdr.stat\_subscription` function on the subscriber node.

- `nstream_writer` - number of transactions streamed to a writer
- `nstream_file` - number of transactions streamed to file
- `nstream_commit` - number of committed streamed transactions
- `nstream_abort` - number of aborted streamed transactions
- `nstream_start` - number of streamed transactions which were started
- `nstream_stop` - number of streamed transactions which were fully received

2.13 Durability & Performance Options

2.13.1 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 provides a *CAMO versus group commit* feature to guarantee durability and visibility by providing a variant of synchronous replication. This is very similar to Postgres' `synchronous_commit` feature for physical standbys, but providing a lot more flexibility for large scale distributed systems.

In addition to Group Commit, BDR also offers two additional modes (which cannot currently be combined with Group Commit):

- **Commit At Most Once (CAMO)**. This feature solves the problem with knowing whether your transaction has COMMITed (and replicated) or not in case of node or network failures 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 a different 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 check for conflicts prior to the commit. Every transaction is applied and prepared 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 *CAMO with Eager All-Node Replication* chapter.

Postgres itself provides **Physical Streaming Replication (PSR)**, which is uni-directional, but offers a *synchronous variant*. For backwards compatibility, BDR still supports configuring synchronous replication via `synchronous_commit` and `synchronous_standby_names`, see *Legacy Synchronous Replication using BDR*, but the use of

CAMO versus group commit is recommended instead in all cases.

This chapter covers the various forms of synchronous or eager replication and its timing aspects.

2.13.2 Terms and Definitions

This and subsequent chapters refer to BDR nodes taking different roles. These are implicitly assigned per transaction and are unrelated even for concurrent transactions.

- the *origin* is the node that receives the transaction from the client or application. It is the node processing the transaction first, initiating replication to other BDR nodes and responding back to the client with a confirmation or an error.
- a *partner* node is a BDR node expected to confirm transactions either according to Group Commit or CAMO requirements.
- a *commit group* is the group of all BDR nodes involved in the commit, i.e. the origin and all of its partner nodes, which may be just a few or all peer nodes.

2.13.3 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. The “Mode” column takes on different meaning depending on the variant. For PSR and Legacy Synchronous Replication with BDR, it refers to the `synchronous_commit` setting. For CAMO, it refers to the `bdr.enable_camo` setting. Lastly, for Group Commit, it refers to the confirmation requirements of the commit scope configuration.

Variant	Mode	Received	Visible	Durable
async BDR	off (default)	no	no	no
PSR	remote_write (2)	yes	no	no (1)
PSR	on (2)	yes	no	yes
PSR	remote_apply (2)	yes	yes	yes
Group Commit	‘ON received’ nodes	yes	no	no
Group Commit	‘ON replicated’ nodes	yes	no	no
Group Commit	‘ON durable’ nodes	yes	no	yes
Group Commit	‘ON visible’ nodes	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
legacy (3)	remote_write (2)	yes	no	no
legacy (3)	on (2)	yes	yes	yes
legacy (3)	remote_apply (2)	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.*
- (3) not recommended, please consider using Group Commit instead.*

Reception ensures the normally operating peer will be able to eventually apply the transaction without requiring any further communication, i.e. even in the face of a full or partial network outage. A crash of a peer node may still require a re-transmission of the transaction, as this confirmation does not involve persistent storage. All modes considered synchronous provide this protection.

Visibility implies the transaction was applied remotely and all other clients will see the results of the transaction on all nodes providing this guarantee immediately after the commit is confirmed by the origin node. Without visibility, other clients connected may not see the results of the transaction and experience stale reads.

Durability relates to the peer node’s storage and provides protection against loss of data after a crash and recovery of the peer node. This can either relate to the reception of the data (as with Physical Streaming Replication) or to visibility (as with Group Commit, CAMO and Eager). The former eliminates the need for retransmissions after a crash, while the latter ensures visibility is maintained across restarts.

2.13.4 Internal Timing of Operations

For a better understanding of how the different modes work, it is helpful to realize PSR and BDR 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 BDR, 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 Group Commit, CAMO, and Eager, note that the origin node waits for a certain number of confirmations 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	replication before or after commit
PSR	persist first	after WAL flush of commit record
BDR	apply first	after WAL flush of commit record
Group Commit	apply first	before COMMIT on origin
CAMO	apply first	before COMMIT on origin
Eager	apply first	before COMMIT on origin

2.13.5 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)	Group Commit	CAMO	Eager	PSR (1)
synchronous_standby_names	n/a	n/a	n/a	req
synchronous_commit	n/a	n/a	n/a	opt
synchronous_replication_availability	n/a	opt	n/a	opt
bdr.enable_camo	n/a	req	n/a	n/a
bdr.commit_scope	req	n/a	opt	n/a
bdr.global_commit_timeout	opt	opt	opt	n/a

- (1) values in this column apply also to `synchronous_commit` and `synchronous_standby_names` being used in combination with BDR.*

2.13.6 Planned Shutdown and Restarts

When using Group Commit with receive confirmations 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 the origin 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.

2.13.7 Legacy Synchronous Replication using BDR

Note: This approach is not recommended. Please consider using *CAMO versus group commit* instead.

Usage

To enable synchronous replication using BDR, 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 some 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 BDR applying the transaction before persisting it, the values `on` and `remote_apply` are equivalent (for logical replication).

Migration to Group Commit

The Group Commit feature of BDR is configured independent of `synchronous_commit` and `synchronous_standby_names`. Instead, the `bdr.commit_scope` GUC allows to select the scope per transaction. And instead of `synchronous_standby_names` configured on each node individually, Group Commit uses globally synchronized Commit Scopes.

Note: While the grammar for `synchronous_standby_names` and Commit Scopes looks very similar, it is important to note that the former does not account for the origin node, but the latter does. Therefore, for example `synchronous_standby_names = 'ANY 1 (..)'` is equivalent to a Commit Scope of `ANY 2 (...)`. This choice makes reasoning about majority easier and reflects that the origin node also contributes to the durability and visibility of the transaction.

2.14 Group Commit

The goal of Group Commit is to provide protection against data loss in case of single node failures or temporary outages. This is achieved by requiring more than one BDR node to successfully receive and confirm a transaction at COMMIT time.

2.14.1 Requirements

During normal operation, Group Commit is completely transparent to the application. Upon failover, the reconciliation phase needs to be explicitly triggered or consolidated by either the application or a proxy in between. HARP provides native support for Group Commit and will trigger the reconciliation phase, making this equally transparent to the client.

On the origin node, a transaction committed with Group Commit uses two-phase commit underneath. Therefore, `max_prepared_transactions` needs to be configured high enough to handle all such transactions originating per node.

2.14.2 Configuration

To use Group Commit, a Commit Scope first needs to be defined. This determines the BDR nodes involved in the commit of a transaction. Once a scope is established, a transaction may be configured to use Group Commit as follows:

```
BEGIN;  
SET LOCAL bdr.commit_scope = example_scope;  
...  
COMMIT;
```

To complete this example, the Commit Scope in question may previously have been defined as:

```
SELECT bdr.add_commit_scope(  
    commit_scope_name := example_scope,  
    origin_node_group := example_bdr_group,  
    rule := ANY 2 (example_bdr_group)  
);
```

This assumes a *Node Group* named `example_bdr_group` exists and includes at least two BDR nodes as members (either directly or in sub-groups). Any transaction committed within the `example_scope` would require one extra confirmation from a BDR node within the group. Together with the origin node, this accounts for “ANY 2” nodes out of the group, on which the transaction is guaranteed to be visible and durable after the commit.

Origin Groups

Rules for Commit Scopes may depend on which node the transaction is committed on, i.e. which node acts as the origin for the transaction. To make this transparent for the application, BDR allows a Commit Scope to define different rules depending on where the transaction originates from.

For example, consider a BDR cluster with nodes spread across two data centers, a left and a right one. Assume the top-level BDR node group is called `top_group`. The following commands may be used to setup sub-groups and create a Commit Scope requiring all nodes in the local data center to confirm the transaction, but only one node from the remote one.

```

- - create sub-groups
SELECT bdr.create_node_group(node_group_name := left_dc,
                             parent_group_name := top_group,
                             join_node_group := false);
SELECT bdr.create_node_group(node_group_name := right_dc,
                             parent_group_name := top_group,
                             join_node_group := false);

- - create a commit scope with individual rules for each sub-group
SELECT bdr.add_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := left_dc,
    rule := ALL (left_dc) AND ANY 1 (right_dc)
);
SELECT bdr.add_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := right_dc,
    rule := ANY 1 (left_dc) AND ALL (right_dc)
);

```

Confirmation Levels

BDR nodes can send confirmations for a transaction at different points in time, similar to *Commit At Most Once*. In increasing levels of protection, from the perspective of the confirming node, these are:

- **received** - a remote BDR node confirms the transaction immediately after having received it, prior to starting local application
- **replicated** - confirm after applying changes of the transaction, but before flushing them to disk
- **durable** - confirm the transaction after all of its changes have been flushed to disk
- **visible** (default) - confirm the transaction after all of its changes have been flushed to disk and it has been made visible to concurrent transactions.

In rules for Commit Scopes, these confirmation levels may be appended to the node group definition (in parenthesis) with ON as follows:

- ANY 1 (right_dc) ON remote_write
- ALL (left_dc) ON remote_commit_flush (default and may as well be omitted)
- ALL (left_dc) ON remote_commit_async AND ANY 1 (right_dc) ON remote_write

2.14.3 Reference

Commit Scope Grammar

For reference, the grammar for commit scopes is as follows:

```

commit_scope: confirmation
             | confirmation AND commit_scope

confirmation: node_def (ON [received|replicated|durable|visible])

```

(continues on next page)

(continued from previous page)

```
node_def:    ANY num (node_group [, ...])
             | MAJORITY (node_group [, ...])
             | ALL (node_group [, ...])
```

Adding a commit scope rule

The function `bdr.add_commit_scope` creates a rule for the given commit scope name and origin node group. If the rule is the same for all nodes in the BDR cluster, invoking this function once for the top-level node group is sufficient to fully define the commit scope.

Alternatively, it may be invoked multiple times with the same `commit_scope_name` but different origin node groups and rules for commit scopes that vary depending on the origin of the transaction.

Synopsis

```
bdr.add_commit_scope(commit_scope_name NAME, origin_node_group NAME,
                    rule TEXT)
```

Changing a commit scope rule

To change a specific rule for a single origin node group within a commit scope, the function `bdr.alter_commit_scope` may be used.

Synopsis

```
bdr.alter_commit_scope(commit_scope_name NAME, origin_node_group NAME,
                      rule TEXT)
```

Removing a commit scope rule

The `bdr.remove_commit_scope` can be used to drop a single rule within a commit scope. If multiple rules are defined for the commit scope, this function must be invoked once per rule to fully remove the entire commit scope.

Synopsis

```
bdr.remove_commit_scope(commit_scope_name NAME, origin_node_group NAME)
```

2.15 Eager Replication

To prevent conflicts after a commit, set the `bdr.commit_scope` parameter to `global`. The default setting of `local` disables eager replication, so BDR will apply changes and resolve potential conflicts post-commit, as described in the *Conflicts chapter*.

In this mode, BDR uses two-phase commit (2PC) internally to detect and resolve conflicts prior to the local commit. It turns a normal `COMMIT` of the application into an implicit two-phase commit, requiring all peer nodes to prepare the transaction, before the origin node continues to commit it. If at least one node is down or unreachable during the prepare phase, the commit will time out after `bdr.global_commit_timeout`, leading to an abort of the transaction. Note that there is no restriction on the use of temporary tables, as exists in explicit 2PC in PostgreSQL.

Once prepared, Eager All-Node Replication employs Raft to reach a commit decision. In case of failures, this allows a remaining majority of nodes to reach a congruent commit or abort decision so they can finish the transaction. This unblocks the objects and resources locked by the transaction and allows the cluster to proceed.

In case all nodes remain operational, the origin will confirm the commit to the client only after all nodes have committed, to ensure that the transaction is immediately visible on all nodes after the commit.

2.15.1 Requirements

Eager All-Node Replication uses prepared transactions internally; therefore all replica nodes need to have a `max_prepared_transactions` configured high enough to be able to handle all incoming transactions (possibly in addition to local two-phase commit and CAMO transactions; see *Configuration: Max Prepared Transactions*). We recommend to configure it the same on all nodes, and high enough to cover the maximum number of concurrent transactions across the cluster for which CAMO or Eager All-Node Replication is used. Other than that, no special configuration is required, and every BDR cluster can run Eager All-Node transactions.

2.15.2 Usage

To enable Eager All-Node Replication, the client needs to switch to global commit scope at session level, or for individual transactions as shown here:

```
BEGIN;

... other commands possible...

SET LOCAL bdr.commit_scope = global;

... other commands possible...
```

The client can continue to simply issue a `COMMIT` at the end of the transaction and let BDR manage the two phases:

```
COMMIT;
```

2.15.3 Error handling

Given that BDR manages the transaction, the client only needs to check the result of the `COMMIT` (as is advisable in any case, including single-node Postgres).

In case of an origin node failure, the remaining nodes will eventually (after at least `bdr.global_commit_timeout`) decide to rollback the globally prepared transaction. Raft prevents inconsistent commit vs. rollback decisions. This, however, requires a majority of connected nodes. Disconnected nodes keep the transactions prepared to be able to eventually commit them (or rollback) as needed to reconcile with the majority of nodes that may have decided and made further progress.

Eager All-Node Replication with CAMO

Eager All-Node Replication goes beyond CAMO and implies it; there is no need to additionally enable `bdr.enable_camo`, if `bdr.commit_scope` is set to `global`. Nor does a CAMO pair need to be configured via `bdr.add_camo_pair()`.

Any other active BDR node may be used in the role of a CAMO partner to query a transaction's status'. However, this non-CAMO usage needs to be indicated to the `bdr.logical_transaction_status` function with a third argument of `require_camo_partner = false`. Otherwise, it may complain about a missing CAMO configuration (which is not required for Eager transactions).

Other than this difference in configuration and invocation of that function, the client needs to adhere to the protocol described for

`pgbench` with CAMO/Failover options. See the reference client implementations.

Limitations

Transactions using Eager Replication cannot yet execute DDL, nor do they support explicit two-phase commit. These may be allowed in later releases. Note that the `TRUNCATE` command is allowed.

Replacing a crashed and unrecoverable BDR node with its physical standby is not currently supported in combination with Eager All Node transactions.

BDR currently offers a global commit scope only; later releases will support Eager Replication with fewer nodes for increased availability.

It is not possible for Eager All Node replication to be combined with `synchronous_replication_availability = 'async'`. Trying to configure both will yield an error.

The Decoding Worker feature is not currently supported in combination with Eager All Node transactions. Installations using Eager must keep `enable_wal_decoder` disabled for the BDR node group using Eager All Node transactions.

Synchronous replication uses a mechanism for transaction confirmation different from Eager. The two are not compatible and must not be used together. Therefore, whenever using Eager All Node transactions, please make sure none of the BDR nodes are configured in `synchronous_standby_names`. Using synchronous replication to a non-BDR node acting as a physical standby is well possible.

2.15.4 Effects of Eager Replication in General

Increased Commit Latency

Adding a synchronization step means additional communication between the nodes, resulting in additional latency at commit time. Eager All Node Replication adds roughly two network round trips (to the furthest peer node in the worst case). Logical standby nodes and nodes still in the process of joining or catching up are not included, but will eventually receive changes.

Before a peer node can confirm its local preparation of the transaction, it also needs to apply it locally. This further adds to the commit latency, depending on the size of the transaction. Note that this is independent of the `synchronous_commit` setting and applies whenever `bdr.commit_scope` is set to `global`.

Increased Abort Rate

Note: The performance of Eager Replication is currently known to be unexpectedly slow (around 10 TPS only). This is expected to be improved in the next release.

With single-node Postgres, or even with BDR in its default asynchronous replication mode, errors at COMMIT time are rare. The additional synchronization step adds a source of errors, so applications need to be prepared to properly handle such errors (usually by applying a retry loop).

The rate of aborts depends solely on the workload. Large transactions changing many rows are much more likely to conflict with other concurrent transactions.

2.16 Commit At Most Once

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 a COMMIT is submitted, the application might not receive a reply from the server and is therefore unsure whether the transaction committed.

The application can't easily decide between the two options of:

- Retrying the transaction with the same data, since this can in some cases cause the data to be entered twice
- 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.

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 depends on the application design and requires application-specific error-handling logic, so it isn't effective in all cases.

The CAMO feature in BDR offers a more general solution and doesn't require an INSERT. When activated by `bdr.enable_camo` or `bdr.commit_scope`, the application receives a message containing the transaction identifier, if already assigned. Otherwise, the first write statement in a transaction sends that information to the client. If the application sends an explicit COMMIT, the protocol ensures that the application receives the notification of the transaction identifier before the COMMIT is sent. If the server doesn't 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 to retry the transaction.

CAMO works in one of two modes:

- Pair mode
- With Eager All Node Replication

In 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. The node that receives the transactions from the application might be referred to as “origin” and the node that confirms these transactions as “partner.” However, there’s no difference in the CAMO configuration for the nodes in the CAMO pair. The pair is symmetric.

When combined with *CAMO 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 must be configured in this mode.

Warning: CAMO requires changes to the user’s application to take advantage of the advanced error handling. Enabling a parameter isn’t enough to gain protection. Reference client implementations are provided to customers upon request.

2.16.1 Requirements

To use CAMO, an application must issue an explicit COMMIT message a separate request (not as part of a multi-statement request). CAMO can’t provide status for transactions issued from procedures or from single-statement transactions that use implicit commits.

2.16.2 Configuration

Assume an existing BDR cluster consists of the nodes `node1` and `node2`. Both nodes are part of a BDR-enabled database called `bdrdemo`, and both part of the same node group `mygroup`. You can configure the nodes to be CAMO partners for each other.

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

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

1. Adjust the application to use the COMMIT error handling that CAMO suggests.

We don’t recommend enabling CAMO at the server level, as this imposes higher latency for all transactions, even when not needed. Instead, selectively enable it for individual transactions by turning on CAMO at the session or transaction level.

To enable at CAMO at the session level:

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

To enable CAMO for individual transactions, 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 between nodes with differences* 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 enough to enable the global commit scope after the start of the transaction. You don't need to set `bdr.enable_camo`.

```
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.

2.16.3 Failure scenarios

Different failure scenarios occur in different configurations.

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 be unrecoverable on the CAMO partner by itself. This isn't an issue as long as the CAMO origin node remains operational, as it redistributes 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, you can use `bdr.synchronous_commit = local` to enforce a flush prior to the pre-commit confirmation. This doesn't work in with either remote write or local mode and has a performance impact due to 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) detects whether its CAMO partner is ready. If not, it temporarily switches to local mode. When in local mode, a node commits transactions locally until switching back to CAMO mode.

This doesn't allow COMMIT status to be retrieved, but it does let you choose availability over consistency. This mode can tolerate a single-node failure. In case both nodes of a CAMO pair fail, they might 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 requires 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` isn't set for the CAMO pair, the origin node switches to local mode immediately.

You can configure the detection on the sending node using PostgreSQL settings controlling keep-alives and timeouts on the TCP connection to the CAMO partner. The `wal_sender_timeout` is the 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 might be lower than the `wal_sender_timeout`, which influences synchronous standbys as well, and for which a good compromise between responsiveness and stability must 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 reconnect at least every 30 seconds. After connectivity is reestablished, it might therefore take up to 30 seconds until the CAMO partner connects back to its origin node. Any lag that accumulated on the CAMO partner further delays the switch back to CAMO protected mode.

Unlike during normal CAMO operation, in local mode there's no additional commit overhead. This can be problematic, as it allows the node to continuously process more transactions than the CAMO pair can normally process. Even if the CAMO partner eventually reconnects and applies transactions, its lag only ever increases in such a situation, preventing reestablishing the CAMO protection. To artificially throttle transactional throughput, BDR provides the `bdr.camo_local_mode_delay` setting, which allows you to delay a COMMIT in local mode by an arbitrary amount of time. We recommend measuring commit times in normal CAMO mode during expected workloads and configuring this delay accordingly. The default is 5 ms, which reflects a local network and a relatively quick CAMO partner response.

Consider the choice of whether to allow local mode in view of the architecture and the availability requirements. The following examples provide some detail.

Example: Symmetric node pair

This example considers 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's therefore suitable for workload patterns where it is acceptable to write concurrently on more than one node, such as in cases that aren't likely to generate conflicts.

With local mode

If local mode is allowed, there's no single point of failure. 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 is unknown.

Without local mode

If local mode isn't allowed, then each node requires the other node for committing transactions, that is, each node is a single point of failure. 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 prevented until the node recovers.

2.16.4 Application use

Overview and requirements

CAMO 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 must retrieve a global identifier for the transaction, consisting of a node id and a transaction id (both 32-bit integers).
- If the current server fails while attempting a COMMIT of a transaction, the application must connect to its CAMO partner, retrieve the status of that transaction, and retry depending on the response.

The application must store the global transaction identifier only for the purpose of verifying the transaction status in case of disconnection during COMMIT. In particular, the application doesn't need an additional persistence layer. If the application fails, it needs only 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, that is, 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 you to toggle the `require_raft`. You can't currently change the nodes of a pairing. You must instead use `bdr.remove_camo_pair` followed by `bdr.add_camo_pair`.

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 by `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 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 can 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 triggers 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 (before `bdr.global_commit_timeout` expires).

Note: This function queries the past or current state. A positive return value doesn't indicate whether the CAMO partner can confirm future transactions.

Fetch the CAMO partner

This function shows the local node's CAMO partner (configured by 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 isn't 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

To check the status of a transaction that was being committed when the node failed, the application must use this function:

```
bdr.logical_transaction_status(node_id, xid, require_camo_partner)
```

With CAMO used in pair mode, use this function only on a node that's part of a CAMO pair. Along with Eager replication, you can use it on all nodes.

In both cases, you must call the function within 15 minutes after the commit was issued. The CAMO partner must regularly purge such meta-information and therefore can't provide correct answers for older transactions.

Before 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 were received but not yet applied.

Despite its name, it's not always a read-only operation. If the status is unknown, the CAMO partner decides whether to commit or abort the transaction, storing that decision locally to ensure consistency going forward.

The client must not call this function before attempting to commit on the origin. Otherwise the transaction might be forced to roll 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. 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 returns one of these results:

- **'committed'::TEXT** — The transaction was 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 was 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 wasn't committed or aborted yet. The transaction might 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. 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 aren't yet committed or aborted. This means that even transactions not yet replicated (or not even started on the origin node) might 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's 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.

2.16.5 Interaction with DDL and global locks

Transactions protected by CAMO can contain DDL operations. However, DDL uses global locks, which already provide some synchronization among nodes. See *DDL Locking Details* for more information.

Combining CAMO with DDL imposes a higher latency and 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.

Nontransactional DDL

The following DDL operations aren't allowed in a transaction block and therefore can't 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

2.16.6 CAMO limitations

- CAMO is designed to query the results of a recently failed COMMIT on the origin node, so in case of disconnection, code the application to immediately request the transaction status from the CAMO partner. Have as little delay as possible after the failure before requesting the status. Applications must 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's no easy way to recover it. Therefore, we recommend that applications wait for outstanding transactions to end before shutting down.
- For the client to apply proper checks, a transaction protected by CAMO can't be a single statement with implicit transaction control. You also can't use CAMO with a transaction-controlling procedure or in a DO block that tries to start or end transactions.
- CAMO resolves commit status but doesn't yet resolve pending notifications on commit. CAMO and Eager replication options don't allow the NOTIFY SQL command or the `pg_notify()` function. They also don't allow LISTEN or UNLISTEN .
- 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.

2.16.7 Performance implications

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

The CAMO partner confirming transactions must store transaction states. Compared to non-CAMO operation, this might require an additional seek for each transaction applied from the origin.

2.16.8 Client application testing

Proper use of CAMO on the client side isn't trivial. We strongly recommend testing the application behavior with the BDR cluster against failure scenarios such as node crashes or network outages.

2.16.9 CAMO versus group commit

CAMO doesn't currently work with *CAMO versus group commit*.

2.17 Lag Control

Data throughput of database applications on a BDR origin node can exceed the rate at which committed data can be safely replicated to downstream peer nodes. If this disparity persists beyond a period of time, or chronically, in high availability applications then organizational objectives related to disaster recovery or business continuity plans may not be satisfied.

The Replication Lag Control (RLC) feature is designed to regulate this imbalance using a dynamic rate limiting device so that data flow between BDR group nodes complies with these organizational objectives. It does so by controlling the extent of replication lag between BDR nodes.

Some of these objectives include the following:

- 1) Recovery Point Objective (RPC) specifies the maximum tolerated amount of data that can be lost due to unplanned events usually expressed as an amount of time. In non-replicated systems, RPC is used to set backup intervals to limit the risk of lost committed data due to failure. For replicated systems, RPC determines the acceptable amount of committed data that has not been safely applied to one or more peer nodes.
- 2) Resource Constraint Objective (RCO) acknowledges that there are finite storage constraints. This storage includes database files, WAL and temporary or intermediate files necessary for continued operation. For replicated systems, as lag increases the demands on these storage resources also increase.
- 3) Group Elasticity Objective (GEO) insures that any node is not originating new data at a clip that cannot be acceptably saved to its peer nodes. When that is the case then the detection of that condition may be used as one metric in the decision to expand the number of database nodes. Similarly, when that condition abates then it may influence the decision to contract the number of database nodes.

Lag Control manages replication lag by controlling the rate at which client connections may commit READ WRITE transactions. Replication lag is measured either as lag time or lag size depending on the objectives to be met. Transaction commit rate is regulated using a configured BDR commit delay time.

2.17.1 Requirements

To get started using Lag Control take the following steps:

- 1) Determine the maximum acceptable commit delay time `bdr.lag_control_max_commit_delay` that can be tolerated for all database applications.
- 2) Decide which lag measure is to be used. Either lag size `bdr.lag_control_max_lag_size` or lag time `bdr.lag_control_max_lag_time` must be chosen.
- 3) Decide on the number of BDR nodes in the group `bdr.lag_control_min_conforming_nodes` that must satisfy the lag measure chosen in step 2) as the minimal acceptable number of nodes.

2.17.2 Configuration

Lag Control is configured on each BDR node in the group using `postgresql.conf` configuration parameters. For Lag Control to be enabled, `bdr.lag_control_max_commit_delay` and either `bdr.lag_control_max_lag_size` or `bdr.lag_control_max_lag_time` must be set to positive non-zero values.

`bdr.lag_control_max_commit_delay` allows, and encourages, a specification of milliseconds with a fractional part including a sub-millisecond setting if appropriate.

By default, `bdr.lag_control_min_conforming_nodes` is set to one (1). For a complete list, see [Lag control](#)

2.17.3 Overview

Lag Control is a dynamic TPS rate limiting mechanism that operates at the client connection level. It is designed to be as unobtrusive as possible while satisfying configured Lag Control constraints. This means that if enough BDR nodes can replicate changes fast enough to remain below configured lag measure thresholds then the BDR runtime commit delay will stay fixed at 0 milliseconds.

If this is not the case, the BDR runtime commit delay will be minimally adjusted as high as necessary, but no higher, until the number of conforming nodes is returned to the minimum threshold.

Even after the minimum node threshold is reached, Lag Control continues to attempt to drive the BDR runtime commit delay back to zero. The BDR commit delay may rise and fall around an equilibrium level most of the time, but if data throughput or lag apply rates improve then the commit delay will decrease over time.

The BDR commit delay is a post commit delay. It occurs after the transaction has committed and after all Postgres resources locked or acquired by the transaction have been released. Therefore, the delay does not prevent concurrent active transactions from observing or modifying its values nor acquiring its resources. The same guarantee cannot be made for external resources managed by Postgres extensions. Notwithstanding extension dependencies, the same guarantee can be made if the BDR extension is listed before extension-based resource managers in `postgresql.conf`.

Strictly speaking, the BDR commit delay is not a per transaction delay. It is the mean value of commit delays over a stream of transactions for a particular client connection. This technique allows the commit delay and fine-grained adjustments of the value to escape the coarse granularity of OS schedulers, clock interrupts and variation due to system load. It also allows the BDR runtime commit delay to settle within microseconds of the lowest duration possible to maintain a lag measure threshold.

Finally, the BDR commit delay should not be conflated with the Postgres commit delay; they are unrelated, perform different functions and should not be substituted for each other.

2.17.4 Transaction Application

The BDR commit delay is applied to all READ WRITE transactions that modify data for user applications. This implies that any transaction that does not modify data, including declared READ WRITE transactions, is exempt from the commit delay.

Asynchronous transaction commit also executes a BDR commit delay. This may appear counterintuitive, but asynchronous commit, by virtue of its performance, can be one of the greatest sources of replication lag.

Postgres and BDR auxiliary processes do not delay at transaction commit. Most notably, BDR writers do not execute a commit delay when applying remote transactions on the local node. This is by design as BDR writers contribute nothing to outgoing replication lag and can reduce incoming replication lag the most by not having their transaction commits throttled by a delay.

2.17.5 Limitations

The maximum commit delay `bdr.lag_control_max_commit_delay` is a ceiling value representing a hard limit. This means that under no circumstance does a commit delay ever exceed the configured value. Conversely, the maximum lag measures `bdr.lag_control_max_lag_size` and `bdr.lag_control_max_lag_time` are soft limits that can be exceeded. When the maximum commit delay is reached, there exists no additional back-pressure on the lag measures to prevent their continued increase.

There is no way to exempt origin transactions that do not modify BDR replication sets from the commit delay. For these transactions, it may be useful to SET LOCAL the maximum transaction delay to zero (0).

2.17.6 Caveats

Application TPS is one of many factors that can affect replication lag. Other factors include the average size of transactions for which BDR commit delay can be less efficacious. In particular, bulk load operations can cause replication lag to rise, which can trigger a concomitant rise in the BDR runtime commit delay beyond the level reasonably expected by normal applications although still under the maximum allowed delay.

Similarly, an application with a very high OLTP requirement and modest data changes could be unduly restrained by the acceptable BDR commit delay setting.

In these cases, it may be useful to use SET [SESSION|LOCAL] command to custom configure Lag Control settings for those applications or modify those applications. For example, bulk load operations are sometimes split into multiple, smaller transactions to limit transaction snapshot duration and WAL retention size or establish a restart point if the bulk load fails. In a bow to Lag Control, those transaction commits can also schedule very long BDR commit delays to allow digestion of the lag contributed by the prior partial bulk load.

2.17.7 Meeting Organizational Objectives

In the example objectives list earlier:

- RPO can be met by setting an appropriate maximum lag time;
- RCO can be met by setting an appropriate maximum lag size;
- GEO can be met by monitoring the BDR runtime commit delay and the BDR runtime lag measures.

As mentioned, when the maximum BDR runtime commit delay is pegged at the BDR configured commit delay limit and the lag measures consistency exceed their BDR configured maximum levels, it can be a marker for BDR group expansion.

2.18 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.

2.18.1 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 features that allow a partition to be attached or detached/dropped without locking the rest of the table (when the underlying Postgres version supports it).

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 `snowflakeid`, `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, 10000000000,
  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.

2.19 Timestamp-Based Snapshots

The Timestamp-Based Snapshots allow reading data in a consistent manner via a user-specified timestamp rather than the usual MVCC snapshot. This can be used to access data on different BDR nodes at a common point-in-time; for example, as a way to compare data on multiple nodes for data quality checking. At this time, this feature does not work with write transactions.

The use of timestamp-based snapshots are enabled via the `snapshot_timestamp` parameter; this accepts either a timestamp value or a special value, 'current', which represents the current timestamp (now). If `snapshot_timestamp` is set, queries will use that timestamp to determine visibility of rows, rather than the usual MVCC semantics.

For example, the following query will return state of the `customers` table at 2018-12-08 02:28:30 GMT:

```
SET snapshot_timestamp = 2018-12-08 02:28:30 GMT;
SELECT count(*) FROM customers;
```

Without BDR, this only works with future timestamps or the above mentioned special 'current' value, so it cannot be used for historical queries.

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.

2.20 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.

2.20.1 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.

2.20.2 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.

2.20.3 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 ...
```

2.20.4 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

DML and 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.

2.20.5 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).

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,
                             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.

2.20.6 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*, 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)

2.20.7 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.

2.21 Stream Triggers

BDR introduces new types of triggers which can be used for additional data processing on the downstream/target node.

- Conflict Triggers
- Transform Triggers

Together, these types of triggers are known as Stream Triggers.

Stream Triggers are designed to be trigger-like in syntax, they leverage the PostgreSQL BEFORE trigger architecture, and are likely to have similar performance characteristics as PostgreSQL BEFORE Triggers.

One trigger function can be used by multiple trigger definitions, just as with normal PostgreSQL triggers. A trigger function is simply a program defined in this form: `CREATE FUNCTION ... RETURNS TRIGGER`. Creating the actual trigger does not require use of the `CREATE TRIGGER` command. Instead, stream triggers are created using the special BDR functions `bdr.create_conflict_trigger()` and `bdr.create_transform_trigger()`.

Once created, the trigger will be visible in the catalog table `pg_trigger`. The stream triggers will be marked as `tgisinternal = true` and `tgenabled = 'D'` and will have name suffix `'_bdrc'` or `'_bdr'`. 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.

2.21.1 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.

2.21.2 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.

2.21.3 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.

2.21.4 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.

2.21.5 Stream Triggers Variables

Both Conflict Trigger and Transform Triggers have access to information about rows and metadata via the predefined variables provided by trigger API and additional information functions provided by BDR.

In PL/pgSQL, the following predefined variables exist:

TG_NAME

Data type name; variable that contains the name of the trigger actually fired. Note that the actual trigger name has a `'_bdr'` 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.

2.21.6 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 the 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 the 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 *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()
```

2.21.7 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.

2.21.8 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

2.21.9 Stream Triggers Manipulation Interfaces

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` .

2.21.10 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 timestamptz;
BEGIN
    SOURCE_NEW := bdr.trigger_get_row(SOURCE_NEW);
    SOURCE_OLD := bdr.trigger_get_row(SOURCE_OLD);
    COMMITTS := bdr.trigger_get_committs(SOURCE_NEW);

    IF (TG_OP = INSERT) THEN
        INSERT INTO log SELECT I, COMMITTS, row_to_json(SOURCE_NEW);
    ELSIF (TG_OP = UPDATE) THEN
        INSERT INTO log SELECT U, COMMITTS, row_to_json(SOURCE_NEW);
    ELSIF (TG_OP = DELETE) THEN
        INSERT INTO log SELECT D, COMMITTS, row_to_json(SOURCE_OLD);
    END IF;

    RETURN NULL; -- do not apply the change
END;
$$;

```

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()

```

(continues on next page)

(continued from previous page)

```

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;
$$;

```

2.22 Explicit Two-Phase Commit (2PC)

An application may opt to use two-phase commit explicitly with BDR. See [Distributed Transaction Processing: The XA Specification](#).

The X/Open Distributed Transaction Processing (DTP) model envisages three software components:

- An application program (AP) that defines transaction boundaries and specifies actions that constitute a transaction.
- Resource managers (RMs, such as databases or file access systems) that provide access to shared resources.
- A separate component called a transaction manager (TM) that assigns identifiers to transactions, monitors their progress, and takes responsibility for transaction completion and for failure recovery.

BDR supports explicit external 2PC using the PREPARE TRANSACTION and COMMIT PREPARED/ROLLBACK PREPARED commands. Externally, a BDR cluster appears to be a single Resource Manager to the Transaction Manager for a single session.

When `bdr.commit_scope` is `local`, the transaction is prepared only on the local node. Once committed, changes will be replicated, and BDR then applies post-commit conflict resolution.

Using `bdr.commit_scope` set to `local` may seem nonsensical with explicit two-phase commit, but the option is offered to allow the user to control the trade-off between transaction latency and robustness.

Explicit two-phase commit does not work in combination with either CAMO or the global commit scope. Future releases may enable this combination.

2.22.1 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;
```

2.23 Catalogs and views

Catalogs and views are presented here in alphabetical order.

2.23.1 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 can see all conflicts for the tables they own, restricted by row-level security.

For details, see [Logging conflicts to a table](#).

bdr.conflict_history columns

Name	Type	Description
sub_id	oid	Which subscription produced this conflict; can be joined to <i>bdr.subscription</i> table
local_xid	xid	Local transaction of the replication process at the time of conflict
local_lsn	pg_lsn	Local transaction of the replication process at the time of conflict
local_time	timestamp with time zone	Local time of the conflict
remote_xid	xid	Transaction that produced the conflicting change on the remote node (an origin)
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 that produced the conflicting change on the remote node (an origin)
conflict_type	text	Detected type of the conflict
conflict_resolution	text	Conflict resolution chosen
conflict_index	regclass	Conflicting index (valid only if the index wasn't dropped since)
relويد	oid	Conflicting relation (valid only 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 (doesn't 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 that produced the local conflicting row (if <i>local_tuple</i> is set and the row isn't frozen)
local_tuple_node_id	oid	Node that produced the local conflicting row (if <i>local_tuple</i> is set and the row isn't frozen)
local_tuple_commit_time	timestamp with time zone	Last known change timestamp of the local conflicting row (if <i>local_tuple</i> is set and the row isn't frozen)

bdr.conflict_history_summary

A view containing user-readable details on row conflict.

bdr.conflict_history_summary columns

Name	Type	Description
nspname	text	Name of the schema
relname	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	timestampz	Last write timestamp
kv_ttl	int	Time to live for the value in milliseconds
kv_expire_ts	timestampz	Expiration timestamp (<i>kv_create_ts</i> + <i>kv_ttl</i>)

bdr.camo_decision_journal

A persistent journal of decisions resolved by a CAMO partner node after a failover, in case `bdr.logical_transaction_status` was invoked. Unlike `bdr.node_pre_commit`, this doesn't cover transactions processed under normal operational conditions (i.e., both nodes of a CAMO pair are running and connected). Entries in this journal aren't ever cleaned up automatically. This is a diagnostic tool that the system doesn't depend on.

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	timestampz	Decision time

bdr.crdt_handlers

This table lists merge (“handlers”) functions for all CRDT data types.

bdr.crdt_handlers Columns

Name	Type	Description
crdt_type_id	regtype	CRDT data type ID
crdt_merge_id	regproc	Merge function for this data type

bdr.ddl_replication

This view lists DDL replication configuration as set up by current DDL filters .

bdr.ddl_replication columns

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

bdr.depend

This table tracks internal object dependencies inside BDR catalogs.

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 [:ref:`bdr.global\\_consensus\\_journal\\_details` <bdr.global_consensus_journal_details>](#) view for more details.

bdr.global_consensus_journal columns

Name	Type	Description
log_index	int8	ID of the journal entry
term	int8	Raft term
origin	oid	ID of node where the request originated
req_id	int8	ID for the request
req_payload	bytea	Payload for the request
trace_context	bytea	Trace context for the request

bdr.global_consensus_journal_details

This view presents Raft messages that were sent and the corresponding responses, using the `bdr.decode_message_payload()` function to decode their payloads.

bdr.global_consensus_journal_details columns

Name	Type	Description
log_index	int8	ID of the journal entry
term	int8	Raft term
request_id	int8	ID of the request
origin_id	oid	ID of the node where the request originated
req_payload	bytea	Payload of the request
origin_node_name	name	Name of the node where the request originated
message_type_no	oid	ID of the BDR message type for the request
message_type	text	Name of the BDR message type for the request
message_payload	text	BDR message payload for the request
re-sponse_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 `:ref:`bdr.global_consensus_journal_details` <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, the `:ref:bdr.global_locks`<`bdr.global_locks`>`` view is preferable because the visible rows in ``bdr.global_lock` don't necessarily reflect all global locking activity.

Don't modify the contents of this table. It is an important BDR catalog.

`bdr.global_lock` columns

Name	Type	Description
<code>ddl_epoch</code>	<code>int8</code>	DDL epoch for the lock
<code>origin_node_id</code>	<code>oid</code>	OID of the node where the global lock has originated
<code>lock_type</code>	<code>oid</code>	Type of the lock (DDL or DML)
<code>nspname</code>	<code>name</code>	Schema name for the locked relation
<code>relname</code>	<code>name</code>	Relation name for the locked relation
<code>groupid</code>	<code>oid</code>	OID of the top level group (for Advisory locks)
<code>key1</code>	<code>integer</code>	First 32-bit key or lower order 32-bits of 64-bit key (for advisory locks)
<code>key2</code>	<code>integer</code>	Second 32-bit key or higher order 32-bits of 64-bit key (for advisory locks)
<code>key_is_bigint</code>	<code>boolean</code>	True if 64-bit integer key is used (for advisory locks)

`bdr.global_locks`

A view containing active global locks on this node. The `:ref:`bdr.global_locks`<bdr.global_locks>`` view exposes BDR's shared-memory lock state tracking, giving administrators 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
<i><code>origin_node_id</code></i>	<code>oid</code>	The OID of the node where the global lock has originated
<i><code>origin_node_name</code></i>	<code>name</code>	Name of the node where the global lock has originated
<i><code>lock_type</code></i>	<code>text</code>	Type of the lock (DDL or DML)
<i><code>relation</code></i>	<code>text</code>	Locked relation name (for DML locks) or keys (for advisory locks)
<i><code>pid</code></i>	<code>int4</code>	PID of the process holding the lock
<i><code>acquire_stage</code></i>	<code>text</code>	Internal state of the lock acquisition process
<i><code>waiters</code></i>	<code>int4</code>	List of backends waiting for the same global lock
<i><code>global_lock_request_time</code></i>	<code>timestampz</code>	Time this global lock acquire was initiated by origin node

continues on next page

Table 24 – continued from previous page

Name	Type	Description
<i>local_lock_request_time</i>	timestampz	Time the local node started trying to acquire the local lock
<i>last_state_change_time</i>	timestampz	Time <i>acquire_stage</i> 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. Can be null if unknown on the current node. This time is stamped at the 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 doesn't include any time taken on DDL epochs or queue flushing. If the lock is reacquired after local node restart, it becomes 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
<i>log_index</i>	int8	ID of the journal entry
<i>log_term</i>	int8	Raft term
<i>snapshot</i>	bytea	Raft snapshot data

bdr.local_consensus_state

This catalog table stores the current state of Raft on the local node.

bdr.local_consensus_state columns

Name	Type	Description
node_id	oid	ID of the node
current_term	int8	Raft term
apply_index	int8	Raft apply index
voted_for	oid	Vote cast by this node in this term
last_known_leader	oid	node_id of last known Raft leader

bdr.local_node

This table identifies the local node in the current database of the current Postgres instance.

bdr.local_node columns

Name	Type	Description
node_id	oid	ID of the node
pub_repsets	text[]	Published replication sets
sub_repsets	text[]	Subscribed replication sets

bdr.local_node_summary

A view containing the same information as :ref:`bdr.node\_summary` <bdr.node_summary>` but only for the local node.

bdr.local_sync_status

Information about status of either subscription or table synchronization process.

bdr.local_sync_status columns

Name	Type	Description
sync_kind	char	The kind of synchronization done
sync_subid	oid	ID of subscription doing the synchronization
sync_nsname	name	Schema name of the synchronized table (if any)
sync_relname	name	Name of the synchronized table (if any)
sync_status	char	Current state of the synchronization
sync_remote_relid	oid	ID of the synchronized table (if any) on the upstream

continues on next page

Table 28 – continued from previous page

Name	Type	Description
sync_end_lsn	pg_lsn	Position at which the synchronization state last changed

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
node_group_name	name	Name of the BDR group
node_region1	text	Node region name, from bdr.node_location
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_name	name	Name of the node
node_group_id	oid	ID of the node group
source_node_id	oid	ID of the source node
synchronize_structure	“char”	Schema synchronization done during the join
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
node_dsn	char	Connection string for the node
proto_version_ranges	int[]	Supported protocol version ranges by the node

bdr.node_catchup_info

This catalog table records relevant catchup 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`) by forwarding. The records in this table persists until the node(`node_id`) is a member of the BDR cluster.

bdr.node_conflict_resolvers

Currently configured conflict resolution for all known conflict types.

bdr.node_conflict_resolvers columns

Name	Type	Description
conflict_type	text	Type of the conflict
conflict_resolver	text	Resolver used for this conflict type

bdr.node_group

This catalog table lists all the BDR node groups.

bdr.node_group columns

Name	Type	Description
node_group_id	oid	ID of the node group
node_group_name	name	Name of the node group
node_group_default_repset	oid	Default replication set for this node group
node_group_default_repset_text	text	Default replication set for this node group
node_group_parent_id	oid	ID of parent group (0 if this is a root group)
node_group_flags	int	The group flags
node_group_uuid	uuid	The uuid of the group

continues on next page

Table 33 – continued from previous page

Name	Type	Description
node_group_apply_delay	interval	How long a subscriber waits before applying changes from the provider
node_group_check_constraints	bool	Whether the apply process checks constraints when applying data
node_group_num_writers	int	Number of writers to use for subscriptions backing this node group
node_group_enable_wal_decoder	bool	Whether the group has enable_wal_decoder set
node_group_streaming_mode	char	Transaction streaming setting: 'O' - off, 'F' - file, 'W' - writer, 'A' - auto, 'D' - default

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
node_group_name	name	Name of the BDR group
def_repset	name	Name of the default repset
def_repset_ops	text[]	Actions replicated by the default repset
def_repset_ext	name	Name of the default “external” repset (usually same as def_repset)
def_repset_ext_ops	text[]	Actions replicated by the default “external” repset (usually same as def_repset_ops)

bdr.node_local_info

A catalog table used to store per-node configuration that’s specific to the local node (as opposed to global view of per-node configuration).

bdr.node_local_info columns

Name	Type	Description
node_id	oid	The OID of the node (including the local node)
applied_state	oid	Internal ID of the node state
ddl_epoch	int8	Last epoch number processed by the node
slot_name	name	Name of the slot used to connect to that node (NULL 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
node_group_name	name	Name of the BDR group
node_id	oid	ID of the node
node_region	text	User-supplied region name
node_location	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
log_name	Name of the logging configuration
log_to_file	Whether it logs to the server log file
log_to_table	Whether it logs to a table, and which table is the target
log_conflict_type	Which conflict types it logs, if NULL means all
log_conflict_res	Which conflict resolutions it logs, if NULL means all

bdr.node_peer_progress

Catalog used to keep track of every node's progress in the replication stream. Every node in the cluster regularly broadcasts its progress every `bdr.replay_progress_frequency` milliseconds to all other nodes (default is 60000 ms, i.e., 1 minute). Expect $N * (N-1)$ rows in this relation.

You might be more interested in the `bdr.node_slots` view for monitoring purposes. See also [Monitoring and logging](#).

bdr.node_peer_progress columns

Name	Type	Description
node_id	oid	The OID of the originating node that reported this position info
peer_node_id	oid	The OID of the node's peer (remote node) for which this position info was reported
last_update_sent_time	timestampz	The time at which the report was sent by the originating node

continues on next page

Table 38 – continued from previous page

Name	Type	Description
last_update_recv_time	timestampz	The time at which the report was received by the local server
last_update_node_lsn	pg_lsn	LSN on the originating node at the time of the report
peer_position	pg_lsn	Latest LSN of the node's peer seen by the originating node
peer_replay_time	timestampz	Latest replay time of peer seen by the reporting node
last_update_horizon_xid	oid	Internal resolution horizon: all lower xids are known resolved on the reporting node
last_update_horizon_lsn	pg_lsn	Internal resolution horizon: same in terms of an LSN of the reporting node

bdr.node_pre_commit

Used internally on a node configured as a Commit At Most Once (CAMO) partner. Shows the decisions a CAMO partner took on transactions in the last 15 minutes.

bdr.node_pre_commit columns

Name	Type	Description
origin_node_id	oid	OID of the node where the transaction executed
origin_xid	oid	Transaction ID on the remote origin node
decision	char	'c' for commit, 'a' for abort
local_xid	xid	Transaction ID on the local node
commit_ts	timestampz	Commit timestamp of the transaction
decision_ts	timestampz	Decision time

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

continues on next page

Table 40 – continued from previous page

Column	Type	Description
catchup_interval	interval	Approximate time required for the peer node to catch up to all the changes that are yet to be applied

Note: The *replay_lag* is set immediately to zero after reconnect. As a workaround, use *replay_lag_bytes* , *replay_lag_size* , or *catchup_interval* .

bdr.node_slots

This view contains information about replication slots used in the current database by BDR.

See Monitoring outgoing replication for guidance on the use and interpretation of this view's fields.

bdr.node_slots columns

Name	Type	Description
target_dbname	name	Database name on the target node
node_group_name	name	Name of the BDR group
node_group_id	oid	The OID of the BDR group
origin_name	name	Name of the origin node
target_name	name	Name of the target node
origin_id	oid	The OID of the origin node
target_id	oid	The OID of the target node
local_slot_name	name	Name of the replication slot according to BDR
slot_name	name	Name of the slot according to Postgres (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#Group Replication Slot))
is_decoder_slot	boolean	Is this slot used by Decoding Worker
plugin	name	Logical decoding plugin using this slot (should be pg-logical_output or bdr)
slot_type	text	Type of the slot (should be logical)
datoid	oid	The OID of the current database
database	name	Name of the current database
temporary	bool	Is the slot temporary
active	bool	Is the slot active (does it have a connection attached to it)
active_pid	int4	The PID of the process attached to the slot
xmin	xid	The XID needed by the slot
catalog_xmin	xid	The catalog XID needed by the slot
restart_lsn	pg_lsn	LSN at which the slot can restart decoding
confirmed_flush_lsn	pg_lsn	Latest confirmed replicated position
usesysid	oid	sysid of the user the replication session is running as
username	name	username of the user the replication session is running as
application_name	text	Application name of the client connection (used by <i>synchronous_standby_names</i>)

continues on next page

Table 41 – continued from previous page

Name	Type	Description
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. As a workaround, use *replay_lag_bytes* or *replay_lag_size*.

bdr.node_summary

This view contains summary information about all BDR nodes known to the local node.

bdr.node_summary columns

Name	Type	Description
node_name	name	Name of the node
node_group_name	name	Name of the BDR group the node is part of
interface_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 that the node is trying to reach (during join or promotion)

continues on next page

Table 42 – continued from previous page

Name	Type	Description
node_seq_id	int4	Sequence identifier of the node used for generating unique sequence numbers
node_local_dbname	name	Database name of the node
set_repl_ops	text	Which operations does the default replication set replicate
node_id	oid	The OID of the node
node_group_id	oid	The OID of the BDR node group

bdr.queue

This table stores the historical record of replicated DDL statements.

bdr.queue columns

Name	Type	Description
queued_at	timestampz	When was the statement queued
role	name	Which role has executed the statement
replication_sets	text[]	Which replication sets was the statement published to
message_type	char	Type of a message
message	json	Payload of the message needed for replication of the statement

bdr.replication_set

A table that stores replication set configuration. For user queries, we recommend instead checking the `bdr.replication_sets` view.

bdr.replication_set columns

Name	Type	Description
set_id	oid	The OID of the replication set
set_nodeid	oid	OID of the node (always local node oid currently)
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_isinternal	boolean	Reserved
set_autoadd_tables	boolean	Indicates if new tables are automatically added to this replication set
set_autoadd_seqs	boolean	Indicates if new sequences are automatically added to this replication set

bdr.replication_set_table

A table that stores replication set table membership. For user queries, we recommend instead checking the `bdr.tables` view.

bdr.replication_set_table columns

Name	Type	Description
set_id	oid	The OID of the replication set
set_reloid	regclass	Local ID of the table
set_att_list	text[]	Reserved
set_row_filter	pg_node_tree	Compiled row filtering expression

bdr.replication_set_ddl

A table that stores replication set ddl replication filters. For user queries, we recommend instead checking the `bdr.ddl_replication` view.

bdr.replication_set_ddl Columns

Name	Type	Description
set_id	oid	The OID of the replication set
set_ddl_name	name	Name of the DDL filter
set_ddl_tag	text	Command tag for the DDL filter
set_ddl_role	text	Role executing the DDL

bdr.replication_sets

A view showing replication sets defined in the BDR group, even if they aren't 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 are automatically added to this replication set
set_autoadd_seqs	boolean	Indicates if new sequences are automatically added to this replication set

bdr.schema_changes

A simple view to show all the changes to schemas in 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_addrname	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 in 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_addrname	text[]	Location of schema change

bdr.sequence_alloc

A view to see the sequences allocated.

bdr.sequence_alloc columns

Name	Type	Description
seqid	regclass	The ID of the sequence
seq_chunk_size	bigint	A sequence number for the chunk within its value
seq_allocated_up_to	bigint	
seq_nallocs	bigint	
seq_last_alloc	timestampz	Last sequence allocated

bdr.sequences

This view lists all sequences with their kind, excluding sequences for internal BDR bookkeeping.

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', 'gallocc')

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. Contains data only 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

continues on next page

Table 53 – continued from previous page

Column	Type	Description
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 <i>track_io_timing</i> is enabled, otherwise zero)
blk_write_time	double precision	Total time spent writing blocks for the relation, in milliseconds (if <i>track_io_timing</i> is enabled, otherwise zero)
lock_acquire_time	double precision	Total time spent acquiring locks on the relation, in milliseconds (if <i>bdr.track_apply_lock_timing</i> is enabled, otherwise zero)

bdr.stat_subscription

Apply statistics for each subscription. Contains data only 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 <i>skip_transaction</i> 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)
nstream_writer	bigint	Number of transactions streamed to writer
nstream_file	bigint	Number of transactions streamed to file
nstream_commit	bigint	Number of streaming transactions committed
nstream_abort	bigint	Number of streaming transactions aborted
nstream_start	bigint	Number of STREAT START messages processed
nstream_stop	bigint	Number of STREAM STOP messages processed

continues on next page

Table 54 – continued from previous page

Column	Type	Description
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 <i>track_io_timing</i> is enabled, otherwise zero)
blk_write_time	double precision	Total time the subscription spent writing blocks, in milliseconds (if <i>track_io_timing</i> 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 their modes.

bdr.subscription columns

Name	Type	Description
sub_id	oid	ID of the subscription
sub_name	name	Name of the subscription
nodegroup_id	oid	ID of nodegroup
origin_node_id	oid	ID of origin node
source_node_id	oid	ID of source node
target_node_id	oid	ID of target node
subscription_mode	char	Mode of subscription
sub_enabled	bool	Whether the subscription is enabled (should be replication)
apply_delay	interval	How much behind should the apply of changes on this subscription be (normally 0)
slot_name	name	Slot on upstream used by this subscription
origin_name	name	Local origin used by this subscription
num_writers	int	Number of writer processes this subscription uses
streaming_mode	char	Streaming configuration for the subscription
replication_sets	text[]	Replication sets replicated by this subscription (NULL = all)
forward_origin	text[]	Origins forwarded by this subscription (NULL = all)

bdr.subscription_summary

This view contains summary information about all BDR subscriptions that the local node has to other nodes.

bdr.subscription_summary columns

Name	Type	Description
node_group_name	name	Name of the BDR group the node is part of
sub_name	name	Name of the subscription
origin_name	name	Name of the origin node
target_name	name	Name of the target node (normally local node)
sub_enabled	bool	Is the subscription enabled
sub_slot_name	name	Slot name on the origin node used by this subscription
sub_replication_sets	text[]	Replication sets subscribed
sub_forward_origins	text[]	Does the subscription accept changes forwarded from other nodes besides the origin
sub_apply_delay	interval	Delay transactions by this much compared to the origin
sub_origin_name	name	Replication origin name used by this subscription
bdr_subscription_mode	char	Subscription mode
subscription_status	text	Status of the subscription worker
node_group_id	oid	The OID of the BDR group the node is part of
sub_id	oid	The OID of the subscription
origin_id	oid	The OID of the origin node
target_id	oid	The OID of the target node
receive_lsn	pg_lsn	Latest LSN of any change or message received (this can go backwards in case of restarts)
receive_commit_lsn	pg_lsn	Latest LSN of last COMMIT received (this can go backwards in case of restarts)
last_xact_replay_lsn	pg_lsn	LSN of last transaction replayed on this subscription
last_xact_flush_lsn	timestampz	LSN of last transaction replayed on this subscription that's flushed durably to disk
last_xact_replay_timestamp	timestampz	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 BDR cluster. We consider replication to be blocked when the subscription 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 first transaction after restart is very big and still being applied, the `replication_blocked` result might be wrong.

If this is a logical standby node, then only the status for its upstream node is shown. Similarly, replication status isn't shown for subscriber-only nodes since they never send replication changes to other nodes.

bdr.replication_status columns

Column	Type	Description
node_id	oid	OID of the local node
node_name	name	Name of the local node
origin_node_id	oid	OID of the origin node
origin_node_name	name	Name of the origin node
sub_id	oid	OID of the subscription for this origin node
sub_name	name	Name of the subscription for this origin node
connected	boolean	Is this node connected to the origin node?
replication_blocked	boolean	Is the replication currently blocked for this origin?
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 for this origin

bdr.tables

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

bdr.tables columns

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

(*) These columns are reserved for future use and should currently be NULL

bdr.trigger

In 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 columns that are easier to read.

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.

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

bdr.writers

Specific information about BDR writer processes.

bdr.writers columns

Name	Type	Description
sub_name	name	Name of the subscription
pid	int	Process ID of the worker process
syncing_rel	int	OID of the relation being synchronized (if any)
streaming_allowed	text	Can this writer be target of direct to writer streaming
is_streaming	bool	Is there transaction being streamed to this writer
remote_xid	xid	Remote transaction id of the transaction being processed (if any)
remote_commit_lsn	pg_lsn	LSN of last commit processed
commit_queue_position	int	Position in the internal commit queue
commit_timestamp	timestampz	Timestamp of last commit processed
nxacts	bigint	Number of transactions processed by this writer
ncommits	bigint	Number of transactions committed by this writer
naborts	bigint	Number of transactions aborted by this writer
nstream_file	bigint	Number of streamed-to-file transactions processed by this writer
nstream_writer	bigint	Number of streamed-to-writer transactions processed by this writer

bdr.worker_tasks

The `bdr.worker_tasks` view shows BDR'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 isn't specific to the current database and BDR node. State for all BDR nodes on the current PostgreSQL instance is shown. Join on the current database to filter it.

`bdr.worker_tasks` doesn't track walsenders and output plugins.

bdr.worker_tasks columns

Column	Type	Description
<code>task_key_worker_role</code>	integer	Worker role identifier
<code>task_key_worker_role_name</code>	text	Worker role name
<code>task_key_dboid</code>	oid	Database identifier, if available
<code>datname</code>	name	Name of the database, if available
<code>task_key_subid</code>	oid	Subscription identifier, if available
<code>sub_name</code>	name	Name of the subscription, if available
<code>task_key_ext_libname</code>	name	Name of the library (most likely bdr)
<code>task_key_ext_funcname</code>	name	Name of the function entry point
<code>task_key_ext_workername</code>	name	Name assigned to the worker
<code>task_key_remoterelid</code>	oid	Identifier of the remote syncing relation, if available
<code>task_pid</code>	integer	Process ID of the worker
<code>task_registered</code>	timestamp with time zone	Worker registration timestamp
<code>since_registered</code>	interval	Interval since the worker registered
<code>task_attached</code>	timestamp with time zone	Worker attach timestamp
<code>since_attached</code>	interval	Interval since the worker attached
<code>task_exited</code>	timestamp with time zone	Worker exit timestamp
<code>since_exited</code>	interval	Interval since the worker exited
<code>task_success</code>	boolean	Is worker still running?
<code>task_next_launch_not_before</code>	timestamp with time zone	Timestamp when the worker will be restarted again
<code>until_launch_allowed</code>	interval	Time remaining for next launch
<code>task_last_launch_requestor_pid</code>	integer	Process ID that requested launch
<code>task_last_launch_request_time</code>	timestamp with time zone	Timestamp when the request was made
<code>since_last_request</code>	interval	Interval since the last request
<code>task_last_launch_request_approved</code>	boolean	Did the last request succeed?
<code>task_nrequests</code>	integer	Number of requests
<code>task_nregistrations</code>	integer	Number of registrations
<code>task_prev_pid</code>	integer	Process ID of the previous generation
<code>task_prev_registered</code>	timestamp with time zone	Timestamp of the previous registered task
<code>since_prev_registered</code>	interval	Interval since the previous registration
<code>task_prev_launched</code>	timestamp with time zone	Timestamp of the previous launch
<code>since_prev_launched</code>	interval	Interval since the previous launch
<code>task_prev_exited</code>	timestamp with time zone	Timestamp when the previous task exited
<code>since_prev_exited</code>	interval	Interval since the previous task exited
<code>task_first_registered</code>	timestamp with time zone	Timestamp when the first registration happened
<code>since_first_registered</code>	interval	Interval since the first registration

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 that 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
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	text	Node name of the camo partner
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)

continues on next page

Table 69 – continued from previous page

Name	Type	Description
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

bdr.camo_pairs

Information regarding all the CAMO pairs configured in all the cluster.

bdr.camo_pairs columns

Name	Type	Description
node_group_id	oid	Node group ID
left_node_id	oid	Node ID
right_node_id	oid	Node ID
require_raft	bool	Whether switching to local mode requires majority

Note: The names *left* and *right* have no special meaning. BDR4 can configure only symmetric CAMO configuration, i.e., both nodes in the pair are CAMO partners for each other.

bdr.commit_scopes

Catalog storing all possible commit scopes that you can use for `bdr.commit_scope` to enable group commit.

bdr.commit_scopes columns

Name	Type	Description
commit_scope_id	oid	ID of the scope to be referenced
commit_scope_name	name	Name of the scope to be referenced
commit_scope_origin_node_group	oid	Node group for which the rule applies, referenced by ID
sync_scope_rule	text	Definition of the scope

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 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
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
<code>origin_node_name</code>	text	Name of the origin of the subscription
<code>target_node_name</code>	text	Name of the target of the subscription
<code>last_xact_replay_timestamp</code>	text	Timestamp of the last replayed transaction
<code>sub_lag_seconds</code>	text	Lag between now and <code>last_xact_replay_timestamp</code>

bdr.group_versions_details

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

bdr.group_versions_details columns

Name	Type	Description
<code>node_id</code>	oid	Internal node ID
<code>node_name</code>	name	Name of the node
<code>postgres_version</code>	text	PostgreSQL version on the node
<code>bdr_version</code>	text	BDR version on the node

2.23.2 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>	int8	Monotonically increasing epoch number
<code>origin_node_id</code>	oid	Internal node ID of the node that requested creation of this epoch
<code>epoch_consume_timeout</code>	timestampz	Timeout of this epoch
<code>epoch_consumed</code>	boolean	Switches to true as soon as the local node has fully processed the epoch

bdr.internal_node_pre_commit

Internal catalog table. Use the `bdr.node_pre_commit` view.

bdr.sequence_kind

An internal state table storing the type of each nonlocal sequence. We recommend the view `bdr.sequences` for diagnostic purposes.

bdr.sequence_kind columns

Name	Type	Description
seqid	oid	Internal OID of the sequence
seqkind	char	Internal sequence kind (‘l’=local, ‘t’=timeshard, ‘s’=snowflakeid, ‘g’=galloc)

bdr.state_journal

An internal node state journal. 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 is `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	timestampz	Point in time the current node observed the state change

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.

2.24.1 Version Information Functions

`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:

$$\text{MAJOR_VERSION} * 10000 + \text{MINOR_VERSION} * 100 + \text{PATCH_RELEASE}$$

2.24.2 System Information Functions

`bdr.get_relation_stats`

Returns the relation information.

`bdr.get_subscription_stats`

Returns the current subscription statistics.

2.24.3 System and Progress Information Parameters

BDR exposes some parameters that can be queried via `SHOW` in `psql` or using `PQparameterStatus` (or equivalent) from a client application. This section lists all such parameters BDR reports to.

bdr.local_node_id

Upon session initialization, this is set to the node id the client is connected to. This allows an application to figure out what node it is connected to even behind a transparent proxy.

It is also used in combination with CAMO, see the Connection pools and proxies section.

bdr.last_committed_lsn

After every COMMIT of an asynchronous transaction, this parameter is updated to point to the end of the commit record on the origin node. In combination with `bdr.wait_for_apply_queue`, this allows applications to perform causal reads across multiple nodes, i.e. to wait until a transaction becomes remotely visible.

transaction_id

As soon as Postgres assigns a transaction id, this parameter is updated to show the transaction id just assigned, if CAMO is enabled.

2.24.4 Consensus Function

bdr.consensus_disable

Disables the consensus worker on the local node until server restart or until it's re-enabled using `bdr.consensus_enable` (whichever happens first).

Warning: Disabling consensus will disable some features of BDR and eventually will impact availability of the BDR cluster if left disabled for prolonged periods of time. This function should only be used in coordination with Technical Support.

bdr.consensus_enable

Re-enabled disabled consensus worker on local node.

bdr.consensus_proto_version

Returns currently used consensus protocol version by the local node.

Needed by the BDR group reconfiguration internal mechanisms.

bdr.consensus_snapshot_export**Synopsis**

```
bdr.consensus_snapshot_export(version integer DEFAULT NULL)
```

Generate a new BDR consensus snapshot from the currently committed-and-applied state of the local node and return it as bytea.

By default, a snapshot for the highest supported Raft version is exported. But that can be overridden by passing an explicit version number.

The exporting node does not have to be the current Raft leader, nor does it need to be completely up to date with the latest state on the leader. However, such snapshot might not be accepted by `bdr.consensus_snapshot_import()` (see below).

The new snapshot is not automatically stored to the local node's `bdr.local_consensus_snapshot` table. It's only returned to the caller.

The generated snapshot may be passed to `bdr.consensus_snapshot_import()` on any other node(s) in the same BDR nodegroup that is behind the exporting node's raft log position.

The local BDR consensus worker must be disabled for this function to work. Typical usage is:

```
SELECT bdr.bdr_consensus_disable();
\copy (SELECT * FROM bdr.consensus_snapshot_export()) TO my_node_consensus_snapshot.data
SELECT bdr.bdr_consensus_enable();
```

While the BDR consensus worker is disabled, DDL locking attempts on the node will fail or time out, gallo sequences will not get new values, Eager and CAMO transactions will pause or ERROR, and other functionality that needs the distributed consensus system will be disrupted. The required downtime is generally very brief.

Depending on the use case, it may be practical to extract a snapshot that already exists from the `snapshot` field of the `bdr.local_consensus_snapshot` table and use that instead. Doing so does not require that the consensus worker be stopped.

bdr.consensus_snapshot_import

Synopsis

```
bdr.consensus_snapshot_import(IN snapshot bytea)
```

Import a consensus snapshot which was exported by `bdr.consensus_snapshot_export()`, usually from another node in the same BDR nodegroup.

It's also possible to use a snapshot extracted directly from the `snapshot` field of the `bdr.local_consensus_snapshot` table on another node.

This function is useful for resetting a BDR node's catalog state to known good state in case of corruption or user mistake.

The snapshot can be imported if the importing node's `apply_index` is less than or equal to the snapshot-exporting node's `commit_index` at the time the snapshot was generated. See `bdr.get_raft_status()`. A node that cannot accept the snapshot because its logs is already too far ahead will raise an ERROR and make no changes. The imported snapshot does not have to be completely up-to-date, as once the snapshot is imported the node will fetch the remaining changes from the current leader.

The BDR consensus worker must be disabled on the importing node for this function to work. See notes on `bdr.consensus_snapshot_export()` for details.

It's possible to use this to force the local node to generate a new Raft snapshot by running:

```
SELECT bdr.consensus_snapshot_import(bdr.consensus_snapshot_export());
```

This may also cause it to truncate its Raft logs up to the current applied log position.

bdr.consensus_snapshot_verify

Synopsis

```
bdr.consensus_snapshot_verify(IN snapshot bytea)
```

Verify the given consensus snapshot which was exported by `bdr.consensus_snapshot_export()`. The snapshot header contains the version with which it was generated and the node will try to verify it against the same version.

The snapshot may have been exported on the same node or any other node in the cluster. If the node verifying the snapshot does not support the version of the exported snapshot, then an error will be raised.

bdr.get_consensus_status

Returns status information about the current consensus (Raft) worker.

bdr.get_raft_status

Returns status information about the current consensus (Raft) worker. Alias for `bdr.get_consensus_status`.

bdr.raft_leadership_transfer

Synopsis

```
bdr.raft_leadership_transfer(IN node_name text, IN wait_for_completion boolean)
```

Request the node identified by `node_name` to be the Raft leader. The request can be initiated from any of the BDR nodes and it will be internally forwarded to the current leader to transfer the leadership to the designated node. The designated node must be an ACTIVE BDR node and with full voting rights.

If `wait_for_completion` is false, the request is served on best-efforts basis. If the node can't become a leader within `bdr.raft_election_timeout` period, then some other capable node will become the leader again. Also, the leadership can change over the period of time per Raft protocol. A true return result only indicates the request had been submitted successfully.

If `wait_for_completion` is true then the function will wait until the given node becomes the new leader and possibly wait infinitely if the requested node fails to become Raft leader (e.g. due to network issues). It is therefore recommended to always set a `statement_timeout` in combination with `wait_for_completion` to prevent an infinite loop.

2.24.5 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.

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",
```

(continues on next page)

(continued from previous page)

```
        "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 to 30618 or below.

`bdr.wait_for_xid_progress`

This function can be used to wait for the given transaction (identified by its `XID`) originated at the given node (identified by its 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
```

2.24.6 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.

2.24.7 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 and logging*.

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 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 and logging*.

This function calls `bdr.run_on_all_nodes()`.

bdr.monitor_local_replslots

This function uses replication slot status information returned from view `pg_replication_slots` (slot active or inactive) to provide a local check considering all replication slots, except the BDR group slots.

Synopsis

```
bdr.monitor_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() &rarr; setof record (pid integer, is_using_lcr boolean, decoder_
↳ slot_name TEXT, lcr_file_name TEXT)
```

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 will be 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() &rarr; setof record (pid integer, decoded_upto_lsn pg_
↳ lsn, waiting BOOL, waiting_for_lsn pg_lsn)
```

Output columns

- `pid` - the PID of the Decoding Worker (corresponds to the column `active_pid` in `pg_replication_slots`)
- `decoded_upto_lsn` - LSN upto which the Decoding Worker has 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 .

bdr.lag_control

If *Lag control* is enabled, this function shows information about the commit delay and number of nodes conforming to their configured lag measure for the local node and current database.

Synopsis

```
bdr.lag_control()
```

Output columns

- `commit_delay` - current runtime commit delay in fractional milliseconds
- `commit_delay_maximum` - configured maximum commit delay in fractional milliseconds
- `commit_delay_adjustment` - Change to runtime commit delay possible during a sample interval in fractional milliseconds
- `conforming_nodes` - current runtime number of nodes conforming to lag measures
- `conforming_nodes_minimum` - configured minimum number of nodes required to conform to lag measures, below which a commit delay adjustment is applied
- `lag_bytes_threshold` - lag size at which a commit delay is applied in kilobytes
- `lag_bytes_maximum` - configured maximum lag size in kilobytes
- `lag_time_threshold` - lag time at which a commit delay is applied in milliseconds
- `lag_time_maximum` - configured maximum lag time in milliseconds
- `sample_interval` - configured minimum time between lag samples and possible commit delay adjustments in milliseconds

2.24.8 Internal Functions

BDR Message Payload Functions

`bdr.decode_message_response_payload` and `bdr.decode_message_payload`

These functions decode the consensus payloads to a more human-readable output.

Used primarily by the `bdr.global_consensus_journal_details` debug view.

bdr.get_global_locks

This function shows information about global locks held on the local node.

Used to implement the `bdr.global_locks` view, to provide a more detailed overview of the locks.

bdr.get_slot_flush_timestamp

Retrieves the timestamp of the last flush position confirmation for a given replication slot.

Used internally to implement the `bdr.node_slots` view.

BDR Internal Function Replication Functions

`bdr.internal_alter_sequence_set_kind`, `internal_replication_set_add_table`, `internal_replication_set_remove_table`

Functions used internally for replication of the various function calls.

No longer used by the current version of BDR. Only exists for backwards compatibility during rolling upgrades.

bdr.internal_submit_join_request

Submits a consensus request for joining a new node.

Needed by the BDR group reconfiguration internal mechanisms.

bdr.isolation_test_session_is_blocked

A helper function, extending (and actually invoking) the original `pg_isolation_test_session_is_blocked` with an additional check for blocks on global locks.

Used for isolation/concurrency tests.

bdr.local_node_info

This function displays information for the local node, needed by the BDR group reconfiguration internal mechanisms.

The view `bdr.local_node_summary` provides similar information useful for user consumption.

bdr.msgb_connect

Function for connecting to the connection pooler of another node, used by the consensus protocol.

bdr.msgb_deliver_message

Function for sending messages to another node's connection pooler, used by the consensus protocol.

bdr.peer_state_name

This function transforms the node state (`node_state`) into a textual representation, and is used mainly to implement the `bdr.node_summary` view.

bdr.request_replay_progress_update

Requests the immediate writing of a 'replay progress update' Raft message. It is used mainly for test purposes, but can be also used to test if the consensus mechanism is working.

bdr.seq_nextval

Internal implementation of sequence increments.

This function will be used instead of standard `nextval` in queries which interact with *BDR Global Sequences*.

Notes

The following are also internal BDR sequence manipulation functions. `bdr.seq_currval` and `bdr.sql_lastval` are used automatically.

bdr.show_subscription_status

Retrieves information about the subscription status, and is used mainly to implement the `bdr.subscription_summary` view.

bdr.conflict_resolution_to_string

Transforms the conflict resolution from oid to text.

The view `bdr.conflict_history_summary` uses this to give user-friendly information for the conflict resolution.

bdr.conflict_type_to_string

Transforms the conflict type from oid to text.

The view `bdr.conflict_history_summary` uses this to give user-friendly information for the conflict resolution.

bdr.get_node_conflict_resolvers

Display a text string of all the conflict resolvers on the local node.

bdr.reset_subscription_stats

Returns a boolean result after resetting the statistics created by subscriptions, as viewed by `bdr.stat_subscription`.

bdr.reset_relation_stats

Returns a boolean result after resetting the relation stats, as viewed by `bdr.stat_relation`.

bdr.pg_xact_origin

Return origin id of a given transaction.

Synopsis

<code>bdr.pg_xact_origin(xmin xid)</code>

Parameters

- `xid` - Transaction id whose origin is returned

bdr.difference_fix_origin_create

Creates a replication origin with a given name passed as argument, but adding a `bdr_` prefix. It returns the internal id of the origin. This performs same functionality as `pg_replication_origin_create()`, except this requires `bdr_superuser` rather than postgres superuser permissions.

Synopsis**bdr.difference_fix_session_setup**

Marks the current session as replaying from the current origin. The function uses the pre-created `bdr_local_only_origin` local replication origin implicitly for the session. It allows replay progress to be reported. It returns void. This performs the same functionality as `pg_replication_origin_session_setup()`, except that this requires `bdr_superuser` rather than postgres superuser permissions. Note that the earlier form of the function: `bdr.difference_fix_session_setup(text)` has been deprecated and will be removed in upcoming releases.

Synopsis

```
bdr.difference_fix_session_setup()
```

bdr.difference_fix_session_reset

Marks the current session as not replaying from any origin, essentially resetting the effect of `bdr.difference_fix_session_setup()` . It returns void. This performs the same functionality as `pg_replication_origin_session_reset()` , except this requires `bdr_superuser` rather than postgres superuser permissions.

Synopsis

```
bdr.difference_fix_session_reset()
```

bdr.difference_fix_xact_set_avoid_conflict

Marks the current transaction as replaying a transaction that has committed at LSN '0/0' and timestamp '2000-01-01'. This performs the same functionality as `pg_replication_origin_xact_setup('0/0', '2000-01-01')` , except this requires `bdr_superuser` rather than postgres superuser permissions.

Synopsis

```
bdr.difference_fix_xact_set_avoid_conflict()
```

bdr.resynchronize_table_from_node(node_name name, relation regclass)

Resynchronizes the relation from a remote node.

Synopsis

```
bdr.resynchronize_table_from_node(node_name name, relation regclass)
```

Parameters

- `node_name` - the node from which to copy/resync the relation data.
- `relation` - the relation to be copied from the remote node.

Notes

This acquires a global DML lock on the relation, truncates the relation locally, and copies data into it from the remote node.

The relation must exist on both nodes with the same name and definition.

Resynchronization of partitioned tables with identical partition definitions, resynchronization partitioned table to non-partitioned table and vice-versa and resynchronization of referenced tables by temporarily dropping and recreating foreign key constraints are all supported.

After running the function on a referenced table, if the referenced column data no longer matches the referencing column values, it throws an error and function should be rerun after resynchronizing the referencing table data.

Furthermore, it supports resynchronization of table with generated columns, by computing the generated column values locally, after copying the data from remote node.

Currently, row_filters are ignored by this function.

The `bdr.resynchronize_table_from_node` function can be only executed by the owner of the table, provided the owner has `bdr_superuser` privileges.

bdr.consensus_kv_store

Stores value in the consistent KV Store.

Returns timestamp of the value expiration time. This depends on `t1`, if `t1` is `NULL`, then this will return `infinity` if the value was deleted it will return `-infinity`.

Synopsis

```
bdr.consensus_kv_store(key text, value jsonb,  
                      prev_value jsonb DEFAULT NULL, t1 int DEFAULT NULL)
```

Parameters

- `key` - an arbitrary unique key to insert, update or delete.
- `value` - json value to store, if `NULL`, any existing record will be deleted
- `prev_value` - if set, the write operation is only done if the current value is equal to `prev_value`.
- `t1` - time to live of the new value, in milliseconds.

Notes

This is an internal function, mainly used by HARP.

Warning: This function should never be used by user applications.

bdr.consensus_kv_fetch

Fetch value from the consistent KV Store in json format.

Synopsis

```
bdr.consensus_kv_fetch(IN key text) RETURNS jsonb
```

Parameters

- key - an arbitrary key to fetch.

Notes

This is an internal function, mainly used by HARP.

Warning: This function should never be used by user applications.

bdr.alter_subscription_skip_changes_upto

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

Wherever possible such problems should be fixed by making changes to the target side. CREATE ing any missing table that's blocking replication, CREATE a needed role, GRANT a necessary permission, etc. But occasionally a problem can't be fixed that way and it may be necessary to skip entirely over a transaction. Changes are skipped as entire transactions, all or nothing. To decide where to skip to, use log output to find the commit LSN, per the example below, or peek the change stream with the logical decoding functions.

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

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

This function only works on disabled subscriptions.

The usual sequence of steps is:

- identify the problem subscription and LSN of the problem commit
- disable the subscription
- save a copy of the transaction(s) using `pg_catalog.pg_logical_slot_peek_changes` on the source node (if possible)
- `bdr.alter_subscription_skip_changes_upto` on the target node
- apply repaired or equivalent changes on the target manually, if necessary
- re-enable the subscription

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

Synopsis

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

Example

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

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

```
ERROR: XX000: CONFLICT: target_table_missing; resolver skip_if_recently_dropped_  
→returned an error: table does not exist  
CONTEXT: during apply of INSERT from remote relation public.break_me in xact with_  
→commit-end lsn 0/300AC18 xid 131315  
committs 2021-02-02 15:11:03.913792+01 (action #2) (effective sess origin id=2 lsn=0/  
→300AC18)  
while consuming I message from receiver for subscription bdr_regression_bdrgroup_node1_  
→node2 (id=2667578509)  
on node node2 (id=3367056606) from upstream node node1 (id=1148549230, reporiginid=2)
```

In this portion of log we have the information we need: the_target_lsn: **0/300AC18** the_subscription: **bdr_regression_bdrgroup_node1_node2**

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

```
SELECT bdr.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.

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

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

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

```
SELECT bdr.alter_subscription_skip_changes_upto(subscription_name,  
    the_target_lsn);
```

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

Finally, re-enable the subscription:

```
SELECT bdr.alter_subscription_enable(the_subscription);
```