EDB Postgres Distributed

Version 4

Built at 2022-09-13T20:48:21

1 EDB Postgres Distributed

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

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

Compatibility matrix

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

- PGD CLI 1 is supported with BDR 4.1 and later.
- BDR DCS in HARP 2 is supported with BDR 3.7.15 and later and 4.0.1 and later.

2 EDB Postgres Distributed Release notes

The EDB Postgres Distributed documentation describes the latest version of EDB Postgres Distributed 4, including minor releases and patches. The release notes provide information on what was new in each release. For new functionality introduced in a minor or patch release, the content also indicates the release that introduced the feature.

Release Date	EDB Postgres Distributed	BDR	HARP	CLI	TPAexec
2022 Aug 22	4.2.0	4.2.0	2.2.0	1.1.0	23.5
2022 June 21	4.1.1	4.1.1	2.1.1	1.0.0	23.2
2022 May 17	4.1.0	4.1.0	2.1.0	1.0.0	23.1
2022 Mar 31	4.0.3	-	2.0.3	-	22.10
2022 Feb 24	4.0.2	4.0.2	2.0.2	-	22.9
2022 Jan 31	4.0.1	4.0.1	2.0.1	-	22.6
2021 Dec 01	4.0.0	4.0.0	2.0.0	-	22.9

2.1 Release notes for EDB Postgres Distributed version 4.2.0

EDB Postgres Distributed version 4.2.0 is a minor release of EDB Postgres Distributed 4, which includes new features as well as fixes for issues identified in previous versions.

Component	Version	Type	Description
BDR	4.2.0	Enhancement	Allow consumption of the reserved galloc sequence slot (BDR-2367, RT83437, RT68255) The galloc sequence slot reserved for future use by background allocator can be consumed in the presence of consensus failure.
BDR	4.2.0	Bug fix	Fix spurious pglogical receiver timeouts on idle connections (RT82315) When there is nothing to replicate and apply, the pglogical receiver may misinterpret the state as timeout and exit with error "ERROR: 08006: terminating pglogical receiver due to timeout". Correct this misinterpretation.
BDR	4.2.0	Bug fix	Fix "insufficient privileges" error for non-superuser trusted extensions (BDR-2350) A CREATE EXTENSION command executed by a non-superuser on PostgreSQL 13+ may succeed on the BDR node where it is executed but will fail to be replicated to other nodes breaking replication. Fix this by allowing such commands to be applied on non-origin nodes.
BDR	4.2.0	Bug fix	Avoid a race condition when reconstructing global_locks upon restart if Parallel Apply is enabled (RT83435, BDR-2383) When parallel apply is enabled, multiple writers may try to recover the same global lock causing a deadlock. The BDR node where this happens will be rendered unrecoverable. Fix this by letting a single writer recover all the global locks.
BDR	4.2.0	Bug fix	Fix spurious segmentation faults when conflicts are logged to bdr.conflict_history (BDR-2403, RT83436, RT83928) When conflicts are logged to the catalog bdr.conflict_history, the pglogical writer process may crash because of a segmentation fault due to an invalid pointer being used. Fix this usage.
BDR	4.2.0	Bug fix	Clean up the replication slot when bdr_init_physical fails (BDR-2364, RT74789) If bdr_init_physical aborts without being able to join the node, it will leave behind an inactive replication slot. Remove such a replication slot when it is inactive before an irregular exit.
HARP	2.2.0	Enhancement	Add support for sslpassword in a connection string, including processing hook (HNG-626)

2.2 Release notes for EDB Postgres Distributed version 4.1.1

EDB Postgres Distributed version 4.1.1 includes bug fixes for BDR 4.1.1 and HARP 2.1.1. See the TPAexec Release Notes for version 23.1 for information on the changes and bug fixes in TPAexec.

Component	Version	Type	Description
BDR	4.1.1	Feature	Add generic function bdr.is_node_connected returns true if the walsender for a given peer is active.
BDR	4.1.1	Feature	Add generic function bdr.is_node_ready returns boolean if the lag is under a specific span.
BDR	4.1.1	Bug fix	Add support for a `--link` argument to bdr_pg_upgrade for using hard-links.
BDR	4.1.1	Bug fix	Prevent removing a `bdr.remove_commit_scope` if still referenced by any `bdr.node_group` as the default commit scope.
BDR	4.1.1	Bug fix	Correct Raft based switching to Local Mode for CAMO pairs of nodes (RT78928)
BDR	4.1.1	Bug fix	Prevent a potential segfault in bdr.drop_node for corner cases (RT81900)
BDR	4.1.1	Bug fix	Prevent use of CAMO or Eager All Node transactions in combination with transaction streaming. Transaction streaming turned out to be problematic in combination with CAMO and Eager All Node transactions. Until this is resolved, BDR now prevents its combined use. This may require CAMO deployments to adjust their configuration to disable transaction streaming, see Transaction Streaming Configuration.
HARP	2.1.1	Enhancement	Log a warning on loss of DCS connection
HARP	2.1.1	Enhancement	Log a warning when metadata refresh is taking too long - usually due to high latency network
HARP	2.1.1	Bug fix	Restart harp_proxy.service on a failure
HARP	2.1.1	Bug fix	Fix concurrency issue with connection management in harpctl
HARP	2.1.1	Bug fix	Don't try to proxy connections to previous leader on unmanaged cluster
HARP	2.1.1	Bug fix	Don't panic in haprctl when location is empty

2.3 Release notes for EDB Postgres Distributed version 4.1.0

EDB Postgres Distributed version 4.1.0 includes the following:

Component	Version	Type	Description
CLI	1.0.0	Feature	Ability to gather information such as the current state of replication, consensus, and nodes for an EDB Postgres Distributed cluster using new command-line interface (CLI).
BDR	4.1.0	Feature	Support in-place major upgrades of Postgres on a data node with a new command-line utility, bdr_pg_upgrade. This utility uses the standard pg_upgrade command, and reduces the time and network bandwidth needed to do major version upgrades of a EDB Postgres Distributed cluster.
BDR	4.1.0	Feature	Enable the ability to configure a replication lag threshold. After the threshold is met, the transaction commits get throttled. This threshold allows limiting RPO without incurring the latency impact on every transaction that comes with synchronous replication.

Component	Version	Type	Description
BDR	4.1.0	Feature	Global sequences are automatically configured based on data type replacing the need to set up custom sequence handling configuration on every node. The new SnowflakeID algorithm replaces Timeshard, which had limitations.
BDR	4.1.0	Feature	Add a new SQL-level interface for configuring synchronous replication durability and visibility options by group rather than by node. This approach allows you to configure all nodes consistently from a single place instead of using config files.
BDR	4.1.0	Feature	Add a new synchronous replication option, Group Commit, which allows a quorum to be required before committing a transaction in a EDB Postgres Distributed group.
BDR	4.1.0	Feature	Allow a Raft request to be required for CAMO switching to Local Mode. Add a `require_raft` 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. (RT78928)
BDR	4.1.0	Feature	Allow replication to continue on `ALTER TABLE ... DETACH PARTITION CONCURRENTLY` of already detached partition. Similarly to how BDR 4 handles `CREATE INDEX CONCURRENTLY` when same index already exists, we now allow replication to continue when `ALTER TABLE ... DETACH PARTITION CONCURRENTLY` is receiver for partition that has been already detached. (RT78362)
BDR	4.1.0	Feature	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
BDR	4.1.0	Feature	Add `bdr.lock_table_locking` configuration variable. When enabled this changes behavior of `LOCK TABLE` command to take take a global DML lock
BDR	4.1.0	Feature	Implement buffered write for LCR segment file. This should reduce I/O and improve CPU usage of the Decoding Worker.
BDR	4.1.0	Feature	Add support for partial unique index lookups for conflict detection. Indexes on expression are however still not supported for conflict detection. (RT78368)
BDR	4.1.0	Feature	Add additional statistics to `bdr.stat_subscription`: - 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
BDR	4.1.0	Feature	Add execute_locally option to bdr.replicate_ddl_command.This allows optional queueing of ddl commands for replication to other groups without executing it locally. (RT73533)
BDR	4.1.0	Feature	Add `fast` argument to `bdr.alter_subscription_disable()`. The argument only influences the behavior of `immediate`. When set to `true` (default) it will stop the workers without letting them finish the current work. (RT79798)
BDR	4.1.0	Feature	Simplify `bdr.{add,remove}_camo_pair` functions to return void.

Component	Version	Type	Description
BDR	4.1.0	Feature	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.
BDR	4.1.0	Feature	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.
BDR	4.1.0	Feature	Improve reporting of remote errors during node join.
BDR	4.1.0	Feature	Make autopartition worker's max naptime configurable.
BDR	4.1.0	Feature	Add ability to request partitions upto the given upper bound with autopartition.
BDR	4.1.0	Feature	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.
BDR	4.1.0	Feature	Add `bdr.accept_connections` configuration variable. When `false`, walsender connections to replication slots using BDR output plugin will fail. This is useful primarily during restore of single node from backup.
BDR	4.1.0	Bug fix	Keep the `lock_timeout` as configured on non-CAMO-partner BDR nodes. A CAMO partner uses a low `lock_timeout` when applying transactions from its origin node. This was inadvertently done for all BDR nodes rather than just the CAMO partner, which may have led to spurious `lock_timeout` errors on pglogical writer processes on normal BDR nodes.
BDR	4.1.0	Bug fix	Show a proper wait event for CAMO / Eager confirmation waits. Show correct "BDR Prepare Phase"/"BDR Commit Phase" in `bdr.stat_activity` instead of the default “unknown wait event”. (RT75900)
BDR	4.1.0	Bug fix	Reduce log for bdr.run_on_nodes. Don't log when setting `bdr.ddl_replication` to off if it's done with the "run_on_nodes" variants of function. This eliminates the flood of logs for monitoring functions. (RT80973)
BDR	4.1.0	Bug fix	Fix replication of arrays of composite types and arrays of builtin types that don't support binary network encoding
BDR	4.1.0	Bug fix	Fix replication of data types created during bootstrap
BDR	4.1.0	Bug fix	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.
BDR	4.1.0	Bug fix	Don't wait for autopartition tasks to complete on parting nodes
BDR	4.1.0	Bug fix	Limit the `bdr.standby_slot_names` check when reporting flush position only to physical slots. Otherwise flush progress is not reported in presence of disconnected nodes when using `bdr.standby_slot_names`. (RT77985, RT78290)
BDR	4.1.0	Bug fix	Request feedback reply from walsender if we are close to wal_receiver_timeout
BDR	4.1.0	Bug fix	Don't record dependency of auto-paritioned 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 `pg_depend` .
BDR	4.1.0	Bug fix	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.
BDR	4.1.0	Bug fix	Only do post `CREATE` commands processing when BDR node exists in the database.
BDR	4.1.0	Bug fix	Don't try to log ERROR conflicts to conflict history table.

Component	Version	Type	Description
BDR	4.1.0	Bug fix	Fixed segfault where a conflict_slot was being used after it was released during multi-insert (COPY) (RT76439).
BDR	4.1.0	Bug fix	Prevent walsender processes spinning when facing lagging standby slots. 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. (RT80295, RT78290)
BDR	4.1.0	Bug fix	Fix handling of `wal_sender_timeout` when `bdr.standby_slot_names` are used (RT78290)
BDR	4.1.0	Bug fix	Fix reporting of disconnected slots in `bdr.monitor_local_replslots`. They could have been previously reported as missing instead of disconnected.
BDR	4.1.0	Bug fix	Fix apply timestamp reporting for down subscriptions in `bdr.get_subscription_progress()` function and in the `bdr.subscription_summary` that uses that function. It would report garbage value before.
BDR	4.1.0	Bug fix	Fix snapshot handling in various places in BDR workers.
BDR	4.1.0	Bug fix	Be more consistent about reporting timestamps and LSNs as NULLs in monitoring functions when there is no available value for those.
BDR	4.1.0	Bug fix	Reduce log information when switching between writer processes.
BDR	4.1.0	Bug fix	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.
BDR	4.1.0	Bug fix	Use 64 bits for calculating lag size in bytes. To eliminate risk of overflow with large lag.
HARP	2.1.0	Feature	The BDR DCS now uses a push notification from the consensus rather than through polling nodes. This change reduces the time for new leader selection and the load that HARP does on the BDR DCS since it doesn't need to poll in short intervals anymore.
HARP	2.1.0	Feature	TPA now restarts each HARP Proxy one by one and wait until they come back to reduce any downtime incurred by the application during software upgrades.
HARP	2.1.0	Feature	The support for embedding PGBouncer directly into HARP Proxy is now deprecated and will be removed in the next major release of HARP. It's now possible to configure TPA to put PGBouncer on the same node as HARP Proxy and point to that HARP Proxy.
HARP	2.1.0	Bug fix	`harpctl promote <node_name>` would occasionally promote a different node than the one specified. This has been fixed. (RT75406)
HARP	2.1.0	Bug fix	Fencing would sometimes fail when using BDR as the Distributed Consensus Service. This has been corrected.
HARP	2.1.0	Bug fix	`harpctl apply` no longer turns off routing for leader after the cluster has been established. (RT80790)
HARP	2.1.0	Bug fix	Harp-manager no longer exits if it cannot start a failed database. Harp-manager will keep retrying with randomly increasing periods. (RT78516)
HARP	2.1.0	Bug fix	The internal pgbouncer proxy implementation had a memory leak. This has been remediated.

2.4 Release notes for EDB Postgres Distributed version 4.0.3

This is a patch release of HARP 2 that includes fixes for issues identified in previous versions.

Component	Version	Type	Description
HARP	2.0.3	Enhancement	HARP Proxy supports read-only user dedicated TLS Certificate (RT78516)
HARP	2.0.3	Bug fix	HARP Proxy continues to try and connect to DCS instead of exiting after 50 seconds. (RT75406)

2.5 Release notes for EDB Postgres Distributed version 4.0.2

This is a maintenance release for BDR 4.0 and HARP 2.0 which includes minor improvements as well as fixes for issues identified in previous versions.

Component	Version	Type	Description
BDR	4.0.2	Enhancement	Add `bdr.max_worker_backoff_delay` (BDR-1767) This changes the handling of the backoff delay to exponentially increase from `bdr.min_worker_backoff_delay` to `bdr.max_worker_backoff_delay` in presence of repeated errors. This reduces log spam and in some cases also prevents unnecessary connection attempts.
BDR	4.0.2	Enhancement	Add `execute_locally` option to `bdr.replicate_ddl_command()` (RT73533) This allows optional queueing of ddl commands for replication to other groups without executing it locally.
BDR	4.0.2	Enhancement	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.
BDR	4.0.2	Bug fix	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.
BDR	4.0.2	Bug fix	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.
BDR	4.0.2	Bug fix	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.

Component	Version	Type	Description
BDR	4.0.2	Bug fix	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.
BDR	4.0.2	Bug fix	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.
BDR	4.0.2	Bug fix	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.
BDR	4.0.2	Bug fix	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.
BDR	4.0.2	Bug fix	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.
BDR	4.0.2	Bug fix	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`.
BDR	4.0.2	Bug fix	Fix replication of data types created during bootstrap (BDR-1784)
BDR	4.0.2	Bug fix	Fix replication of arrays of builtin types that don't have binary transfer support (BDR-1042)
BDR	4.0.2	Bug fix	Prevent CAMO configuration warnings if CAMO is not being used (BDR-1825)
HARP	2.0.2	Enhancement	BDR consensus now generally available. HARP offers multiple options for Distributed Consensus Service (DCS) source: etcd and BDR. The BDR consensus option can be used in deployments where etcd isn't present. Use of the BDR consensus option is no longer considered beta and is now supported for use in production environments.

Component	Version	Type	Description
HARP	2.0.2	Enhancement	Transport layer proxy now generally available. HARP offers multiple proxy options for routing connections between the client application and database: application layer (L7) and transport layer (L4). The network layer 4 or transport layer proxy simply forwards network packets, and layer 7 terminates network traffic. The transport layer proxy, previously called simple proxy, is no longer considered beta and is now supported for use in production environments.

2.6 Release notes for EDB Postgres Distributed version 4.0.1

This is a maintenance release for BDR 4.0 and HARP 2.0 which includes minor Enhancements as well as fixes for issues identified in previous versions.

Component	Version	Type	Description
BDR	4.0.1	Enhancement	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.
BDR	4.0.1	Enhancement	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.
BDR	4.0.1	Enhancement	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.
BDR	4.0.1	Enhancement	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.
BDR	4.0.1	Bug fix	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.

Component	Version	Type	Description
BDR	4.0.1	Bug fix	Fix Raft snapshot handling of `bdr.camo_pairs` (BDR-1753) The previous release would not correctly propagate changes to the CAMO pair configuration when they were received via Raft snapshot.
BDR	4.0.1	Bug fix	Correctly handle Raft snapshots from BDR 3.7 after upgrades (BDR-1754)
BDR	4.0.1	Bug fix	Upgrading a CAMO configured cluster taking into account the `bdr.camo_pairs` in the snapshot while still excluding the ability to perform in place upgrade of a cluster (due to upgrade limitations unrelated to CAMO).
BDR	4.0.1	Bug fix	Switch from CAMO to Local Mode only after timeouts (RT74892) Do not use the `catchup_interval` estimate when switching from CAMO protected to Local Mode, as that could induce inadvertent switching due to load spikes. Use the estimate only when switching from Local Mode back to CAMO protected (to prevent toggling forth and back due to lag on the CAMO partner).
BDR	4.0.1	Bug fix	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.
BDR	4.0.1	Bug fix	Prevent duplicate values generated locally by galloc sequence in high concurrency situations when the new chunk is used (RT76528) The galloc sequence could have temporarily produce duplicate value when switching which chunk is used locally (but not across nodes) if there were multiple sessions waiting for the new value. This is now fixed.
BDR	4.0.1	Bug fix	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.
BDR	4.0.1	Bug fix	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.

Component	Version	Type	Description
BDR	4.0.1	Bug fix	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.
BDR	4.0.1	Bug fix	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.
HARP	2.0.1	Enhancement	Support for selecting a leader per location rather than relying on DCS like etcd to have separate setup in different locations. This still requires a majority of nodes to survive loss of a location, so an odd number of both locations and database nodes is recommended.
HARP	2.0.1	Enhancement	The BDR DCS now uses a push notification from the consensus rather than through polling nodes. This change reduces the time for new leader selection and the load that HARP does on the BDR DCS since it doesn't need to poll in short intervals anymore.
HARP	2.0.1	Enhancement	TPA now restarts each HARP Proxy one by one and wait until they come back to reduce any downtime incurred by the application during software upgrades.
HARP	2.0.1	Enhancement	The support for embedding PGBouncer directly into HARP Proxy is now deprecated and will be removed in the next major release of HARP. It's now possible to configure TPA to put PGBouncer on the same node as HARP Proxy and point to that HARP Proxy.
HARP	2.0.1	Bug fix	`harpctl promote <node_name>` would occasionally promote a different node than the one specified. This has been fixed. [Support Ticket #75406]
HARP	2.0.1	Bug fix	Fencing would sometimes fail when using BDR as the Distributed Consensus Service. This has been corrected.
HARP	2.0.1	Bug fix	`harpctl apply` no longer turns off routing for leader after the cluster has been established. [Support Ticket #80790]
HARP	2.0.1	Bug fix	Harp-manager no longer exits if it cannot start a failed database. Harp-manager will keep retrying with randomly increasing periods. [Support Ticket #78516]
HARP	2.0.1	Bug fix	The internal pgbouncer proxy implementation had a memory leak. This has been remediated.

2.7 Release notes for EDB Postgres Distributed version 4.0.0

EDB Postgres Distributed version 4.0.0 contains BDR 34.0 and HARP 2.0. 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.

Component	Version	Type	Description
BDR	4.0.0	Feature	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
BDR	4.0.0	Feature	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
BDR	4.0.0	Feature	Support streaming of large transactions. This allows BDR to stream a large transaction (greater than `logical_decoding_work_mem` 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.

Component	Version	Type	Description
BDR	4.0.0	Feature	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 `bdr_config` command line utility or `bdr.bdr_features()` database function. See Choosing a Postgres distribution for more details.
BDR	4.0.0	Feature	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 `pglogical.` to `bdr.`.
BDR	4.0.0	Feature	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. - `COPY` and `CREATE INDEX CONCURRENTLY` 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 `bdr.min_worker_backoff_delay` now defaults to 1s so that subscriptions retry connection only once per second on error
BDR	4.0.0	Feature	Greatly reduced the chance of false positives in conflict detection during node join for table that use origin based conflict detection
BDR	4.0.0	Feature	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 `bdr.camo_partner_of` and `bdr.camo_origin_for` and replaces the functions `bdr.get_configured_camo_origin_for()` and `get_configured_camo_partner_of` with `bdr.get_configured_camo_partner`.
HARP	2.0.0	Change	Complete rewrite of system in golang to optimize all operations
HARP	2.0.0	Change	Cluster state can now be bootstrapped or revised via YAML
HARP	2.0.0	Feature	Configuration now in YAML, configuration file changed from `harp.ini` to `config.yml`

Component	Version	Type	Description
HARP	2.0.0	Feature	HARP Proxy deprecates need for HAProxy in supported architecture. The use of HARP Router to translate DCS contents into appropriate online or offline states for HTTP-based URI requests meant a load balancer or HAProxy was necessary to determine the lead master. HARP Proxy now does this automatically without periodic iterative status checks.
HARP	2.0.0	Feature	Utilizes DCS key subscription to respond directly to state changes. With relevant cluster state changes, the cluster responds immediately, resulting in improved failover and switchover times.
HARP	2.0.0	Feature	Compatibility with etcd SSL settings. It is now possible to communicate with etcd through SSL encryption.
HARP	2.0.0	Feature	Zero transaction lag on switchover. Transactions are not routed to the new lead node until all replicated transactions are replayed, thereby reducing the potential for conflicts.
HARP	2.0.0	Feature	Experimental BDR Consensus layer. Using BDR Consensus as the Distributed Consensus Service (DCS) reduces the amount of change needed for implementations.
HARP	2.0.0	Feature	Experimental built-in proxy. Proxy implementation for increased session control.

3 Known issues

This section discusses currently known issues in EDB Postgres Distributed 4.

Data Consistency

Read about Conflicts to understand the implications of the asynchronous operation mode in terms of data consistency.

List of issues

These known issues are tracked in BDR's ticketing system and are expected to be resolved in a future release.

- Performance of HARP in terms of failover and switchover time depends non-linearly on the latencies between DCS nodes. Which is why we currently recommend using etcd cluster per region for HARP in case of EDB Postgres Distributed deployment over multiple regions (typically the Gold and Platinum layouts). TPAexec already sets up the

etcd do run per region cluster for these when `harp_consensus_protocol` option is set to `etcd` in the `config.yml`.

It's recommended to increase the `leader_lease_duration` HARP option (`harp_leader_lease_duration` in TPAexec) for DCS deployments across higher latency network.

- If the resolver for the `update_origin_change` conflict is set to `skip`, `synchronous_commit=remote_apply` is used, and concurrent updates of the same row are repeatedly applied on two different nodes, then one of the update statements might hang due to a deadlock with the BDR writer. As mentioned in the Conflicts chapter, `skip` is not the default resolver for the `update_origin_change` conflict, and this combination isn't intended to be used in production. It discards one of the two conflicting updates based on the order of arrival on that node, which is likely to cause a divergent cluster. In the rare situation that you do choose to use the `skip` conflict resolver, note the issue with the use of the `remote_apply` mode.
- The Decoding Worker feature doesn't work with CAMO/EAGER/Group Commit. Installations using CAMO/Eager/Group Commit must keep `enable_wal_decoder` disabled.
- Decoding Worker works only with the default replication sets.
- Lag control doesn't adjust commit delay in any way on a fully isolated node, that is, in case all other nodes are unreachable or not operational. As soon as at least one node is connected, replication lag control picks up its work and adjusts the BDR commit delay again.
- For time-based lag control, BDR currently uses the lag time (measured by commit timestamps) rather than the estimated catchup time that's based on historic apply rate.
- Changing the CAMO partners in a CAMO pair isn't currently possible. It's possible only to add or remove a pair. Adding or removing a pair doesn't need a restart of Postgres or even a reload of the configuration.
- Group Commit cannot be combined with CAMO or Eager All Node replication. Eager Replication currently only works by using the "global" BDR commit scope.
- Neither Eager replication nor Group Commit support `synchronous_replication_availability = 'async'`.
- Group Commit doesn't support a timeout of the commit after `bdr.global_commit_timeout`.
- Transactions using Eager Replication can't yet execute DDL, nor do they support explicit two-phase commit. The TRUNCATE command is allowed.
- Not all DDL can be run when either CAMO or Group Commit is used.
- Parallel apply is not currently supported in combination with Group Commit, please make sure to disable it when using Group Commit by either setting `num_writers` to 1 for the node group (using `bdr.alter_node_group_config`) or via the GUC `bdr.writers_per_subscription` (see Configuration of Generic Replication).
- There currently is no protection against altering or removing a commit scope. Running transactions in a commit scope that is concurrently being altered or removed can lead to the transaction blocking or replication stalling completely due to an error on the downstream node attempting to apply the transaction. Ensure that any transactions using a specific commit scope have finished before altering or removing it.

List of limitations

This is a (non-comprehensive) list of limitations that are expected and are by design. They are not expected to be

resolved in the future.

- Replacing a node with its physical standby doesn't work for nodes that use CAMO/Eager/Group Commit. Combining physical standbys and BDR in general isn't recommended, even if otherwise possible.
- A `galloc` sequence might skip some chunks if the sequence is created in a rolled back transaction and then created again with the same name. This can also occur if it is created and dropped when DDL replication isn't active and then it is created again when DDL replication is active. The impact of the problem is mild, because the sequence guarantees aren't violated. The sequence skips only some initial chunks. Also, as a workaround you can specify the starting value for the sequence as an argument to the `bdr.alter_sequence_set_kind()` function.
- Legacy BDR synchronous replication uses a mechanism for transaction confirmation different from the one used by CAMO, Eager, and Group Commit. The two are not compatible and must not be used together. Therefore, nodes that appear in `synchronous_standby_names` must not be part of CAMO, Eager, or Group Commit configuration. Using synchronous replication to other nodes, including both logical and physical standby is possible.

4 Terminology

The terminology that follows is important for understanding EDB Postgres Distributed functionality and the requirements that it addresses in the realms of high availability, replication, and clustering.

Asynchronous replication

Copies data to cluster members after the transaction completes on the origin node. Asynchronous replication can provide higher performance and lower latency than synchronous replication. However, it introduces the potential for conflicts because of multiple concurrent changes. You must manage any conflicts that arise.

Availability

The probability that a system will operate satisfactorily at a given time when used in a stated environment. For many people, this is the overall amount of uptime versus downtime for an application. (See also Nines)

CAMO or commit-at-most-once

Wraps Eager Replication with additional transaction management at the application level to guard against a transaction being executed more than once. This is critical for high-value transactions found in payments solutions. It is roughly equivalent to the Oracle feature Transaction Guard.

Clustering

An approach for high availability in which multiple redundant systems are managed to avoid single points of failure. It appears to the end user as one system.

Data sharding

Enables scaling out a database by breaking up data into chunks called *shards* and distributing them across separate nodes.

Eager Replication for BDR

Conflict-free replication with all cluster members; technically, this is synchronous logical replication using two phase-commit (2PC).

Eventual consistency

A distributed computing consistency model stating changes to the same item in different cluster members will converge to the same value. With BDR this is achieved through asynchronous logical replication with conflict resolution and conflict-free replicated data types.

Failover

The automated process that recognizes a failure in a highly available database cluster and takes action to connect the application to another active database. The goal is to minimize downtime and data loss.

Horizontal scaling** or **scale out

A modern distributed computing approach that manages workloads across multiple nodes, such as scaling out a web server to handle increased traffic.

Logical replication

Provides more flexibility than physical replication in terms of selecting the data replicated between databases in a cluster. Also important is that cluster members can be on different versions of the database software.

Nines

A measure of availability expressed as a percentage of uptime in a given year. Three nines (99.9%) allows for 43.83 minutes of downtime per month. Four nines (99.99%) allows for 4.38 minutes of downtime per month. Five nines (99.999%) allows for 26.3 seconds of downtime per month.

Node

One database server in a cluster. A term "node" differs from the term "database server" because there is more than one node in a cluster. A node includes the database server, the OS, and the physical hardware, which is always separate from other nodes in a high-availability context.

Physical replication

Copies all changes from a database to one or more standby cluster members by copying an exact copy of database disk blocks. While fast, this method has downsides. For example, only one master node can run write transactions. Also, you can use this method only where all cluster members are on the same major version of the database software, in addition to several other more complex restrictions.

Read scalability

Can be achieved by introducing one or more read replica nodes to a cluster and have the application direct writes to the primary node and reads to the replica nodes. As the read workload grows, you can increase the number of read replica nodes to maintain performance.

Recovery point objective (RPO)

The maximum targeted period in which data might be lost due to a disruption in delivery of an application. A very low or minimal RPO is a driver for very high availability.

Recovery time objective (RTO)

The targeted length of time for restoring the disrupted application. A very low or minimal RTO is a driver for very high availability.

Single point of failure (SPOF)

The identification of a component in a deployed architecture that has no redundancy and therefore prevents you from achieving higher levels of availability.

Switchover

A planned change in connection between the application and the active database node in a cluster, typically done for maintenance.

Synchronous replication

When changes are updated at all participating nodes at the same time, typically leveraging two-phase commit. While this approach delivers immediate consistency and avoids conflicts, a performance cost in latency occurs due to the coordination required across nodes.

Two-phase commit (2PC)

A multi-step process for achieving consistency across multiple database nodes.

Vertical scaling** or **scale up

A traditional computing approach of increasing a resource (CPU, memory, storage, network) to support a given workload until the physical limits of that architecture are reached, e.g., Oracle Exadata.

Write scalability

Occurs when replicating the writes from the original node to other cluster members becomes less expensive. In vertical-scaled architectures, write scalability is possible due to shared resources. However, in horizontal scaled (or nothing-shared) architectures, this is possible only in very limited scenarios.

5 Overview

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

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

Postgres server

Three different Postgres distributions can be used:

- PostgreSQL - open source
- EDB Postgres Extended Server - PostgreSQL compatible and optimized for replication
- EDB Postgres Advanced Server - Oracle compatible, optimized for replication, and additional enterprise features

What Postgres distribution and version is right for you depends on the features you need. See the feature matrix in Choosing a Postgres distribution for detailed comparison.

BDR

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

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

HARP

HARP is connection management tool for a EDB Postgres Distributed cluster.

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

---

6 BDR (Bi-Directional Replication)

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 Postgres clusters with high availability up to five 9s.

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

By default, BDR uses asynchronous replication, applying changes on the peer nodes only after the local commit. Multiple synchronous replication options are also available.

Basic architecture

Multiple groups

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

the whole BDR cluster.

Multiple masters

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

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

You don't have to write to all the masters all of the time. A frequent configuration directs writes mostly to just one master.

Asynchronous, by default

Changes made on one BDR node aren't replicated to other nodes until they're committed locally. As a result, the data isn't exactly the same on all nodes at any given time. Some nodes have data that hasn't 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's no forwarding of data in BDR except in special circumstances such as adding and removing nodes. Data can arrive from outside the EDB Postgres Distributed 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 on 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 aren't 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's safe to do so.

High availability

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

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

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

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

Architectural options and performance

Always On architectures

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

The group is the basic building block consisting of 2+ nodes (servers). In a group, each node is in a different availability zone, 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, you can easily repair or replace it by copying an existing node from the group.

The Always On architectures are built from either one group in a single location or two groups in two separate locaions. Each group provides HA and IS. When two groups are leveraged in remote locations, they together also provide disaster recovery (DR).

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

One node in each group 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 isn't the goal of this architecture.

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

Secondary applications might execute against the shadow nodes, although these are reduced or interrupted if the main application begins using that node.

In the future, one node will be elected as the main replicator to other groups, limiting CPU overhead of replication as the cluster grows and minimizing the bandwidth to other groups.

Supported PostgreSQL database servers

BDR is compatible with Postgres, EDB Postgres Extended Server, and EDB Postgres Advanced Server distributions and can be deployed as a standard Postgres extension. See Compatibility matrix for details of supported version combinations.

Some key BDR features depend on certain core capabilities being available in the targeted Postgres database server. Therefore, BDR users must also adopt the Postgres database server distribution that's best suited to their business needs. For example, if having the BDR feature Commit At Most Once (CAMO) is mission critical to your use case, don't adopt the community PostgreSQL distribution because it doesn't have the core capability required to handle CAMO. See the full feature matrix compatibility in Choosing a Postgres distribution.

BDR offers close to native Postgres compatibility. However, some access patterns don't necessarily work as well in multi-node setup as they do on a single instance. There are also some limitations in what can be safely replicated in multi-node setting. Application usage goes into detail on how BDR behaves from an application development perspective.

Characteristics affecting BDR performance

By default, BDR keeps one copy of each table on each node in the group, and any changes propagate 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 isn't affected by the number of nodes. Thus, adding nodes increases linearly the total possible SELECT throughput.

If an INSERT, UPDATE, and DELETE (DML) is performed locally, then the changes propagate 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 can handle more TPS than a single node.

Conflict handling has a cost that acts to reduce the throughput. The throughput then depends on how much contention the application displays in practice. Applications with very low contention perform better than a single node. Applications with high contention can perform worse than a single node. These results are consistent with any multi-master technology. They aren't particular to BDR.

Synchronous replilcation options can send changes concurrently to multiple nodes so that the replication lag is minimized. Adding more nodes means using more CPU for replication, so peak TPS reduces slightly as each node is added.

If the workload tries to use all CPU resources, then this resource constrains replication, which can then affect the replication lag.

In summary, adding more master nodes to a BDR group doesn't result in significant write throughput increase when most tables are replicated because all the writes will be replayed on all nodes. Because BDR writes are in general more effective than writes coming from Postgres clients by way of SQL, some performance increase can be achieved. Read throughput generally scales linearly with the number of nodes.

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 isn't recommended and might not be supported.

Refer to the `TPAexec Architecture User Manual` for your architecture.

Log messages and documentation are currently available only in English.

Clocks and timezones

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

Synchronize server clocks using NTP or other solutions.

Clock synchronization isn't critical to performance, as it is with some other solutions. Clock skew can impact origin conflict detection, although BDR provides controls to report and manage any skew that exists. BDR also provides row-version conflict detection, as described in Conflict detection.

Limits

BDR can run hundreds of nodes on good-enough hardware and network. However, for mesh-based deployments, we generally don't recommend running more than 32 nodes in one cluster. Each master node can be protected by multiple physical or logical standby nodes. There's no specific limit on the number of standby nodes, but typical usage is to have 2–3 standbys per master. Standby nodes don't add connections to the mesh network, so they aren't included in the 32-node recommendation.

BDR currently has a 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 you use only one BDR database per PostgreSQL instance.

The minimum recommended number of nodes in a group is three to provide fault tolerance for BDR's consensus mechanism. With just two nodes, consensus would fail if one of the nodes was unresponsive. Consensus is required for some BDR operations such as distributed sequence generation. For more information about the consensus mechanism used by EDB Postgres Distributed, see Architectural details.

6.1 Application use

Learn about the application from a user perspective.

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 REPLICA 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 REPLICA 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 SERIALIAZABLE 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 Conflicts.

Sequences need special handling, described in 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 does'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.

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.

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 DDL replication.

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

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.

Replicating between nodes with differences

By default, DDL is automatically sent to all nodes. You can control this manually, as described in 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 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 DDL replication to specify how to alter a role password only on a subset of nodes or locally.

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 EDB Postgres Distributed 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.

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.

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.

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 EDB Postgres Distributed.

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 n`bdr-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'tt 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.

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.

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 EDB Postgres Distributed 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
```

6.2 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.

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 * 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.

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.

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

  See PostgreSQL documentation for more details.

  BDR can'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 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 — Don't use global locking for DDL operations.
  - on — Use global locking for all DDL operations.
  - dml — 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` — Log a warning about this fact. The warnings are logged once per minute (the default) at the maximum to prevent flooding the server log.
  - `WAIT` — Wait 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 other subscribers not listed in there to lag behind the provider if the required number of listed nodes are not keeping up. Monitoring is thus essential.

Another use case where `bdr.standby_slot_names` is useful is when using a subscriber-only node, to ensure that it does not move ahead of any of the regular BDR nodes. This can best be achieved by listing the logical slots of all regular BDR peer nodes in combination with setting `bdr.standby_slots_min_confirmed` to at least one.

`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 `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 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 EDB Postgres Distributed 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 >= `log_min_messages` can produce a very large volume of log output, so don't enabled 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.

6.3 Node management

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

At present, each node can be a member of just one node group. (This might be extended in later releases.) Each node can subscribe to one or more replication sets to give fine-grained control over replication.

A BDR group might also contain zero or more subgroups, allowing you to create a variety of different architectures.

Creating and joining a BDR group

For BDR, every node must connect to every other node. To make configuration easy, when a new node joins, it 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 data source name (DSN), 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, like `postgresql://myhost:5432/mydb`.

The SQL function `bdr.create_node_group()` creates 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 for other nodes to 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 only the target database. The earlier behavior was to back up the entire database cluster. With this utility, the activity completes faster and also uses less space because it excludes unwanted databases. If you specify only the target database, 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 subscribes to an upstream peer, before replication can begin the system must copy the existing data from the peer nodes to the local node. This copy must be carefully

coordinated so that the local and remote data starts out identical. It's not enough to use `pg_dump` yourself. The BDR extension provides built-in facilities for making this initial copy.

During the join process, the BDR extension synchronizes 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, restart the join process 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 be empty except for the BDR extension. However, it's important that all required database users and roles are created.

Optionally, you can skip the schema synchronization using the `synchronize_structure` parameter of the `bdr.join_node_group()` function. In this case, the schema must already exist on the newly joining node.

We recommend that you select the source node that has the best connection (the closest) as the source node for joining. Doing so lowers the time needed for the join to finish.

Coordinate the join procedure using the Raft consensus algorithm, which requires most existing nodes to be online and reachable.

The logical join procedure (which uses the `bdr.join_node_group()` function) performs data sync doing `COPY` operations and uses 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 is 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.

The join process uses only one node as the source, so it 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 is 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) can conflict with changes from other nodes. In this case, the newly joined node can be resolved differently to other nodes, causing a divergence. As a result, we recommend not running a node join when significant replication lag exists between nodes. 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` can 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 might fail if there's concurrent DDL activity on the source node. Retrying the join works in that case.

Joining a heterogeneous cluster

BDR 4.0 node can join a EDB Postgres Distributed cluster running 3.7.x at a specific minimum maintenance release (such as 3.7.6) or a mix of 3.7 and 4.0 nodes. This procedure is useful when you want to upgrade not just the BDR major version but also the underlying PostgreSQL major version. You can achieve this by joining a 3.7 node running on PostgreSQL 12 or 13 to a EDB Postgres Distributed cluster running 3.6.x on PostgreSQL 11. The new node can also run 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 we recommend that you quickly bring the cluster into a homogenous state by parting the older nodes once enough new nodes join the cluster. Don't run any DDLs that might not be available on the older versions and vice versa.

A node joining with a different major PostgreSQL release can't use physical backup taken with `bdr_init_physical`,

and the node must join using the logical join method. This is necessary because the major PostgreSQL releases aren't on-disk compatible with each other.

When a 3.7 node joins the cluster using a 3.6 node as a source, certain configurations, such as conflict resolution, aren't copied from the source node. The node must be configured after it joins 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. The DSN must work as the connection string from the client connecting to the node in which it's specified. An example of such a set of parameters using a client certificate is:

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

With this setup, the files `bdr_client.crt`, `bdr_client.key`, and `root.crt` must be present in the data directory on each node, with the appropriate permissions. For `verify-full` mode, the server's SSL certificate is checked to ensure that it's 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 doesn't 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 requires 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 might be to use `SCRAM-SHA-256` passwords instead of client certificates and not verify the server identity as long as the certificate is properly signed. Here the DSN parameters might be:

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

The corresponding `pg_hba.conf` lines are:

```
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 needs a `.pgpass` file containing the correct password.

Witness nodes

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

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

Logical standby nodes

BDR allows you to create a *logical standby node*, also known as an offload node, a read-only node, receive-only node, or

logical-read replicas. A master node can have zero, one, or more logical standby nodes.

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

Later, if you want, 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 or 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. If there are multiple logical standby nodes, the other nodes can't follow the new master, so the effectiveness of this technique is limited to one logical standby.

In case a new standby is created from an existing BDR node, the needed replication slots for operation aren't 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 might require a full 16 MB before slots are synced or created on the streaming replica. If a failover or switchover occurs during this interval, the streaming standby can't be promoted to replace its BDR node, as the group slot and other dependent slots don't 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 EDB Postgres Distributed cluster by performing WAL switches and requesting all BDR peer nodes to replay their progress updates. This causes the group slot to move ahead in a short time span. 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's important to ensure that the slot's sync-up completes on the standby before promoting it. You can run the following query on the standby in the target database to monitor and ensure that the slots 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;
```

You can also 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 causes the group slot to move ahead in a short time and also hastens the slot sync-up activity on the standby. You can run the following queries 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 on the standby to check that these queries do help in faster slot sync-up on that standby.

Logical standby nodes can be protected using physical standby nodes, if desired, so Master->LogicalStandby->PhysicalStandby. You can't cascade from LogicalStandby to LogicalStandby.

A logical standby does allow write transactions, so the restrictions of a physical standby don't apply. You can use this 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 aren't sent to other nodes. All

transactions that commit after promotion are sent onwards. If you perform writes to a logical standby, take care to quiesce the database before promotion.

You might make DDL changes to logical standby nodes but they aren't replicated and they don't attempt to take global DDL locks. BDR functions that act similarly to DDL also aren't replicated. See DDL replication. If you made incompatible DDL changes to a logical standby, then the database is a *divergent node*. Promotion of a divergent node currently results in replication failing. As a result, plan to either ensure that a logical standby node is kept free of divergent changes if you intend to use it as a standby, or ensure that divergent nodes are never promoted.

Physical standby nodes

BDR also enables you to create traditional physical standby failover nodes. These are commonly intended to directly replace a BDR node in the cluster after a short promotion procedure. As with any standard Postgres cluster, a node can 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 are 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 don't include pglogical
    - `hot_standby = on`
    - `hot_standby_feedback = on`
- The primary has:
  - `postgresql.conf`:
    - `bdr.standby_slot_names` specifies 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, you need to address some additional concerns.

Once established, the standby requires enough 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 live and can survive a failover only if they report a nonzero `confirmed_flush_lsn` as reported by `pg_replication_slots`.

As a consequence, check physical standby nodes in newly initialized BDR clusters with low amounts of write activity 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 needed 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. This is because the group replication happens only when the physical standby nodes are up to date.

For these reasons, we generally recommend to use either logical standby nodes or a subscribe-only group instead of physical standby nodes. They both have better operational characteristics in comparison.

You can 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:

- Assume control of the same IP address or hostname as the primary.
- Inform the EDB Postgres Distributed cluster of the change in address by executing the bdr.alter_node_interface function on all other BDR nodes.

Once this is done, the other BDR nodes reestablish communication with the newly promoted standby -> primary node. Since replication slots are synchronized only periodically, this new primary might reflect a lower LSN than expected by the existing BDR nodes. If this is the case, BDR fast forwards 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's important to set it in a EDB Postgres Distributed 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 nonparticipating node member that, regardless, affects the state of the group slot.

Since the standby doesn't directly communicate with the other BDR nodes, the `standby_slot_names` parameter informs BDR to consider named slots as needed constraints on the group slot as well. When set, the group slot is held if the standby shows lag, even if the group slot is normally advanced.

As with any physical replica, this type of standby can 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's possible to mix physical standby and other BDR nodes in `synchronous_standby_names`. CAMO and Eager All-Node Replication use different synchronization mechanisms and don't work with synchronous replication. Make sure `synchronous_standby_names` doesn't include the CAMO partner (if CAMO is used) or any BDR node at all (if Eager All-Node Replication is used). Instead use only non-BDR nodes, for example, a physical standby.

Subgroups

A group can also contain zero or more subgroups. Each subgroup can be allocated to a specific purpose in the top-level parent group. The node_group_type specifies the type when the subgroup is created.

Subscriber-only groups

As the name suggests, this type of node subscribes only to replication changes from other nodes in the cluster. However, 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 to the cluster. They can receive replication changes from all other nodes in the cluster and hence aren't affected 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 doesn't have

Raft voting rights and hence can't become a Raft leader or participate in the leader election. Also, while it receives replicated DDLs, it doesn't participate in DDL or DML lock acquisition. In other words, a currently down `subscriber-only` node doesn't stop a DML lock from being acquired.

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

To make use of `subscriber-only` nodes, first create a BDR group of type `subscriber-only`. Make it a subgroup of the group from which the member nodes receive the replication changes. Once you create the subgroup, all nodes that intend to become `subscriber-only` nodes must join the subgroup. You can create more than one subgroup of `subscriber-only` type, and they can have different parent groups.

Once a node successfully joins the `subscriber-only` subgroup, it becomes a `subscriber-only` node and starts receiving replication changes for the parent group. Any changes made directly on the `subscriber-only` node aren't 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 already received and applied replication changes from the parted node that no other node in the cluster currently has, then that causes inconsistency between the nodes.

For now, you can solve this by setting `bdr.standby_slot_names` and `bdr.standby_slots_min_confirmed` so that there is always a fully active BDR node that is ahead of the `subscriber-only` nodes.

This might be improved in a future release. We might 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.

Decoding worker

BDR4 provides an option to enable a decoding worker process that performs decoding once, no matter how many nodes are 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. You can use `bdr.alter_node_group_config()` to enable or disable the decoding worker for a BDR group.

When the decoding worker is enabled, BDR stores logical change record (LCR) files to allow buffering of changes between decoding and when all subscribing nodes received data. LCR files are stored under the `pg_logical` directory in each local node's data directory. The number and size of the LCR files varies as replication lag increases, so this also needs monitoring. The LCRs that aren't 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 use of LCR per node. By default these are `false`. For production and use of LCRs, enable the decoding worker for the BDR group and set these GUCs to to `true` on each of the nodes in the BDR group.

- `bdr.enable_wal_decoder` — When turned `false`, all WAL senders using LCRs restart 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's running. A logical standby is sent changes from all the nodes in the BDR group through a single source. Hence a WAL sender serving a logical standby can't use LCRs right now.

A subscriber-only node receives changes from respective nodes directly. 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 a decoding worker isn't enabled. In the future, it might be possible to remove WAL corresponding the LCRs, if they aren't otherwise required.

For reference, the first 24 characters of an LCR file name are similar to those in a WAL file name. The first 8 characters of the name are all '0' right now. In the future, they are expected to represent the TimeLineId similar to the first 8 characters of a WAL segment file name. 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, logical changes are reordered according to the commit order of the transactions they belong to. Hence their placement in the LCR segments doesn't match the placement of corresponding WAL in the WAL segments.

The set of last 16 characters represents the subsegment number in an LCR segment. Each LCR file corresponds to a subsegment. 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 1 GB.

Node restart and down node recovery

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

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

If the amount of write activity on each node isn't uniform, the catchup period from nodes with more data can take significantly longer than other nodes. Eventually, the slot state changes to `bdr.node_slots.state = streaming`.

Nodes that are offline for longer periods, such as hours or days, can begin to cause resource issues for various reasons. Don't plan on extended outages without understanding the following issues.

Each node retains change information (using one replication slot for each peer node) so it can later replay changes to a

temporarily unreachable node. If a peer node remains offline indefinitely, this accumulated change information eventually causes the node to run out of storage space for PostgreSQL transaction logs (*WAL* in `pg_wal`), and likely causes 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, it might 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 must monitor for node outages (see monitoring) and make sure nodes have enough free disk space. If the workload is predictable, you might be able 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.

Don't manually remove replication slots created by BDR. If you do, the cluster becomes damaged and the node that was using the slot must be parted from the cluster, as described in Replication slots created by BDR.

While a node is offline, the other nodes might not yet have received the same set of data from the offline node, so this might appear as a slight divergence across nodes. The parting process corrects this imbalance across nodes. (Later versions might do this earlier.)

Replication slots created by BDR

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

- 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

!!! Warning Don't drop those slots. BDR creates and manages them and drops them when or if necessary.

On the other hand, you can create or drop replication slots required by software like Barman or logical replication using the appropriate commands for the software without any effect on BDR. Don't start slot names used by other software with the prefix `bdr_`.

For example, in a cluster composed of the three 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 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 that was down during join starts replicating again.
- Part nodes from the cluster consistently, even if some nodes haven't 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 fast forwarded only periodically in response to Raft progress messages from other nodes.

!!! Warning Don't drop the group slot. Although usually inactive, it's still vital to the proper operation of the EDB Postgres Distributed cluster. If you drop it, then some or all of the features can stop working or have incorrect outcomes.

Hashing long identifiers

The name of a replication slot—like any other PostgreSQL identifier—can't 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. Shortening an identifier is carried out by replacing the final section of the string with a hash of the string itself.

For example, consider a cluster that replicates a database named `db20xxxxxxxxxxxxxxx` (20 bytes long) using a BDR group named `group20xxxxxxxxxxxxx` (20 bytes long). The logical replication slot associated to node `a30xxxxxxxxxxxxxxxxxxxxxxxxxxx` (30 bytes long) is called since `3597186`, `be9cbd0`, and `7f304a2` are respectively the hashes of `db20xxxxxxxxxxxxxxx`, `group20xxxxxxxxxxxxx`, and `a30xxxxxxxxxxxxxxxxxxxxxxxxxx`.

```
bdr_db20xxxx3597186_group20xbe9cbd0_a30xxxxxxxxxxxx7f304a2
```

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're removing a node permanently. If you permanently shut down a node and don't tell the other nodes, then performance suffers and eventually the whole system stops 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. You can call the `bdr.part_node()` function from any active node in the BDR group, including the node that you're removing.

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 manages a vote between nodes to see which node has the most recent data from the parting node. Then all remaining nodes 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 might be added again under the same name as a parted node. In rare cases, you might want to clear all metadata of a parted node by using the function `bdr.drop_node()`.

Uninstalling BDR

Dropping the BDR extension removes all the BDR objects in a node, including metadata tables. You can do this with the following command:

```
DROP EXTENSION bdr;
```

If the database depends on some BDR-specific objects, then you can't drop the BDR extension. Examples include:

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

Remove those dependencies before dropping the BDR extension. For example, drop the dependent objects, alter the column type to a non-BDR equivalent, or change the sequence type back to `local`.

!!! Warning You can drop the BDR extension only if the node was successfully parted from its BDR node group or if it's the last node in the group. Dropping BDR metadata breaks replication to and 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. You can solve the problem by disconnecting the session and then reconnecting the client or by restarting the instance.

There's also a `bdr.drop_node()` function. Use this function only in emergencies, such as if there's a problem with parting.

Listing BDR topology

Listing BDR groups

The following simple query lists all the BDR node groups of which the current node is a member. It currently returns only one row.

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

You can display the configuration of each node group 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

You can extract the list of all nodes in a given node group (such as `mygroup`) 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';
```

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

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

Node management interfaces

You can add and remove nodes 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 lowercase letters, numbers, hyphens, and underscores.
- `local_dsn` — Connection string to the node.

Notes

This function 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 reports an error if run again.

This function is a transactional function. You can roll it back and the changes made by it are visible to the current transaction.

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

bdr.drop_node

Drops a node.

!!! Warning This function isn't intended for regular use. Execute it only if instructed by 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. Only Technical Support uses a forced node drop in case of emergencies related to parting.

Notes

Before you run this, part the node using `bdr.part_node()`.

This function removes metadata for a given node from the local database. The node can be the local node, in which case all the node metadata are removed, including information about remote nodes. 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 because each node has a unique sequence number assigned to it, for use by snowflakeid and timeshard sequences. PARTED nodes aren't automatically cleaned up. If this becomes a problem, you can use this function 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 only lowercase letters, numbers, and underscores.
- `parent_group_name` — The name of the parent group for the subgroup.
- `join_node_group` — This parameter helps a node to decide whether 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 doesn't want to join. This can be `false` only if you specify `parent_group_name`.
- `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 receive changes only from the fully joined nodes in the cluster, but they never send replication changes to other nodes. See [Subscriber-only nodes](#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 other values are reserved for future use.` NULL` implies a normal general-purpose node group is created.

Notes

This function passes a request to the local consensus worker that's running for the local node.

The function isn't transactional. The creation of the group is a background process, so once the function finishes, you can't roll back the changes. Also, the changes might not be immediately visible to the current transaction. You can call `bdr.wait_for_join_completion` to wait until they are.

The group creation doesn't hold any locks.

bdr.alter_node_group_config

This function changes the configuration parameters of an existing BDR group. Options with NULL value (default for all of them) aren't 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,
                                        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 backward compatibility.
- `update_to_insert` — Reserved for backward compatibility.

```
  versions of BDR. Use `bdr.alter_node_set_conflict_resolver` instead.
```

- `ignore_redundant_updates` — Reserved for backward compatibility.
- `check_full_tuple` — Reserved for backward compatibility.
- `apply_delay` — Reserved for backward compatibility.
- `check_constraints` — Whether the apply process checks the constraints when writing replicated data. This option is deprecated and will 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`) is used. Valid values are either `-1` or a positive integer.
- `enable_wal_decoder` — Enables/disables the decoding worker process. You can't enable the decoding worker process 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're 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 tries to intelligently pick between `file` and `writer`, depending on the transaction property and available resources. You can't enable `streaming_mode` 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 applies only to the top-level node group. You can use individual rules for different origin groups of the same commit scope. See Origin groups for more details.

Notes

This function passes a request to the group consensus mechanism to change the defaults. The changes made are replicated globally using the consensus mechanism.

The function isn't transactional. The request is processed in the background so you can't roll back the function call. Also, the changes might not be immediately visible to the current transaction.

This function doesn't hold any locks.

!!! Warning When you use this function to change the `apply_delay` value, the change doesn't apply to nodes that are already members of the group. This restriction has little consequence on production use because this value normally isn't 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 an existing (source) node in the BDR group you want to add the local node to.
- `node_group_name` — Optional name of the BDR group. Defaults to NULL, which tries to detect the group name from information present on the source node.
- `pause_in_standby` — Optionally tells the join process to join only 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 the kind of structure (schema) synchronization to do during the join. Valid options are `all`, which synchronizes the complete database structure, and `none`, which doesn't synchronize any structure. However, it still synchronizes data.

If `wait_for_completion` is specified as `false`, this is an asynchronous call that returns as soon as the joining procedure starts. You can see progress of the join in logs and the `bdr.state_journal_details` information view or by calling the `bdr.wait_for_join_completion()` function after `bdr.join_node_group()` returns.

Notes

This function passes a request to the group consensus mechanism by way of the node that the `join_target_dsn` connection string points to. The changes made are replicated globally by the consensus mechanism.

The function isn't transactional. The joining process happens in the background and you can't roll it back. The changes are visible only to the local transaction if `wait_for_completion` was set to `true` or by calling `bdr.wait_for_join_completion` later.

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

Node join doesn't 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 passes a request to the group consensus mechanism to change the defaults. The changes made are replicated globally by the consensus mechanism.

The function isn't transactional. The promotion process happens in the background, and you can't roll it back. The changes are visible only 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 doesn't block other `bdr.promote_node` calls but prevents 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 doesn't remove data from the node.

You can call the function from any active node in the BDR group, including the node that you're removing. However, once the node is parted, it can't part other nodes in the cluster.

!!! Note If you're parting the local node, you must set `wait_for_completion` to `false`. Otherwise, it reports an error.

!!! Warning This action is permanent. If you want 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 doesn't return until the node is fully parted from the cluster. Otherwise the function starts the parting procedure and returns immediately without waiting. Always set to `false` when executing on the local node or when using `force`.
- `force` — Forces removal of the node on the local node. This sets the node state locally if consensus can't be reached or if the node parting process is stuck.

!!! Warning Using `force = true` can leave the BDR group in an inconsistent state. Use it only to recover from failures in which you can't remove the node any other way.

Notes

This function passes a request to the group consensus mechanism to part the given node. The changes made are replicated globally by the consensus mechanism. The parting process happens in the background, and you can't roll it back. The changes made by the parting process are visible only to the local transaction if `wait_for_completion` was set to `true`.

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

The parting process doesn't 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

Run this function and make the changes only on the local node. This means that you normally execute it on every node in the BDR group, including the node that is being changed.

This function is transactional. You can roll it 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. This is 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 are enabled.
- `immediate` — This currently has no effect.

Notes

This function isn't replicated and affects only local node subscriptions (either a specific node or all nodes).

This function is transactional. You can roll it back, and the current transaction can see any catalog changes. The subscription workers are 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. This is 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 are disabled.
- `immediate` — Used to force the action immediately, stopping all the workers associated with the disabled subscription. When this option is `true`, you can't run this function inside of the transaction block.
- `fast` — This argument 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 isn't replicated and affects only local node subscriptions (either a specific subscription or all subscriptions).

This function is transactional. You can roll it back, and the current transaction can see any catalog changes. However, the timing of the subscription worker stopping depends on the value of `immediate`. If set to `true`, the workers receive the stop without waiting for the `COMMIT`. If the `fast` argument is set to `true`, the interruption of the workers doesn't wait for current work to finish.

Node-management commands

BDR also provides a command-line utility for adding nodes to the BDR group using 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 that's added to PostgreSQL's bin directory.

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

If the specified data directory isn't empty, this is used as the base for the new node. If the data directory is already active as a physical standby node, you need to stop the standby before running `bdr_init_physical`, which manages Postgres. Initially it waits for catchup and then promotes to a master node before joining the BDR group. The `--standby` option, if used, turns 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 drops all PostgreSQL-native logical replication subscriptions from the database (or disables 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 use for the new node. It can be either an empty or nonexistent directory or a directory populated using the `pg_basebackup -X stream` command (required).

- `-l, --log-file=FILE` — Use FILE for logging. The 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 are 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 with an empty/nonexistent data directory (-D option). This is a feature of EDB Postgres Extended Server only.
- `-S` — Instead of dropping logical replication subscriptions, 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 don't 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 base backup or an existing base backup is being promoted to a new BDR node. Thus the `--replication-sets` option affects only the data published and subscribed to after the node joins the BDR node group. This behavior is different from the way replication sets are used in a logical join, as when using `bdr.join_node_group()`.

The operator can truncate unwanted tables after the join completes. Refer to the `bdr.tables` catalog to determine replication set membership and identify tables that aren't members of any subscribed-to replication set. We strongly recommend that you truncate the tables rather than drop them, because:

- DDL replication sets aren't necessarily the same as row (DML) replication sets, so you might inadvertently drop the table on other nodes.
- If you later want to add the table to a replication set and you dropped it on some subset of nodes, you need to re-create it only on those nodes without creating DDL conflicts before you can add it to any replication sets.

It's simpler and safer to truncate your nonreplicated tables, leaving them present but empty.

A future version of BDR might automatically omit or remove tables that aren't part of the selected replication sets for a physical join, so your application should not rely on details of the behavior documented here.

6.4 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 DDL statements (mostly long running)
- 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 certain DDL changes easier 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, you must add a DDL replication filter to the replication set. See DDL replication filtering.

BDR is significantly different from standalone PostgreSQL when it comes to DDL replication. Treating it the same is the most common issue with BDR.

The main difference from table replication is that DDL replication doesn't replicate the result of the DDL but the statement itself. This works very well in most cases, although it 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 (not built-in). For example, the DDL statement must execute correctly in the default encoding used on each node.

DDL replication options

The `bdr.ddl_replication` parameter specifies replication behavior.

`bdr.ddl_replication = on` is the default and replicates DDL to the default replication set, which by default means all nodes. Nondefault replication sets don't 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, for example, 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
```

While we don't recommend it, you can 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 makes BDR skip both the global locking and the replication of executed DDL commands. 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 executes concurrently.

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

Executing DDL on BDR systems

A BDR group isn't the same as a standalone PostgreSQL server. It's based on asynchronous multi-master replication without central locking and without a transaction coordinator. This has important implications when executing DDL.

DDL that executes in parallel continues to do so with BDR. DDL execution respects the parameters that affect parallel operation on each node as it executes, so you might notice differences in the settings between nodes.

Prevent the execution of conflicting DDL, otherwise DDL replication causes errors and the replication stops.

BDR offers three 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 isn't the default, but we recommend that you use this setting if you can control where DDL is executed from. Doing so ensures 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 execute from any node concurrently and you want to ensure correctness.

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

These options can be set only by the bdr_superuser, by the superuser, or in the `config file`.

When using the `bdr.replicate_ddl_command`, you can set this parameter directly with the third argument, using the specified `bdr.ddl_locking` setting only for the DDL commands passed to that function.

DDL locking details

Two kinds of locks enforce correctness of replicated DDL with BDR.

The first kind is known as a global DDL lock and is used only 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 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 tables.

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 by the regular replication stream, and the nodes respond by the replication stream as well. So it's important that nodes (or at least a majority of the nodes) run without much replication delay. Otherwise it might take a 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 are executed in the same sequence on all other nodes.

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 ran 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 might take a long time to acquire the lock. Hence it's preferable to run DDLs from a single node or the nodes that 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. These failures can occur when you 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 waiting 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.

The global DML lock holds an EXCLUSIVE LOCK on the table on each node, so it blocks DML, other DDL, VACUUM, and index commands against that table while it runs. This is true even if the global DML lock is held for a command that normally doesn't take an EXCLUSIVE LOCK or higher.

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

If a DDL statement isn't replicated, no global locks are 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.

Some BDR functions make DDL changes. For those functions, DDL locking behavior applies. This is noted in the docs for each function.

Thus, `ddl_locking = dml` is safe only when you can guarantee that no conflicting DDL is executed from other nodes. With this setting, the statements that require only the global DDL lock don't use the global locking at all.

`ddl_locking = off` is safe only when you can guarantee that there are no conflicting DDL and no conflicting DML operations on the database objects DDL executes on. If you turn locking off and then experience difficulties, you might lose in-flight changes to data. The user application team needs to resolve any issues caused.

In some cases, concurrently executing DDL can properly be serialized. If these serialization failures occur, the DDL might reexecute.

DDL replication isn't active on logical standby nodes until they are promoted.

Some BDR management functions act like DDL, meaning that they attempt to take global locks, and their actions are 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 the current transaction doesn't required global DML lock.

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

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, make transactions performing DDL short, don't combine them with lots of row changes, and avoid long running foreign key or other constraint rechecks.
- For `ALTER TABLE`, use `ADD CONSTRAINT NOT VALID` followed by another transaction with `VALIDATE CONSTRAINT` rather than using `ADD CONSTRAINT` alone. `VALIDATE CONSTRAINT` waits until replayed on all nodes, which gives a noticeable delay to receive confirmations.
- When indexing, use the `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. You can still do this using a single SQL statement, as shown in the following example. Global locking rules still apply, so be careful not to lock yourself out with this type of usage, which is more of a workaround.

```
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 with `CREATE INDEX CONCURRENTLY`, noting that DDL replication must be disabled for the whole session because `CREATE INDEX CONCURRENTLY` is a multi-transaction command. Avoid `CREATE INDEX` 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. Avoid using `REINDEX` because of the AccessExclusiveLocks it holds.

Instead, use `REINDEX CONCURRENTLY` (or `reindexdb --concurrently`), which is available in PG12+ or 2QPG11+.

You can disable DDL replication 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 take the DDL lock only once. This might not be desirable if the table-level locks interfere with normal operations.

If DDL is holding up the system for too long, you can safely cancel the DDL on the originating node with `Control-C` in psql or with `pg_cancel_backend()`. You can't cancel a DDL lock from any other node.

You can control how long the global lock takes with optional global locking timeout settings. `bdr.global_lock_timeout` limits how long the wait for acquiring the global lock can take before it's canceled. `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. You can disable all of these timeouts by setting their values to zero.

Once the DDL operation has committed on the originating node, you can't cancel or abort it. The BDR group must wait for it to apply successfully on other nodes that confirmed the global lock and for them to acknowledge replay. For this reason, keep DDL transactions short and fast.

Handling DDL with down nodes

If the node initiating the global DDL lock goes down after it acquired the global lock (either DDL or DML), the lock stays active. The global locks don't time out, even if timeouts were set. In case the node comes back up, it releases all the global locks that it holds.

If it stays down for a long time (or indefinitely), remove the node from the BDR group to release the global locks. This is 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 confirmed the global lock but before the command acquiring it executed, the execution of that command requesting the lock continues as if the node were up.

As mentioned earlier, the global DDL lock requires only a majority of the nodes to respond, and so it works if part of the cluster is down, as long as a majority is running and reachable. But the DML lock can't be acquired unless the whole cluster is available.

With global DDL or global DML lock, if another node goes down, the command continues normally, and the lock is released.

Statement-specific DDL replication concerns

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

BDR prevents some DDL statements from running when it's active on a database. This protects the consistency of the system by disallowing statements that can't be replicated correctly or for which replication isn't yet supported.

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

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

Any DDL that references or relies on a temporary object can't be replicated by BDR and throws an error if executed with DDL replication enabled.

BDR DDL command handling matrix

The following table describes the utility or DDL commands that are allowed, the ones that are replicated, and the type of global lock they take when they're replicated.

For some more complex statements like `ALTER TABLE`, these can differ depending on the subcommands executed. Every such command has detailed explanation under the following table.

Command	Allowed	Replicated	Lock
ALTER AGGREGATE	Y	Y	DDL
ALTER CAST	Y	Y	DDL
ALTER COLLATION	Y	Y	DDL
ALTER CONVERSION	Y	Y	DDL
ALTER DATABASE	Y	N	N
ALTER DATABASE LINK	Y	Y	DDL
ALTER DEFAULT PRIVILEGES	Y	Y	DDL
ALTER DIRECTORY	Y	Y	DDL
ALTER DOMAIN	Y	Y	DDL

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

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

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

Command	Allowed	Replicated	Lock
DROP FOREIGN TABLE	Y	Y	DDL
DROP FUNCTION	Y	Y	DDL
DROP INDEX	Y	Y	DDL
DROP LANGUAGE	Y	Y	DDL
DROP MATERIALIZED VIEW	Y	N	N
DROP OPERATOR	Y	Y	DDL
DROP OPERATOR CLASS	Y	Y	DDL
DROP OPERATOR FAMILY	Y	Y	DDL
DROP OWNED	Y	Y	DDL
DROP PACKAGE	Y	Y	DDL
DROP PACKAGE BODY	Y	Y	DDL
DROP POLICY	Y	Y	DDL
DROP PROCEDURE	Y	Y	DDL
DROP PROFILE	Y	Y	DDL
DROP PUBLICATION	Y	Y	DDL
DROP QUEUE	Y	Y	DDL
DROP QUEUE TABLE	Y	Y	DDL
DROP REDACTION POLICY	Y	Y	DDL
DROP RESOURCE GROUP	Y	N	N
DROP ROLE	Y	Y	DDL
DROP ROUTINE	Y	Y	DDL
DROP RULE	Y	Y	DDL
DROP SCHEMA	Y	Y	DDL
DROP SEQUENCE	Y	Y	DDL
DROP SERVER	Y	Y	DDL
DROP STATISTICS	Y	Y	DDL
DROP SUBSCRIPTION	Y	Y	DDL
DROP SYNONYM	Y	Y	DDL
DROP TABLE	Y	Y	DML
DROP TABLESPACE	Y	N	N
DROP TEXT SEARCH CONFIGURATION	Y	Y	DDL
DROP TEXT SEARCH DICTIONARY	Y	Y	DDL
DROP TEXT SEARCH PARSER	Y	Y	DDL
DROP TEXT SEARCH TEMPLATE	Y	Y	DDL
DROP TRANSFORM	Y	Y	DDL
DROP TRIGGER	Y	Y	DDL
DROP TYPE	Y	Y	DDL
DROP TYPE BODY	Y	Y	DDL
DROP USER MAPPING	Y	Y	DDL
DROP VIEW	Y	Y	DDL
EXECUTE	Y	N	N
EXPLAIN	Y	Details	Details
FETCH	Y	N	N

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

ALTER SEQUENCE

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

`ALTER SEQUENCE ... RENAME` isn't supported on galloc sequences (only). `ALTER SEQUENCE ... SET SCHEMA` isn't supported on galloc sequences (only).

ALTER TABLE

Generally, `ALTER TABLE` commands are allowed. However, several subcommands aren't supported.

ALTER TABLE disallowed commands

Some variants of `ALTER TABLE` currently aren't allowed on a BDR node:

- `ADD COLUMN ... DEFAULT (non-immutable expression)` — This is not allowed because it currently results in different data on different nodes. See Adding a column for a suggested workaround.
- `ADD CONSTRAINT ... EXCLUDE` — Exclusion constraints aren't supported for now. Exclusion constraints don't make much sense in an asynchronous system and lead to changes that can't be replayed.
- `ALTER TABLE ... SET WITH[OUT] OIDS` — Isn't supported for the same reasons as in `CREATE TABLE`.
- `ALTER COLUMN ... SET STORAGE external` — Is rejected if the column is one of the columns of the replica identity for the table.
- `RENAME` — Can't rename an Autopartitioned table.
- `SET SCHEMA` — Can't set the schema of an Autopartitioned table.
- `ALTER COLUMN ... TYPE` — Changing a column's type isn't supported if the command causes the whole table to be rewritten, which occurs when the change isn't binary coercible. Binary coercible changes might be allowed only 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)` isn't binary coercible and therefore normally disallowed. Nonreplicated `ALTER COLUMN ... TYPE`, can be allowed if the column is automatically castable to the new type (it doesn't contain the `USING` clause). An example follows. Table rewrites 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 workaround.
- `ALTER TABLE ... ADD FOREIGN KEY` — Isn't supported if current user doesn't have permission to read the referenced table or if the referenced table has RLS restrictions enabled that the current user can't 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 on the time zone of the session and so isn't immutable.

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

Starting in BDR 3.7.4, you can add certain types of constraints, such as `CHECK` and `FOREIGN KEY` constraints, without taking a DML lock. But this requires a two-step process of first creating a `NOT VALID` constraint and then validating the constraint in a separate transaction with the `ALTER TABLE ... VALIDATE CONSTRAINT` command. See Adding a CONSTRAINT for more details.

ALTER TABLE locking

The following variants of `ALTER TABLE` take only 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 doesn't 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` still takes 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 doesn't cause a table rewrite. It executes 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 command isn't possible because the change from `VARCHAR(128)` to `VARCHAR(20)` isn't binary coercible.

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

For workarounds, see Restricted DDL workarounds.

It's useful to provide context for different types of `ALTER TABLE ... ALTER COLUMN TYPE` (ATCT) operations that are possible in general and in nonreplicated environments.

Some ATCT operations update only the metadata of the underlying column type and don't 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);
```

You can also change the column type to `VARCHAR` or `TEXT` data types 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 leads 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 nontext types, consider col3 above with type INTEGER. An ATCT operation that tries to convert to SMALLINT or BIGINT fails 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 of these failing cases, there's an automatic assignment cast from the current types to the target types. However, there's no binary coercibility, which ends up causing a rewrite of the underlying table data.

In such cases, in controlled DBA environments, you can change the type of a column to an automatically castable one by adopting a rolling upgrade for the type of this column in a nonreplicated environment on all the nodes, one by one. Suppose the DDL isn't replicated and the change of the column type is to an automatically castable one. You can then allow the rewrite locally on the node performing the alter, along with concurrent activity on other nodes on this same table. You can then repeat this nonreplicated ATCT operation on all the nodes one by one to bring about the desired change of the column type across the entire EDB Postgres Distributed cluster. Because this involves a rewrite, the

activity still takes the DML lock for a brief period and thus requires that the whole cluster is available. With these specifics in place, you can carry out the rolling upgrade of the nonreplicated alter activity like this:

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

Due to automatic assignment casts being available for many data types, this local nonreplicated ATCT operation supports a wide variety of conversions. Also, ATCT operations that use a `USING` clause are likely to fail because of the lack of automatic assignment casts. This example shows a few common conversions with automatic assignment casts:

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

This example isn't an exhaustive list of possibly allowable ATCT operations in a nonreplicated environment. Not all ATCT operations work. The cases where no automatic assignment is possible fail even if you disable DDL replication. So, while conversion from numeric types to text types works in a nonreplicated environment, conversion back from text type to numeric types fails.

```
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 nonreplicated environments support a variety of type conversions, the rewrite can still fail if the underlying table data contains values that you can't assign to the new data type. For example, suppose the current type for a column is `VARCHAR(256)` and you try a nonreplicated ATCT operation to convert it into `VARCHAR(128)`. If there's any existing data in the table that's wider than 128 bytes, then the rewrite operation fails 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 succeeds. However, replication might fail if other nodes that haven't yet performed the nonreplicated rolling data type upgrade introduce new data that is wider than 128 bytes concurrently to this local ATCT operation. This brings replication to a halt in the cluster. So be aware of the data type restrictions and characteristics at the database and application levels while performing these nonreplicated rolling data type upgrade operations. We strongly recommend that you perform and test such ATCT operations in controlled and fully aware DBA environments. These ATCT operations are asymmetric, and backing out certain changes that fail can lead to table rewrites that take a long time.

Also, you can't perform the implicit castable ALTER activity in transaction blocks.

ALTER TYPE

`ALTER TYPE` is replicated, but a global DML lock isn't applied to all tables that use that data type, since PostgreSQL doesn't record those dependencies. See Restricted DDL workarounds.

COMMENT ON

All variants of `COMMENT ON` are allowed, but `COMMENT ON TABLESPACE/DATABASE/LARGE OBJECT` isn't 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` isn't allowed on a BDR node.

CREATE TABLE AS and SELECT INTO

`CREATE TABLE AS` and `SELECT INTO` are allowed only if all subcommands 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` follows replication rules of the analyzed statement.

EXPLAIN ANALYZE Locking

`EXPLAIN ANALYZE` follows locking rules of the analyzed statement.

GRANT and REVOKE

Generally `GRANT` and `REVOKE` statements are supported, however `GRANT/REVOKE ON TABLESPACE/LARGE OBJECT` aren't replicated.

LOCK TABLE

LOCK TABLE isn't replicated, but it might acquire the global DML lock when `bdr.lock_table_locking` is set `on`.

You can also use The `bdr.global_lock_table()` function to explicitly request a global DML lock.

SECURITY LABEL

All variants of `SECURITY LABEL` are allowed, but `SECURITY LABEL ON TABLESPACE/DATABASE/LARGE OBJECT` isn't replicated.

TRUNCATE Replication

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

TRUNCATE Locking

Even though `TRUNCATE` isn't replicated the same way as other DDL, it can 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 don't have to have the same grants, password, and so on, but they must exist.

BDR replicates 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:

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

In general, either:

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

We recommended that you run all role management commands in 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. Otherwise BDR apply stalls with an error until the role is created on the other nodes.

!!! Note BDR doesn't capture and replicate role management statements when they run on a non-BDR-enabled database in 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` is replicated, but a `CREATE USER` run in `postgres` isn't.

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 the desired result that either isn't allowed as a single statement or requires excessive locking.

Adding a CONSTRAINT

You can add `CHECK` and `FOREIGN KEY` constraints without requiring a DML lock. This involves a two-step process.

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

Execute these steps in two different transactions. Both these steps take DDL lock only on the table and hence can be run even when one or more nodes are down. But to validate a constraint, BDR must ensure that:

- All nodes in the cluster see the `ADD CONSTRAINT` command.
- The node validating the constraint applied replication changes from all other nodes prior to creating the NOT VALID constraint on those nodes.

So even though the new mechanism doesn't need all nodes to be up while validating the constraint, it still requires that all nodes applied the `ALTER TABLE .. ADD CONSTRAINT ... NOT VALID` command and made enough progress. BDR waits for a consistent state to be reached before validating the constraint.

The new facility requires the cluster to run with Raft protocol version 24 and beyond. If the Raft protocol isn't yet upgraded, the old mechanism is 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 approach splits schema changes and row changes into separate transactions that BDR can execute and results in consistent data across all nodes in a BDR group.

For best results, batch the update into chunks so that you don't update more than a few tens or hundreds of thousands of rows at once. You can do this using a `PROCEDURE` with embedded transactions.

The last batch of changes must run in a transaction that takes a global DML lock on the table. Otherwise you can miss rows that are inserted concurrently into the table on other nodes.

If required, you can run `ALTER TABLE mytable ALTER COLUMN newcolumn NOT NULL;` 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);
```

You can rewrite this statement to avoid a table rewrite by making the restriction a table constraint rather than a datatype change. The constraint can then be validated in a subsequent command to avoid long locks, if you want.

```
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;
```

If the validation fails, then you can `UPDATE` just the failing rows. You can use this technique 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 helps reduce replication lag if it's a big table. Updating by range of IDs or whatever method you prefer is fine. Alternatively, you can update the whole table in one pass for smaller tables.

`CREATE INDEX ...` any required indexes on the new column. It's 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.

1. `BEGIN` a transaction.
2. `DROP` the trigger you added.
3. `ALTER TABLE` to add any `DEFAULT` required on the column.
4. `DROP` the old column.
5. `ALTER TABLE mytable RENAME COLUMN newcolumn TO oldcolumn`.
6. `COMMIT`.

!!! Note Because you're dropping a column, you might have to re-create views, procedures, and so on that depend on the table. Be careful if you `CASCADE` drop the column, as you must be sure you re-create everything that referred to it.

Changing other types

The `ALTER TYPE` statement is replicated, but affected tables aren't locked.

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

This example ensures that the DDL is written to all nodes before using the new value in DML statements:

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

BDR functions that behave like DDL

The following BDR management functions act like DDL. This means that, if DDL replication is active and DDL filter settings allow it, they attempt to take global locks and their actions are replicate. 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`

6.5 Security and roles

Only superusers can create the BDR extension. However, if you want, you can set up the `pgextwlist` extension and configure it to allow a non-superuser to create a BDR extension.

Configuring and managing BDR doesn't 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 doesn't require that administrators have access to user data.

Arrangements for securing conflicts are discussed in Logging conflicts to a table.

You can monitor conflicts using the `BDR.conflict_history_summary` view.

Catalog tables

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

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

BDR functions and operators

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

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

Granting privileges on catalog objects

Administrators must not grant explicit privileges on catalog objects such as tables, views, and functions. Manage access to those objects by granting one of the roles described 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 don't have the exact same version of BDR (and therefore of the BDR catalog).

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

Role management

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

When a new BDR node joins the BDR group, existing users aren't 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 doesn't know which users access which database and so can't 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 reexecuting that on the newly created node. Other mechanisms are possible, depending on your identity and access management solution (IAM) but aren't automated at this time.

Roles and replication

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

DML changes to tables are replicated as the table-owning user on the target node. We recommend but do not enforce that a table be owned by the same user on each node.

Suppose table A is owned by user X on node1 and owned by user Y on node2. If user Y has higher privileges than user X, this might 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 are replicated without reenforcing 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 isn't what you want, specify a row_filter that avoids replicating all rows. We recommend but don't enforce that the row security policies on all nodes be identical or at least compatible.

The user bdr_superuser controls replication for BDR and can add or remove any table from any replication set. bdr_superuser doesn't need any privileges over individual tables, nor is this recommended. If you need to restrict access to replication set functions, restricted versions of these functions can be implemented as `SECURITY DEFINER` functions and granted to the appropriate users.

Connection role

When allocating a new BDR node, the user supplied in the DSN for the `local_dsn` argument of `bdr.create_node` and the `join_target_dsn` of `bdr.join_node_group` are used frequently to refer to, create, and manage database objects.

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, you can specify a more restricted user 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 from permissions from the owner roles.

Once all nodes are joined, the permissions can 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.

Privilege restrictions

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

`GRANT ALL` still grants both TRIGGER and REFERENCES privileges, so we recommend that you state privileges explicitly. For example, use `GRANT SELECT, INSERT, UPDATE, DELETE, TRUNCATE` instead of `ALL`.

Foreign key privileges

`ALTER TABLE ... ADD FOREIGN KEY` is supported only if the user has SELECT privilege on the referenced table or if the referenced table has RLS restrictions enabled that the current user can't bypass.

Thus, the REFERENCES privilege isn't sufficient to allow creating a foreign key with BDR. Relying solely on the REFERENCES privilege isn't typically useful since it makes the validation check execute using triggers rather than a table scan. It is typically too expensive to use successfully.

Triggers

In PostgreSQL, both the owner of a table and anyone who was granted the TRIGGER privilege can create triggers. Triggers granted by the non-table owner execute as the table owner in BDR, which might 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 the function used by the trigger).
- Users who have TRIGGER privilege on the table can create a trigger only 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 the 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 both in standard PostgreSQL and BDR.

This logic is built on the fact that, in PostgreSQL, the owner of the trigger isn't 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 that aren't owned by the owner of the table and don't use SECURITY DEFINER functions, you can't 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 might assume other search_path settings which then fail when they execute on apply. To ensure this doesn't occur, resolve object references clearly using either the default search_path only (always use fully qualified references to objects, e.g., schema.objectname), or set the search path for a function using `ALTER FUNCTION ... SET search_path = ...` for the functions affected.

BDR default/predefined roles

BDR predefined roles are created when the BDR extension is installed. 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.

The `GRANT ROLE` DDL statement doesn't participate in BDR replication. Thus, 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

Many of these functions have additional privileges required before you can use them. For example, you must be the table owner to successfully execute bdr.alter_sequence_set_kind. These additional rules are described 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's useful to have a user that can see conflicts for all tables, you can optionally grant the role bdr_read_all_conflicts to that user.

Verification

BDR was verified using the following tools and approaches.

Coverity

Coverity Scan was 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 was 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

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

Test 5.2 can PASS if audited manually, but it doesn't have an automated test.

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

6.6 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 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 CRDT.
- 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 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.

!!! Warning "Upgrade Notes" All the SQL-visible interfaces are in the `bdr` schema. All the previously deprecated interfaces in the `bdr_conflicts` or `bdr_crdt` schema were removed and don't work on 3.7+ nodes or in groups that contain at least one 3.7+ node. Use the ones in the `bdr` schema that are already present in all BDR versions.

How conflicts happen

Inter-node conflicts arise as a result of sequences of events that 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.

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 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 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 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 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](#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 LiveCompare 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 is 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 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.

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 EDB Postgres Distributed 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.

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 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.

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 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.

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 namesq 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.

	insert_exists	update_differing	update_origin_change	update_missing	update_recently
error	X	X	X	X	X
skip	X	X	X	X	X
skip_if_recently_dropped					
update_if_newer	X	X	X		
update	X	X	X		

	insert_exists	update_differing	update_origin_change	update_missing	update_recently
insert_or_skip				X	X
insert_or_error				X	X
ignore					
ignore_if_null					
use_default_value					
conflict_trigger	X	X	X	X	X

Default conflict resolvers

Conflict type	Resolver
insert_exists	update_if_newer
update_differing	update_if_newer
update_origin_change	update_if_newer
update_missing	insert_or_skip
update_recently_deleted	skip
update_pkey_exists	update_if_newer
multiple_unique_conflicts	error
delete_recently_updated	skip
delete_missing	skip
target_column_missing	ignore_if_null
source_column_missing	use_default_value
target_table_missing	skip_if_recently_dropped
apply_error_ddl	error

List of conflict resolutions

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

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)
```

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 LiveCompare documentation for further details.

6.7 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 the `nextval()` function
- `serial` and `bigserial` columns or, alternatively, `GENERATED BY DEFAULT AS IDENTITY` columns

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

BDR global sequences

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

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

Using BDR global sequences allows you to avoid the problems with insert conflicts. If you define a `PRIMARY KEY` or `UNIQUE` constraint on a column that's using a global sequence, no node can 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 they are crash-safe. To use them, you must be 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 doesn't require inter-node communication at any point. It's faster and more robust and has the useful property of recording the timestamp at which the values were created.

SnowflakeId sequences have the restriction that they work only for 64-bit BIGINT datatypes and produce values 19 digits long, which might be too long for use in some host language datatypes such as Javascript Integer types. Globally allocated sequences allocate a local range of values that can be replenished as needed by inter-node consensus, making them suitable for either BIGINT or INTEGER sequences.

You can create a global sequence 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.

BDR also provides the configuration variable `bdr.default_sequence_kind`, which determines the kind of sequence to create 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 that consist of time, nodeid, and counter components. You can't use it with INTEGER sequences (so you can use it for `bigserial` but not for `serial`).
- `timeshard`, which is the older version of SnowflakeId sequence and is provided for backward compatibility only. The SnowflakeId is preferred.
- `distributed`, which is a special value that you can use only for `bdr.default_sequence_kind`. It selects `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 you can use them 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 don't require any inter-node communication after the node join process completes. So you can continue to use them even if there's the risk of extended network partitions. They aren't affected by replication lag or inter-node latency.

SnowflakeId sequences generate unique ids in a different way from standard sequences. The algorithm uses three 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 are always different. The third component is the number generated by the local sequence.

While adding a unique node id to the sequence number is enough to ensure there are no conflicts, we also want to keep another useful property of sequences. 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 are at least 19 digits long. There's no practical 32-bit `integer` version, so you can't use it with `serial` sequences. Use globally allocated range sequences instead.

For SnowflakeId there's a limit of 4096 sequence values generated per millisecond on any given node (about 4 million sequence values per second). In case the sequence value generation wraps around within a given millisecond, the SnowflakeId sequence waits until the next millisecond and gets a fresh value for that millisecond.

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

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

The `INCREMENT` option on a sequence used as input for SnowflakeId sequences is effectively ignored. This might 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 advances to a new noncolliding value by the time the application can do anything with the cached values.

Similarly, you might change the `START`, `MINVALUE`, `MAXVALUE`, and `CACHE` settings on the underlying sequence, but there's no benefit to doing so. The sequence's low 14 bits are used and the rest is discarded, so the value range limits don't affect the function's result. For the same reason, `setval()` isn't useful for SnowflakeId sequences.

Timeshard sequences

Timeshard sequences are provided for backward compatibility with existing installations but aren't recommended for new application use. We recommend using the SnowflakeId sequence instead.

Timeshard is very similar to SnowflakeId but has different limits and fewer protections and slower 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's no protection against wraparound within a given millisecond. Schemas using the timeshard sequence must protect the use of the `UNIQUE` constraint when using timeshard values for given column.
- The timestamp component of timeshard sequence runs out of values in the year 2050 and, if used in combination with bigint, the values wrap to negative numbers in the year 2033. This means that sequences generated after 2033

have negative values. This is a 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). SnowflakeIds 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 is used up.

Unlike SnowflakeId sequences, `galloc` sequences support all sequence data types provided by PostgreSQL: `smallint`, `integer`, and `bigint`. This means that you can use `galloc` sequences in environments where 64-bit sequences are problematic. Examples include 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 allocates 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 increase monotonically. However, viewed globally, the values generated aren't ordered at all. This might cause a loss of performance due to the effects on b-tree indexes and typically means that generated values aren't 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. This could lead to delays or operational issues if the majority of the BDR group isn't accessible. This might 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 the `galloc` kind. The `INCREMENT BY` option also works correctly. However, you can't assign an increment value that's equal to or more than the above ranges assigned for each sequence datatype. `setval()` doesn't reset the global state for `galloc` sequences; don't use it.

A few limitations apply to `galloc` sequences. BDR tracks `galloc` sequences in a special BDR catalog bdr.sequence_alloc. This catalog is required to track the currently allocated chunks for the `galloc` sequences. The sequence name and namespace is stored in this catalog. Since the sequence chunk allocation is managed by Raft, whereas any changes to the sequence name/namespace is managed by the replication stream, BDR currently doesn't support renaming `galloc` sequences or moving them to another namespace or renaming the namespace that contains a `galloc` sequence. 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, you can 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 quickly results in the values returned by `nextval()` exceeding the `int4` range if more than two nodes are in use.

The following example shows 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 contains the following values:

```
SELECT * FROM seqtest;
     id
------------
          2
 2000000002
(2 rows)
```

However, attempting the same operation on a third node fails with an `integer out of range` error, as the sequence generated the value `4000000002`.

!!! Tip You can retrieve the current data type of a sequence from the PostgreSQL pg_sequences view. You can modify the data type of a sequence with `ALTER SEQUENCE ... AS ...`, for example, `ALTER SEQUENCE public.sequence AS integer`, as long as its current value doesn't exceed 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's rewritten and restarts from the defined start value of the local sequence. If this happens on an existing sequence in a production database, you need to query the current value and 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're already using offset and you have writes from multiple nodes, you need to check what is the greatest used value and restart the sequence at least to the next value.

```
-- determine highest sequence value across all nodes
SELECT max((x->'response'->0->>'nextval')::bigint)
    FROM json_array_elements(
        bdr.run_on_all_nodes(
            E'SELECT nextval(\'public.sequence\');'
            )::jsonb AS x;

-- turn into a galloc sequence
SELECT bdr.alter_sequence_set_kind('public.sequence'::regclass, 'galloc', $MAX +
$MARGIN);
```

Since users can't 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 gives information on the chunk size and the ranges 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, which is 2 per node, and the chunk size is 1000000 that's related to an integer sequence.

```
SELECT * FROM bdr.sequence_alloc
    WHERE seqid = 'public.categories_category_seq'::regclass;
          seqid          | seq_chunk_size | seq_allocated_up_to | seq_nallocs |
seq_last_alloc
-------------------------+----------------+---------------------+-------------+---
--------------------------
 categories_category_seq |        1000000 |             4000333 |           4 |
2020-05-21 20:02:15.957835+00
(1 row)
```

To see the ranges currently assigned to a given sequence on each node, use these queries:

- Node `Node1` is using range from `333` to `2000333`.

```
SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = '(0,2)'; -- first range
 range_start | range_end
-------------+-----------
         334 |   1000333
(1 row)

SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = '(0,3)'; -- second range
 range_start | range_end
-------------+-----------
     1000334 |   2000333
(1 row)
```

- Node `Node2` is using range from `2000004` to `4000003`.

```
SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = '(0,2)'; -- first range
 range_start | range_end
-------------+-----------
     2000334 |   3000333
(1 row)

SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = '(0,3)'; -- second range
 range_start | range_end
-------------+-----------
     3000334 |   4000333
```

!!! NOTE You can't combine it to a single query (like `WHERE ctid IN ('(0,2)', '(0,3)')`) as that still shows only the first range.

When a node finishes a chunk, it asks a consensus for a new one and gets the first available. In the example, it's from 4000334 to 5000333. This is the new reserved chunk and starts to consume the old reserved chunk.

UUIDs, KSUUIDs, and other approaches

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

- UUIDs and their BDR variant, KSUUIDs
- Local sequences with a different offset per node (i.e., manual)
- An externally coordinated natural key

BDR applications can't use other methods safely: counter-table-based approaches relying on `SELECT ... FOR UPDATE`, `UPDATE ... RETURNING ...` or similar for sequence generation doesn't work correctly in BDR because BDR doesn't take row locks between nodes. The same values are generated on more than one node. For the same reason, the usual strategies for "gapless" sequence generation don't work with BDR. In most cases, the application coordinates generation of sequences that must be gapless from some external source using two-phase commit. Or it generates them only on one node in the BDR group.

UUIDs and KSUUIDs

`UUID` keys instead avoid sequences entirely and use 128-bit universal unique identifiers. These are random or pseudorandom values that are so large that it's nearly impossible for the same value to be generated twice. There's no need for nodes to have continuous communication when using `UUID` keys.

In the unlikely event of a collision, conflict detection chooses the newer of the two inserted records to retain. Conflict logging, if enabled, records such an event. However, it's exceptionally unlikely to ever occur, since collisions become practically likely only after about `2^64` keys are generated.

The main downside of `UUID` keys is that they're somewhat inefficient in terms of space and the network. They consume more space not only as a primary key but also where referenced in foreign keys and when transmitted on the wire. Also, not all applications cope well with `UUID` keys.

BDR provides functions for working with a K-Sortable variant of `UUID` data, known as KSUUID, which generates values that can be stored using the PostgreSQL standard `UUID` data type. A `KSUUID` value is similar to `UUIDv1` in that it stores both timestamp and random data, following the `UUID` standard. The difference is that `KSUUID` is K-Sortable, meaning that it's weakly sortable by timestamp. This makes it more useful as a database key 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 don't include the MAC of the computer on which they were generated, so there are no security concerns from using them.

`KSUUID` v2 is now recommended in all cases. You can directly sort values generated with regular comparison operators.

There are two versions of `KSUUID` in BDR: v1 and v2. The legacy `KSUUID` v1 is deprecated but is kept in order to support existing installations. Don't use it for new installations. The internal contents of v1 and v2 aren't compatible. As such, the functions to manipulate them also aren't compatible. The v2 of `KSUUID` also no longer stores the `UUID` version number.

Step and 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 sequence generates 1002, 2002, 3002, and so on. This scheme works well even if the nodes can't communicate for extended periods. However, the designer must specify a maximum number of nodes when establishing the schema, and it requires per-node configuration. Mistakes can easily lead to overlapping sequences.

It's 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()`, give each node a different offset starting value, for example:

```
-- On node 1
SELECT setval('some_seq', 1);

-- On node 2
SELECT setval('some_seq', 2);

 -- ... etc
```

Be sure to allow a large enough `INCREMENT` to leave room for all the nodes you might ever want to add, since changing it in future is difficult and disruptive.

If you use `bigint` values, there's no practical concern about key exhaustion, even if you use offsets of 10000 or more. It would take hundreds of years, with hundreds of machines, doing millions of inserts per second, to have any chance of approaching exhaustion.

BDR doesn't 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. You can create such a function by temporarily disabling DDL replication and creating a constant SQL

function. Alternatively, you can use a one-row table that isn't part of a replication set to store a different value in each node.

Global sequence management interfaces

BDR provides an interface for converting between a standard PostgreSQL sequence and the BDR global sequence.

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 visible only by way of the `bdr.sequences` view. In all other ways, the sequence appears as a normal sequence.

BDR treats this function as `DDL`, so DDL replication and global locking applies, if it's currently active. See DDL Replication.

Synopsis

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

Parameters

- `seqoid` — Name or Oid of the sequence to alter.
- `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 uses the sequence start value as the starting point. When there are existing values that were used by the sequence before it was changed to `galloc`, we recommend moving the starting point so that the newly generated values don't 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 takes a global `DDL` lock. It also locks the sequence locally.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

Only the owner of the sequence can execute the `bdr.alter_sequence_set_kind` function unless `bdr.backwards_compatibility` is set is set to 30618 or lower.

bdr.extract_timestamp_from_snowflakeid

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

timestamptz.

Synopsis

```
bdr.extract_timestamp_from_snowflakeid(snowflakeid bigint)
```

Parameters

- `snowflakeid` — Value of a snowflakeid sequence.

Notes

This function executes only on the local node.

bdr.extract_nodeid_from_snowflakeid

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

Synopsis

```
bdr.extract_nodeid_from_snowflakeid(snowflakeid bigint)
```

Parameters

- `snowflakeid` — Value of a snowflakeid sequence.

Notes

This function executes only on the local node.

bdr.extract_localseqid_from_snowflakeid

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

Synopsis

```
bdr.extract_localseqid_from_snowflakeid(snowflakeid bigint)
```

Parameters

- `snowflakeid` — Value of a snowflakeid sequence.

Notes

This function executes only 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` that's 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 uses an index scan on the column `id`.

Synopsis

```
bdr.timestamp_to_snowflakeid(ts timestamptz)
```

Parameters

- `ts` — Timestamp to use for the snowflakeid sequence generation.

Notes

This function executes only on the 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 executes only 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 executes only 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 executes only 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` that's 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 uses an index scan on the column `id`.

Synopsis

```
bdr.timestamp_to_timeshard(ts timestamptz)
```

Parameters

- `ts` — Timestamp to use for the timeshard sequence generation.

Notes

This function executes only on the local node.

KSUUID v2 Functions

Functions for working with `KSUUID` v2 data, K-Sortable UUID data.

bdr.gen_ksuuid_v2

This function generates a new `KSUUID` v2 value using the value of timestamp passed as an argument or current system time if NULL is passed. If you want to generate KSUUID automatically using the system time, pass a NULL argument.

The return value is of type UUID.

Synopsis

```
bdr.gen_ksuuid_v2(timestamptz)
```

Notes

This function executes only on the local node.

bdr.ksuuid_v2_cmp

This function compares the `KSUUID` v2 values.

It returns `1` if the first value is newer, `-1` if the 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 executes only on the 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 executes only on the local node.

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 executes only on the local node.

bdr.uuid_v1_cmp

This function compares the `KSUUID` v1 values.

It returns 1 if the first value is newer, -1 if the second value is lower, or zero if they are equal.

Synopsis

```
bdr.uuid_v1_cmp(uuid, uuid)
```

Notes

This function executes only 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 timestamptz.

Synopsis

```
bdr.extract_timestamp_from_ksuuid(uuid)
```

Parameters

- `UUID` — `KSUUID` v1 value to extract timestamp from.

Notes

This function executes on the local node.

6.8 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.

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 timestampstz)
```

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`.

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.

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.

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)
```

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.

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);
```

6.9 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 as traditional 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 without any user action needed.

CRDTs require the table to have column-level conflict resolution enabled, as documented in CLCD.

The only action you need to take is to use a particular data type in CREATE/ALTER TABLE rather than standard built-in data types such as integer. For example, 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);
```

Suppose you issue the following SQL on two nodes at same time:

```
UPDATE non_crdt_example
   SET counter = counter + 1    -- "reflexive" update
 WHERE id = 1;
```

After both updates are applied, you can see the resulting values using this query:

```
SELECT * FROM non_crdt_example WHERE id = 1;
   id |   counter
 -----+-----------
    1 |         1
(1 row)
```

This code shows that you lost one of the increments due to the `update_if_newer` conflict resolver. If you use the CRDT counter data type instead, the result looks 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 issue the following SQL on two nodes at same time, and 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 example 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 the example shows.

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. You can use the special hash operator (`#`) to request only the current numeric value without using the special operator (the default behavior). If the full state is dumped using `bdr.crdt_raw_value = on`, then the value can reload only with `bdr.crdt_raw_value = on`.

!!! Note The `bdr.crdt_raw_value` applies formatting only of data returned to clients, that is, simple column references in the select list. Any column references in other parts of the query (such as `WHERE` clause or even expressions in the select list) might still require use of the `#` operator.

Another class of CRDT data types is referred to *delta CRDT* types. These are a special subclass of operation-based CRDTs.

With delta CRDTs, any update to a value is compared to the previous value on the same node. 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);
```

Suppose you 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;
```

After both updates are applied, you can see the resulting values using this query:

```
SELECT id, counter FROM crdt_delta_example WHERE id = 1;
   id | counter
 -----+---------
    1 |       4
(1 row)
```

With a regular `integer` column, the result is `2`. But when you update the row with a delta CRDT counter, you start with the OLD row version, make a NEW row version, and send both to the remote node. There, compare them with the version found there (e.g., 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`. This makes them available without having to tweak `search_path`.

An important question is how query planning and optimization works with these new data types. CRDT types are handled transparently. Both `ANALYZE` and the optimizer work, so estimation and query planning works fine without having to do anything else.

State-based and operation-based CRDTs

Following the notation from [1], both operation-based and state-based CRDTs are implemented.

Operation-based CRDT types (CmCRDT)

The implementation of operation-based types is trivial because the operation isn't transferred explicitly but computed from the old and new row received from the remote node.

Currently, these operation-based CRDTs are implemented:

- `crdt_delta_counter` — `bigint` counter (increments/decrements)
- `crdt_delta_sum` — `numeric` sum (increments/decrements)

These types leverage existing data types (for example, `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 for which the method for computing the delta is known, but the result is simple and cheap (both in terms of space and CPU) and has a couple of additional benefits. For example, it can leverage operators/syntax for the underlying data type.

The main disadvantage is that you can't reset this value reliably in an asynchronous and concurrent environment.

!!! Note Implementing more complicated operation-based types by creating custom data types is possible, storing the state and the last operation. (Every change is decoded and transferred, so multiple operations aren't needed). But at that point, the main benefits (simplicity, reuse of existing data types) are lost without gaining any advantage compared to state-based types (for example, still no capability to reset) except for the space requirements. (A per-node state isn't needed.)

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, four state-based CRDTs are implemented:

- `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 added 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 doesn't happen in a cluster that operates properly), 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 includes even the original change from A. With operation-based CRDTs, node C doesn't receive the change until the A-C network connection starts working again.

The main disadvantages of CvCRDTs are higher costs in terms of disk space and CPU usage. A bit of information for each node is needed, including nodes that were already removed from the cluster. The complex nature of the state (serialized into varlena types) means increased CPU use.

Disk-space requirements

An important consideration is the overhead associated with CRDT types, particularly the on-disk size.

For operation-based types, this is 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's no dependency on the number of nodes because operation-based CRDT types don't 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's 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 you 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 doesn't matter how many nodes are in the cluster if the values are never updated on multiple nodes. It also doesn't matter whether the updates were concurrent (causing a conflict).

```
In addition, it doesn't matter how many of those nodes were already
removed from the cluster. There's no way to compact the state yet.
```

CRDT types versus conflicts handling

As tables can contain both CRDT and non-CRDT columns (most columns are expected to be non-CRDT), you need to do both the regular conflict resolution and CRDT merge.

The conflict resolution happens first and is responsible for deciding the tuple to keep (applytuple) and the one to discard. The merge phase happens next, merging data for CRDT columns from the discarded tuple into the applytuple.

!!! Note This handling makes CRDT types somewhat more expensive compared to plain conflict resolution because the merge needs to happen every time. This is the case even when the conflict resolution can use one of the fast-paths (such as those modified in the current transaction).

CRDT types versus conflict reporting

By default, detected conflicts are individually reported. Without CRDT types, this makes 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, conflict reporting is skipped when the conflict can be fully resolved by CRDT merge, that is, if each column meets at least one of these two conditions:

- The values in local and remote tuple are the same (NULL or equal).
- It uses a CRDT data type and so can be merged.

!!! Note This means that the conflict reporting is also skipped when there are no CRDT columns but all values in local/remote tuples are equal.

Resetting CRDT values

Resetting CRDT values is possible but requires special handling. The asynchronous nature of the cluster means that different nodes might see the reset operation at different places in the change stream no matter how it's implemented. Different nodes might also initiate a reset concurrently, that is, 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 that might work well on a single-node fail for 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 doesn't work. It throws away the state for the other nodes but only locally. It's 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 might seem to work because the update is interpreted as a subtraction of `-cnt`. But it works only in the absence of concurrent resets. Once two nodes attempt to do a reset at the same time, the delta is applied twice, getting a negative value (which isn't expected from a reset).

It might also seem that you can use `DELETE + INSERT` as a reset, but this approach has a couple of weaknesses, too. If the row is reinserted with the same key, it's not guaranteed that all nodes see it at the same position in the stream of operations with respect to changes from other nodes. BDR specifically discourages reusing 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 ...;
```

"Reliably" means the values don't have the two issues of multiple concurrent resets and divergence.

Operation-based CRDT types can be reset reliably only using Eager Replication, since this avoids multiple concurrent resets. You can also use Eager Replication to set either kind of CRDT to a specific value.

Implemented CRDT data types

Currently, there are six CRDT data types implemented:

- Grow-only counter and sum
- Positive-negative counter and sum
- 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], might be implemented later.

You can list the currently implemented CRDT data types 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 nonnegative values (`value + int` and `counter + bigint` operators).
- You can obtain the current value of the counter either using `#` operator or by casting it to `bigint`.
- Isn't 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`).
- You can inspect the internal state 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 nonnegative values (`sum + numeric`).

- You can obtain the current value of the sum either by using the `#` operator or by casting it to `numeric`.
- Isn't 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`).
- Can inspect internal state 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).
- You can obtain the current value of the counter either by using the `#` operator or by casting to `bigint`.
- Isn't 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`).
- You can inspect the internal state 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`).
- You can obtain the current value of the sum either by using then `#` operator or by casting to `numeric`.
- Isn't 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`).
- You can inspect the internal state using `crdt_pnsum_to_text`.

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

INSERT INTO crdt_test VALUES (1, 0);       -- initialized to 0
```

```
INSERT INTO crdt_test VALUES (2, 1298.24); -- initialized to 1298.24
INSERT INTO crdt_test VALUES (3, -45.31);  -- initialized to -45.31

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

-- increment sums
UPDATE crdt_test SET pnsum = pnsum + 1.44     WHERE id = 1;
UPDATE crdt_test SET pnsum = pnsum + 12.20    WHERE id = 2;
UPDATE crdt_test SET pnsum = pnsum + (-24.34) WHERE id = 3;

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

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

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

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

-- get current sum value after the reset
SELECT id, pnsum::numeric, pnsum FROM crdt_test;
```

delta counter (`crdt_delta_counter`)

- Is defined a `bigint` domain, so works exactly like a `bigint` column.
- Supports increments with both positive and negative values.
- Is compatible with simple assignments like `counter = value` (common when the new value is computed somewhere in the application).
- There's no simple way to reset the value reliably.

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

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

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

```
-- increment counters
UPDATE crdt_test SET cnt = cnt + 1      WHERE id = 1;
UPDATE crdt_test SET cnt = cnt + 120    WHERE id = 2;
UPDATE crdt_test SET cnt = cnt + (-244) WHERE id = 3;

-- decrement counters
UPDATE crdt_test SET cnt = cnt - 73    WHERE id = 1;
UPDATE crdt_test SET cnt = cnt - 19283 WHERE id = 2;
UPDATE crdt_test SET cnt = cnt - (-12) WHERE id = 3;

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

delta sum (`crdt_delta_sum`)

- Is defined as a `numeric` domain so works exactly like a `numeric` column.
- Supports increments with both positive and negative values.
- Is compatible with simple assignments like `sum = value` (common when the new value is computed somewhere in the application).
- There's 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

6.10 Transaction streaming

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

Beginning with PostgreSQL 14, transactions can optionally be decoded and sent to subscribers before they're 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 subscribers as soon as the transaction commits.

BDR enhancements

PostgreSQL's built-in transaction streaming has the following limitations:

- While you no longer need to spill changes to disk on the publisher, you must write changes to disk on each subscriber.
- If the transaction aborts, the work (changes received by each subscriber and the associated storage I/O) is wasted.

However, starting with version 3.7, BDR supports parallel apply, enabling multiple writer processes on each subscriber. This capability is leveraged to provide the following enhancements:

- Decoded transactions can be streamed directly to a writer on the subscriber.
- Decoded transactions don't need to be stored on disk on subscribers.
- You don't need to wait for the transaction to commit before starting to apply the transaction on the subscriber.

Caveats

- You must enable parallel apply.
- Workloads consisting of many small and conflicting transactions can lead to frequent deadlocks between writers.

!!! Note Direct streaming to writer is still an experimental feature. Use it with caution. Specifically, it might not work well with conflict resolutions since the commit timestamp of the streaming might not be available. (The transaction might not yet have committed on the origin.)

Configuration

Configure transaction streaming in two locations:

- At node level, using the GUC bdr.default_streaming_mode
- At group level, using the function bdr.alter_node_group_config()

Node configuration using bdr.default_streaming_mode

Permitted values are:

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

- `auto`

Default value is `auto`.

Changing this setting requires a restart of the pglogical receiver process for each subscription for the setting to take effect. You can achieve this with a server restart.

If `bdr.default_streaming_mode` is set any value other than `off`, the subscriber requests transaction streaming from the publisher. How this is provided can also depend on the group configuration setting. See Node configuration using bdr.default_streaming_mode for details.

Group configuration using bdr.alter_node_group_config()

You can use the parameter `streaming_mode` in the function bdr.alter_node_group_config() to set the group transaction streaming configuration.

Permitted values are:

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

The default value is `default`.

The value of the current setting is contained in the column `node_group_streaming_mode` from the view 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 the subscriber level by the GUC `bdr.default_streaming_mode`. Unless set to `off` (which disables transaction streaming), the subscriber requests transaction streaming.

If the publisher can provide transaction streaming, it streams transactions whenever the transaction size exceeds the threshold set in `logical_decoding_work_mem`. The publisher usually has no control over whether the transactions is streamed to a file or to a writer. Except for some situations (such as COPY), it might hint for 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's 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 is streamed to a file. See bdr.writers to check BDR writer status.

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

Group	Node	Streamed to
off	(any)	(none)

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

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 only if the publisher hinted to do this.

Currently the publisher hints for the subscriber to stream to the writer for the following transaction types. These are known to be conflict free and can be safely handled by the writer.

- `COPY`
- `CREATE INDEX CONCURRENTLY`

Monitoring

You can monitor the use of transaction streaming using the 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 that were started.
- `nstream_stop` — Number of streamed transactions that were fully received.

6.11 Durability and performance options

Overview

Synchronous or *Eager Replication* synchronizes between at least two nodes of the cluster before committing a transaction. This synchronization provides three properties of interest to applications that are related but can all be implemented individually:

- *Durability*: Writing to multiple nodes increases crash resilience and allows you to recover the data 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 Group Commit feature to guarantee durability and visibility by providing a variant of synchronous replication. This feature is similar to the Postgres `synchronous_commit` feature for physical standbys but provides

more flexibility for large-scale distributed systems.

In addition to Group Commit, BDR also offers two other modes that can't currently be combined with Group Commit:

- Commit At Most Once (CAMO). This feature solves the problem with knowing whether your transaction has committed (and replicated) in case of node or network failures COMMIT. Normally, it might be hard to know whether the COMMIT was processed in. With this feature, your application can find out what happened, even if your new database connection is to a node different from your previous connection. For more information about this feature, see Commit At Most Once.
- 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 reduces performance but provides strong consistency guarantees. For more information about this feature, see Eager All-Node Replication.

Postgres provides Physical Streaming Replication (PSR), which is unidirectional but offers a synchronous variant. For backward compatibility, BDR still supports configuring synchronous replication with `synchronous_commit` and `synchronous_standby_names`. See Legacy synchronous replication, but the use of Group Commit is recommended instead in all cases.

Terms and definitions

BDR nodes can take 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's 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, that is, the origin and all of its partner nodes, which can be just a few or all peer nodes.

Comparison

Most options for synchronous replication available to BDR allow for different levels of synchronization, offering different tradeoffs 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. And 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

Variant	Mode	Received	Visible	Durable
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. Consider using Group Commit instead.

Reception ensures the peer operating normally can eventually apply the transaction without requiring any further communication, even in the face of a full or partial network outage. A crash of a peer node might still require retransmission of the transaction, as this confirmation doesn't involve persistent storage. All modes considered synchronous provide this protection.

Visibility implies the transaction was applied remotely. All other clients 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 might 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.

Internal timing of operations

For a better understanding of how the different modes work, it's helpful to realize PSR and BDR apply transactions 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, 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 precommit 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

Configuration

The following table provides an overview of the configuration settings that are required to be set to a nondefault 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.

Planned shutdown and restarts

When using Group Commit with receive confirmations or CAMO in combination with `remote_write`, take care 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 can lead to loss of data, as if both nodes failed.

To ensure the apply queue gets flushed to disk, use either `smart` or `fast` shutdown for maintenance tasks. This approach maintains the required synchronization level and prevents loss of data.

Legacy synchronous replication using BDR

!!! Note We don't recommend this approach. Consider using Group Commit instead.

Usage

To enable synchronous replication using BDR, you need to add the application name of the relevant BDR peer nodes to `synchronous_standby_names`. The use of `FIRST x` or `ANY x` offers a some flexibility if this doesn't conflict with the requirements of non-BDR standby nodes.

Once you've added it, you can configure the level of synchronization per transaction using `synchronous_commit`, which defaults to `on`. This setting means 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 you 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 similar, the former doesn't 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.

6.12 Group Commit

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

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 triggers 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, configure `max_prepared_transactions` high enough to handle all such transactions originating per node.

Configuration

To use Group Commit, first define a commit scope. This determines the BDR nodes involved in the commit of a transaction. Once a scope is established, you can configure a transaction to use Group Commit as follows:

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

For this example, you might previously have defined the commit scope 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 subgroups. Any transaction committed in the `example_scope` requires one extra confirmation from a BDR node in 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 can depend on the node the transaction is committed on, that is, the node that 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 EDB Postgres Distributed 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`. You can use the following commands to set up subgroups 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 receiving it, prior to starting the 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 are flushed to disk.
- `visible` (default) — Confirm the transaction after all of its changes are flushed to disk and it's visible to concurrent transactions.

In rules for commit scopes, you can append these confirmation levels to the node group definition in parenthesis with `ON` as follows:

- `ANY 2 (right_dc) ON replicated`
- `ALL (left_dc) ON visible` (default and may as well be omitted)
- `ALL (left_dc) ON received AND ANY 1 (right_dc) ON durable`

Reference

Commit scope grammar

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

```
commit_scope:
    confirmation [AND ...]

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

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

!!! 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.

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 EDB Postgres Distributed cluster, invoking this function once for the top-level node group is enough to fully define the commit scope.

Alternatively, you can invoke it 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 in a commit scope, you can use the function `bdr.alter_commit_scope`.

Synopsis

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

Removing a commit scope rule

You can use `bdr.remove_commit_scope` to drop a single rule in a commit scope. If you define multiple rules for the commit scope, you must invoke this function once per rule to fully remove the entire commit scope.

Synopsis

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

!!! Note Removing a commit scope that is still used as default by a node group is not allowed

---

6.13 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 applies changes and resolves potential conflicts post-commit, as described in the Conflicts.

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 times out after `bdr.global_commit_timeout`, leading to an abort of the transaction. There's 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 confirms the commit to the client only after all nodes have committed to

ensure that the transaction is immediately visible on all nodes after the commit.

Requirements

Eager All-Node Replication uses prepared transactions internally. Therefore all replica nodes need to have a `max_prepared_transactions` configured high enough to handle all incoming transactions, possibly in addition to local two-phase commit and CAMO transactions. (See Configuration: Max-prepared transactions). We recommend configuring 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 EDB Postgres Distributed cluster can run Eager All-Node transactions.

Usage

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

```
BEGIN;

... other commands possible...

SET LOCAL bdr.commit_scope = 'global';

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

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

```
COMMIT;
```

Error handling

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

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

Eager All-Node Replication with CAMO

Eager All-Node Replication goes beyond CAMO and implies it. There's no need to also enable `bdr.enable_camo` if `bdr.commit_scope` is set to `global`. Nor does a CAMO pair need to be configured with `bdr.add_camo_pair()`.

You can use any other active BDR node in the role of a CAMO partner to query a transaction's status. However, you must indicate this non-CAMO usage to the `bdr.logical_transaction_status` function with a third argument of `require_camo_partner = false`. Otherwise, it might complain about a missing CAMO configuration (which isn't required for Eager transactions).

Other than this difference in configuration and invocation of that function, the client needs to adhere to the protocol described for CAMO. Reference client implementations are provided to customers on request .

Limitations

Transactions using Eager Replication can't yet execute DDL, nor do they support explicit two-phase commit. These might be allowed in later releases. The `TRUNCATE` command is allowed.

Replacing a crashed and unrecoverable BDR node with its physical standby isn't 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.

You can't combine Eager All-Node Replication with `synchronous_replication_availability = 'async'`. Trying to configure both causes an error.

Eager All-Node transactions are not currently supported in combination with the Decoding Worker feature nor with transaction streaming. Installations using Eager must keep `enable_wal_decoder` and `streaming_mode` disabled for the BDR node group.

Synchronous replication uses a mechanism for transaction confirmation different from Eager. The two aren't compatible, and you must not use them together. Therefore, whenever using Eager All-Node transactions, 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 possible.

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 roundtrips (to the furthest peer node in the worst case). Logical standby nodes and nodes still in the process of joining or catching up aren't included but 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. 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.

6.14 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 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 on request.

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.

Configuration

Assume an existing EDB Postgres Distributed cluster consists of the nodes `node1` and `node2`. Both notes 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 EDB Postgres Distributed 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 CAME 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 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.

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.

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.

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`

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.

- CAMO is not currently compatible with transaction streaming. Please ensure to disable transaction streaming when planning to use CAMO. This can be configured globally or in the BDR node group, see Transaction Streaming Configuration.

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.

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.

CAMO versus group commit

CAMO doesn't currently work with group commit.

6.15 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 might 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:

- Recovery point objective (RPO) specifies the maximum tolerated amount of data that can be lost due to unplanned events, usually expressed as an amount of time. In nonreplicated systems, RPO is used to set backup intervals to limit the risk of lost committed data due to failure. For replicated systems, RPO determines the acceptable amount of committed data that hasn't been safely applied to one or more peer nodes.

- Resource constraint objective (RCO) acknowledges that there are finite storage constraints. This storage includes database files, WAL, and temporary or intermediate files needed for continued operation. For replicated systems, as lag increases the demands on these storage resources also increase.

- Group elasticity objective (GEO) ensures that any node isn't originating new data at a clip that can't be acceptably saved to its peer nodes. When that is the case then the detection of that condition can be used as one metric in the decision to expand the number of database nodes. Similarly, when that condition abates then it might influence the decision to contract the number of database nodes.

Lag control manages replication lag by controlling the rate at which client connections can commit READ WRITE

transactions. Replication lag is measured either as lag time or lag size, depending on the objectives to meet. Transaction commit rate is regulated using a configured BDR commit-delay time.

Requirements

To get started using lag control:

1. Determine the maximum acceptable commit delay time `bdr.lag_control_max_commit_delay` that can be tolerated for all database applications.
2. Decide on the lag measure to use. Choose either lag size `bdr.lag_control_max_lag_size` or lag time `bdr.lag_control_max_lag_time`.
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.

Configuration

Lag control is configured on each BDR node in the group using `postgresql.conf` configuration parameters. To enable lag control, set `bdr.lag_control_max_commit_delay` and either `bdr.lag_control_max_lag_size` or `bdr.lag_control_max_lag_time` 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 1. For a complete list, see Lag control

Overview

Lag control is a dynamic TPS rate-limiting mechanism that operates at the client connection level. It's 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 stays fixed at 0 milliseconds.

If this isn't the case, minimally adjust the BDR runtime commit delay as high as needed, but no higher, until the number of conforming nodes returns 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 might rise and fall around an equilibrium level most of the time, but if data throughput or lag-apply rates improve then the commit delay decreases 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 are released. Therefore, the delay doesn't prevent concurrent active transactions from observing or modifying its values or acquiring its resources. The same guarantee can't be made for external resources managed by Postgres extensions. Regardless of 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.

!!! Note Don't conflate the BDR commit delay with the Postgres commit delay. They are unrelated and perform different functions. Don't substitute one for the other.

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 doesn't modify data, including declared READ WRITE transactions, is exempt from the commit delay.

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

Postgres and BDR auxillary processes don't delay at transaction commit. Most notably, BDR writers don't 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.

Limitations

The maximum commit delay `bdr.lag_control_max_commit_delay` is a ceiling value representing a hard limit. This means that a commit delay never exceeds 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's no additional back pressure on the lag measures to prevent their continued increase.

There's no way to exempt origin transactions that don't modify BDR replication sets from the commit delay. For these transactions, it can be useful to SET LOCAL the maximum transaction delay to 0.

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 effective. 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 can be unduly restrained by the acceptable BDR commit delay setting.

In these cases, it can be useful to use the `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 deference 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.

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 consistently exceed their BDR-configured maximum levels, this scenario can be a marker for BDR group expansion.

6.16 AutoPartition

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

You can create new partitions regularly and then drop them when the data retention period expires.

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

Auto creation of partitions

`bdr.autopartition()` creates or alters the definition of automatic range partitioning for a table. If no definition exists, it's created. Otherwise, later executions will alter the definition.

`bdr.autopartition()` doesn't lock the actual table. It changes the definition of when and how new partition maintenance actions 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 isn't 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` causes a new partition to be added each day, with partition bounds that are one day apart.

If the partition column is connected to a `snowflakeid`, `timeshard`, or `ksuuid` sequence, you must specify the `partition_increment` 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` causes 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 you don't specify `partition_initial_lowerbound`, 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` is set to CURRENT DATE. If `partition_increment` is specified as `1 Hour`, then `partition_initial_lowerbound` is set to the current hour of the current date. The bounds for the subsequent partitions are set using the `partition_increment` value.

The system always tries to have a certain minimum number of advance partitions. To decide whether to create new partitions, it uses the specified `partition_autocreate_expression`. This can be an expression that can be evaluated by SQL, 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 are 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`, you can specify the `partition_autocreate_expression` 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. Create an index on the `partcol` so that the query runs efficiently. If the `partition_autocreate_expression` isn't specified for a partition table on column type `integer`, `smallint`, or `bigint`, then the system sets it to `max(partcol)`.

If the `data_retention_period` is set, partitions are dropped after this period. Partitions are dropped at the same time as new partitions are added, to minimize locking. If this value isn't set, you must drop the partitions manually.

The `data_retention_period` parameter is supported only for timestamp (and related) based partitions. The period is calculated by considering the upper bound of the partition. 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. You can change this behavior 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 isn't a replicated table and hence it might not be necessary or even desirable to have all partitions on all nodes. For example, the built-in `bdr.conflict_history` table isn't 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.

You can't later change tables marked as `managed_locally` to be managed globally and vice versa.

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

You aren't expected to manually create or drop partitions for tables managed by AutoPartition. Doing so can make the AutoPartition metadata inconsistent and might 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 is created.
- `partition_autocreate_expression` — Used to detect if it's time to create new partitions.
- `minimum_advance_partitions` — The system attempts 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 value must be greater than `migrate_after_period`.
- `managed_locally` — If true, then the partitions are managed locally.
- `enabled` — Allows activity to be disabled or paused and later resumed or reenabled.

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 five advance partitions when there are only two more partitions remaining (each partition can hold 1 billion orders):

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

Create one AutoPartition

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

Synopsis

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

Parameters

- `relname` — Name or Oid of the parent table to attach to.

- `partname` — Name of the new AutoPartition.
- `lowerb` — The lower bound of the partition.
- `upperb` — The upper bound of the partition.
- `nodes` — List of nodes that the new partition resides on.

Stopping automatic 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 was successfully dropped, the function returns `true`.

Synopsis

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

Parameters

- `relname` — The name of the partitioned table to drop.

Notes

This places 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 doesn't guarantee that the partitions also exists on the remote nodes.

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 doesn't 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 or disable AutoPartitioning

Use `bdr.autopartition_enable()` to enable AutoPartitioning on the given table. If AutoPartitioning is already enabled, then no action occurs. 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 returns `unknown`. `In-progress` is the typical status.

6.17 Timestamp-based snapshots

The timestamp-based snapshots allow reading data in a consistent manner by using a user-specified timestamp rather than the usual MVCC snapshot. You can use this to access data on different BDR nodes at a common point in time. For example, you can use this as a way to compare data on multiple nodes for data-quality checking.

This feature doesn't currently work with write transactions.

Enable the use of timestamp-based snapshots using the `snapshot_timestamp` parameter. This parameter accepts either a timestamp value or a special value, `'current'`, which represents the current timestamp (now). If `snapshot_timestamp` is set, queries use that timestamp to determine visibility of rows rather than the usual MVCC semantics.

For example, the following query returns the 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 works only with future timestamps or the special 'current' value, so you can't use it for historical queries.

BDR works with and improves on that feature in a multi-node environment. First, BDR makes sure that all connections to other nodes replicate any outstanding data that was added to the database before the specified timestamp. This ensures that the timestamp-based snapshot is consistent across the whole multi-master group. Second, BDR adds a parameter called `bdr.timestamp_snapshot_keep`. This specifies a window of time when you can execute queries against the recent history on that node.

You can specify any interval, but be aware that VACUUM (including autovacuum) doesn't clean dead rows that are newer than up to twice the specified interval. This also means that transaction ids aren't 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, although you can change that as needed.

Once the query is accepted for execution, the query might run for longer than `bdr.timestamp_snapshot_keep` without problem, just as normal.

Also, information about how far the snapshots were kept doesn't survive server restart. The oldest usable timestamp for the timestamp-based snapshot is the time of last restart of the PostgreSQL instance.

You 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. You can use this to run parallel queries across nodes, when used 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 doesn't increase inter-node traffic or bandwidth. Only the timestamp value is passed in addition to query data.

6.18 Replication sets

A replication set is a group of tables that a BDR node can subscribe to. You can use replication sets to create more complex replication topologies than regular symmetric multi-master where each node is an exact copy of the other nodes.

Every BDR group creates a replication set with the same name as the group. This replication set is the default replication set, which is used for all user tables and DDL replication. All nodes are subscribed to it. In other words, by default all user tables are replicated between all nodes.

Using replication sets

You can create replication sets using `create_replication_set()`, specifying whether to include insert, update, delete, or truncate actions. One option lets you add existing tables to the set, and a second option defines whether to add tables when they are created.

You can also manually define the tables to add or remove from a replication set.

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

Once the node is joined, you can still remove tables from the replication set, but you must add new tables using a resync operation.

By default, a newly defined replication set doesn't replicate DDL or BDR administration function calls. Use `replication_set_add_ddl_filter` to define the commands to replicate.

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

You can use functions to alter these definitions later or to drop the replication set.

!!! Note Don't use the default replication set for selective replication. Don't drop or modify the default replication set on any of the BDR nodes in the cluster as it is also used by default for DDL replication and administration function calls.

Behavior of partitioned tables

BDR supports partitioned tables transparently, meaning that you can add a partitioned table to a replication set. Changes that involve any of the partitions are 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.

You can add individual partitions to the replication set, in which case they are 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 doesn't have to be executed.

!!! Note If a root partitioned table is part of any replication set, memberships of individual partitions are ignored. only the membership of that root table is taken into account.

Behavior with foreign keys

A foreign key constraint ensures that each row in the referencing table matches a row in the referenced table. Therefore, if the referencing table is a member of a replication set, the referenced table must also be a member of the same replication set.

The current version of BDR doesn't automatically check for or enforce this condition. When adding a table to a replication set, the database administrator must make sure that all the tables referenced by foreign keys are also added.

You can use the following query to list all the foreign keys and replication sets that don't satisfy this requirement. The referencing table is a member of the replication set, while the referenced table isn't:

```
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` isn't.

The `TRUNCATE CASCADE` command takes into account the replication set membership before replicating the command. For example:

```
TRUNCATE table1 CASCADE;
```

This becomes a `TRUNCATE` without cascade on all the tables that are part of the replication set only:

```
TRUNCATE table1, referencing_table1, referencing_table2 ...
```

Replication set management

Management of replication sets.

With the exception of `bdr.alter_node_replication_sets`, the following functions are considered to be `DDL`. DDL replication and global locking apply to them, if that's currently active. See DDL replication.

bdr.create_replication_set

This function creates a replication set.

Replication of this command is affected by DDL replication configuration including DDL filtering settings.

Synopsis

```
bdr.create_replication_set(set_name name,
                           replicate_insert boolean DEFAULT true,
                           replicate_update boolean DEFAULT true,
                           replicate_delete boolean DEFAULT true,
                           replicate_truncate boolean DEFAULT true,
                           autoadd_tables boolean DEFAULT false,
                           autoadd_existing boolean DEFAULT true)
```

Parameters

- `set_name` — Name of the new replication set. Must be unique across the BDR group.
- `replicate_insert` — Indicates whether to replicate inserts into tables in this replication set.
- `replicate_update` — Indicates whether to replicate updates of tables in this replication set.
- `replicate_delete` — Indicates whether to replicate deletes from tables in this replication set.
- `replicate_truncate` — Indicates whether to replicate truncates of tables in this replication set.
- `autoadd_tables` — Indicates whether to replicate newly created (future) tables to this replication set
- `autoadd_existing` — Indicates whether to add all existing user tables to this replication set. This parameter has an effect only if `autoadd_tables` is set to `true`.

Notes

By default, new replication sets don't replicate DDL or BDR administration function calls. See ddl filters for how to set up DDL replication for replication sets. A preexisting DDL filter is set up for the default group replication set that replicates all DDL and admin function calls. It's created when the group is created but can be dropped in case you don't want 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 takes a `DDL` global lock.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. 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 to replicate inserts into tables in this replication set.
- `replicate_update` — Indicates whether to replicate updates of tables in this replication set.
- `replicate_delete` — Indicates whether to replicate deletes from tables in this replication set.
- `replicate_truncate` — Indicates whether to replicate truncates of tables in this replication set.
- `autoadd_tables` — Indicates whether to add newly created (future) tables to this replication set.

Any of the options that are set to NULL (the default) 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 takes a `DDL` global lock.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. 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 takes a `DDL` global lock.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

!!! Warning Don't drop a replication set that's being used by at least another node, because doing so stops replication on that node. If that happens, unsubscribe the affected node from that replication set. For the same reason, don't drop a replication set with a join operation in progress when the node being joined is a member of that replication set. Replication set membership is checked only at the beginning of the join. This happens because the information on replication set usage is local to each node, so that you can configure it on a node before it joins the group.

You can manage replication set subscription for a node using `alter_node_replication_sets`.

bdr.alter_node_replication_sets

This function changes the replication sets a node publishes and is subscribed to.

Synopsis

```
bdr.alter_node_replication_sets(node_name name,
                                set_names text[])
```

Parameters

- `node_name` — The node to modify. Currently has to be local node.
- `set_names` — Array of replication sets to replicate to the specified node. An empty array results in the use of the group default replication set.

Notes

This function is executed only on the local node and isn't replicated in any manner.

The replication sets listed aren't checked for existence, since this function is designed to execute before the node joins. Be careful to specify replication set names correctly to avoid errors.

This allows for calling the function not only on the node that's part of the BDR group but also on a node that hasn't joined any group yet. This approach limits the data synchronized during the join. However, the schema is always fully synchronized without regard to the replication sets setting. All tables are copied across, not just the ones specified in the replication set. You can drop unwanted tables 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 otherwise prevent replication of DDL.

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

Replication set membership

You can add tables to or remove them from one or more replication sets. This affects replication only 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 with the node replication sets configuration to determine the actions to replicate to which node. The decision is done using the union of all the memberships and replication set options. Suppose that a table is a member of replication set A that replicates only INSERT actions and replication set B that replicates only UPDATE actions. Both INSERT and UPDATE act8ions are 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 adds a table to a replication set and starts replicating changes from this moment (or rather transaction commit). Any existing data the table might have on a node isn't 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 isn't defined (that is, set to 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 replicate the row. A False value doesn't replicate the row. Expressions can't contain subqueries or refer to variables other than columns of the current row being replicated. You can't reference system columns.

`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 completely avoids sending data for filtered rows. Hence network bandwidth is reduced and overhead on the target node is applied.

`row_filter` never removes `TRUNCATE` commands for a specific table. You can filter away `TRUNCATE` commands at the replication set level.

You can replicate just some columns of a table. See Replicating between nodes with differences.

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 takes a `DML` global lock on the relation that's being added to the replication set if the `row_filter` isn't NULL. Otherwise it takes just a `DDL` global lock.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. 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 the same replication mechanism as `DDL` statements. This means the replication is affected by the ddl filters configuration.

The function takes a `DDL` global lock.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

Listing replication sets

You can list existing replication sets with the following query:

```
SELECT set_name
FROM bdr.replication_sets;
```

You can use this query to list all the tables in a given replication set:

```
SELECT nspname, relname
FROM bdr.tables
WHERE set_name = 'myrepset';
```

In Behavior with foreign keys, we show a query that lists all the foreign keys whose referenced table isn't 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 code produces output like this:

```
  node_id   | node_name |    pub_repsets     |   sub_repsets
------------+-----------+--------------------+--------------------
 1834550102 | s01db01   | {bdrglobal,bdrs01} | {bdrglobal,bdrs01}
(1 row)
```

To execute the same query against all nodes in the cluster, you can use the following query. This approach gets the replication sets associated with all nodes at the same time.

```
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 shows, for example:

```
  node_id   | node_name |    pub_repsets     |    sub_repsets
------------+-----------+--------------------+--------------------
 933864801  | s02db01   | {bdrglobal,bdrs02} | {bdrglobal,bdrs02}
 1834550102 | s01db01   | {bdrglobal,bdrs01} | {bdrglobal,bdrs01}
 3898940082 | s01db02   | {bdrglobal,bdrs01} | {bdrglobal,bdrs01}
 1102086297 | s02db02   | {bdrglobal,bdrs02} | {bdrglobal,bdrs02}
(4 rows)
```

DDL replication filtering

By default, the replication of all supported DDL happens by way of the default BDR group replication set. This is achieved with a DDL filter with the same name as the BDR group. This filter is added to the default BDR group replication set when the BDR group is created.

You can adjust this 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 is replicated only once, even if it's matched by multiple filters on multiple replication sets.

You can list existing DDL filters with the following query, which shows for each filter the regular expression applied to the command tag and to the role name:

```
SELECT * FROM bdr.ddl_replication;
```

You can use the following functions to manipulate DDL filters. They are considered to be `DDL` and are 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 is replicated to any node that's subscribed to that set. This also affects replication of BDR admin functions.

This doesn't prevent execution of DDL on any node. It only alters whether DDL is replicated to other nodes. Suppose two nodes have a replication filter between them that excludes all index commands. 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 the 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.

You can filter the BDR admin functions used by using a tagname matching the qualified function name. For example, `bdr.replication_set_add_table` is the command tag for the function of the same name. In this case, this tag allows all BDR functions to be filtered using `bdr.*`.

The `role_name` is used for matching against the current role that 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,
                                   base_relation_name text DEFAULT NULL,
                                   query_match text DEFAULT NULL,
                                   exclusive boolean DEFAULT FALSE)
```

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
- `base_relation_name` — reserved for future use, must be NULL
- `query_match` — regular expression for matching the query; NULL means match all queries
- `exclusive` — if true, other matched filters are not taken into consideration (that is, only the exclusive filter is applied), when multiple exclusive filters match, we throw an error. This is useful for routing specific commands to specific replication set, while keeping the default replication through the main replication set.

Notes

This function uses the same replication mechanism as `DDL` statements. This means that the replication is affected by the ddl filters configuration. This also means that replication of changes to ddl filter configuration is affected by the existing ddl filter configuration.

The function takes a `DDL` global lock.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

To view the defined replication filters, 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 shows 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. This also means that replication of changes to the DDL filter configuration is affected by the existing DDL filter configuration.

The function takes a `DDL` global lock.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

6.19 Stream triggers

BDR introduces new types of triggers that you can use 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.

Multiple trigger definitions can use one trigger function, just as with normal PostgreSQL triggers. A trigger function is a program defined in this form: `CREATE FUNCTION ... RETURNS TRIGGER`. Creating the trigger doesn't require use of the `CREATE TRIGGER` command. Instead, create stream triggers using the special BDR functions `bdr.create_conflict_trigger()` and `bdr.create_transform_trigger()`.

Once created, the trigger is visible in the catalog table `pg_trigger`. The stream triggers are marked as `tgisinternal = true` and `tgenabled = 'D'` and have the name suffix '_bdrc' or '_bdrt'. The view `bdr.triggers` provides information on the triggers in relation to the table, the name of the procedure that is being executed, the event that triggers it, and the trigger type.

Stream triggers aren't 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. This mechanism prevents the trigger from executing as a normal SQL trigger.

These triggers execute on the downstream or target node. There's no option for them to execute on the origin node. However, you might want to consider the use of `row_filter` expressions on the origin.

Also, any DML that is applied while executing a stream trigger isn't replicated to other BDR nodes and doesn't trigger the execution of standard local triggers. This is intentional. You can use it, for example, to log changes or conflicts captured by a stream trigger into a table that is crash-safe and specific of that node. See Stream triggers examples for a working example.

Trigger execution during Apply

Transform triggers execute first—once for each incoming change in the triggering table. These triggers fire before we attempt 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's no target row, then no further processing occurs for those change types.

We then execute any normal triggers that previously were explicitly enabled as replica triggers at table-level:

```
ALTER TABLE tablename
ENABLE REPLICA TRIGGER trigger_name;
```

We then decide whether a potential conflict exists. 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 row-type of the target table.

Any column that exists on the input row but not on the target table triggers 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 example, 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. The `ignore_if_null` resolver isn't appropriate for a rolling upgrade because it breaks replication as soon as the user inserts a tuple with a non-NULL value in the new column in any of the upgraded nodes.

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.

You can do this with the following query, which you must execute separately on each node. Replace `node1` with the actual node name.

```
SELECT bdr.alter_node_set_conflict_resolver('node1',
    'target_column_missing', 'ignore');
```

Data loss and divergence risk

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` exists only on node 1.

If the conflict resolver is set to `ignore`, then there can be rows on node 1 where `c` isn't null, for example, `(pk=1, col=100)`. That row is replicated to node 2, and the value in column `c` is discarded, for example, `(pk=1)`.

If column `c` is then added to the table on node 2, it is at first 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.

The default `ignore_if_null` resolver isn't affected by this risk because any row replicated to node 2 has `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. This practice helps to ensure that you detect and fix any divergence.

Terminology of row-types

We use these row-types:

- `SOURCE_OLD` is the row before update, that is, the key.
- `SOURCE_NEW` is the new row coming from another node.
- `TARGET` is the row that exists on the node already, that is, the conflicting row.

Conflict triggers

Conflict triggers execute when a conflict is detected by BDR. They decide what happens when the conflict has occurred.

- If the trigger function returns a row, the action is applied to the target.
- If the trigger function returns a NULL row, the action is skipped.

For example, if the trigger is called for a `DELETE`, the trigger returns NULL if it wants to skip the `DELETE`. If you want the `DELETE` to proceed, then return a row value: either `SOURCE_OLD` or `TARGET` works. 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 can perform other SQL actions as it chooses, but those actions are only applied locally, not replicated.

When a real data conflict occurs between two or more nodes, two or more concurrent changes are occurring. When we apply those changes, the conflict resolution occurs independently on each node. This means the conflict resolution occurs once on each node and can occur with a significant time difference between them. As a result, communication between the multiple executions of the conflict trigger isn't possible. 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 occurs. Technical Support recommends that you formally test all conflict triggers using the isolationtester tool supplied with BDR.

!!! Warning - You can specify multiple conflict triggers on a single table, but they must match a distinct event. That is, each conflict must match only a single conflict trigger. - We don't recommend multiple triggers matching the same event on the same table. They might result in inconsistent behavior and will not be allowed in a future release.

If the same conflict trigger matches more than one event, you can use the `TG_OP` variable in 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, you can call a conflict trigger even when only one change is occurring. Since, in this case, there's no real conflict, this conflict detection mechanism can generate false-positive conflicts. The conflict trigger must handle all of those identically.

In some cases, timestamp conflict detection doesn't 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` don't see any conflict. If no conflict is seen, the conflict trigger are never called. In the same situation but using row version conflict detection, a conflict is seen, which a conflict trigger can then handle.

The trigger function has access to additional state information as well as the data row involved in the conflict, depending on the operation type:

- On `INSERT`, conflict triggers can access the `SOURCE_NEW` row from the source and `TARGET` row.
- On `UPDATE`, conflict triggers can access the `SOURCE_OLD` and `SOURCE_NEW` row from the source and `TARGET` row.
- On `DELETE`, conflict triggers can access the `SOURCE_OLD` row from the source and `TARGET` row.

You can use the function `bdr.trigger_get_row()` 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 errors due to a difference in timing of execution. Hence, avoid updating primary keys in conflict triggers.

Transform triggers

These triggers are similar to conflict triggers, except they are executed for every row on the data stream against the specific table. The behavior of return values and the exposed variables is similar, but transform triggers execute before a target row is identified, so there is no `TARGET` row.

You can specify multiple transform triggers on each table in BDR. 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 the further action taken:

- If the trigger function returns a row, it's applied to the target.
- If the trigger function returns a `NULL` row, there's no further action to perform. Unexecuted triggers never execute.
- The trigger function can 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 can access the `SOURCE_NEW` row from the source.
- On `UPDATE`, transform triggers can access the `SOURCE_OLD` and `SOURCE_NEW` row from the source.
- On `DELETE`, transform triggers can access the `SOURCE_OLD` row from the source.

You can use the function `bdr.trigger_get_row()` to retrieve `SOURCE_OLD` or `SOURCE_NEW` rows. `TARGET` row isn't 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:

- A transform trigger gets called for every incoming change. BEFORE triggers aren't called at all for `UPDATE` and `DELETE` changes if a matching row in a table isn't found.
- Transform triggers are called before partition table routing occurs.
- Transform triggers have access to the lookup key via `SOURCE_OLD`, which isn't available to normal SQL triggers.

Stream triggers variables

Both conflict triggers and transform triggers have access to information about rows and metadata by way of the predefined variables provided by the trigger API and additional information functions provided by BDR.

In PL/pgSQL, you can use the predefined variables that follow.

TG_NAME

Data type name. This variable contains the name of the trigger actually fired. The actual trigger name has a '_bdrt' or '_bdrc' suffix (depending on trigger type) compared to the name provided during trigger creation.

TG_WHEN

Data type text. This variable says `BEFORE` for both conflict and transform triggers. You can get the stream trigger type by calling the `bdr.trigger_get_type()` information function. See bdr.trigger_get_type.

TG_LEVEL

Data type text: a string of `ROW`.

TG_OP

Data type text: a string of `INSERT`, `UPDATE`, or `DELETE` identifying the operation for which the trigger was fired.

TG_RELID

Data type oid: the object ID of the table that caused the trigger invocation.

TG_TABLE_NAME

Data type name: the name of the table that caused the trigger invocation.

TG_TABLE_SCHEMA

Data type name: the name of the schema of the table that caused the trigger invocation. For partitioned tables, this is the name of the root table.

TG_NARGS

Data type integer: the number of arguments given to the trigger function in the `bdr.create_conflict_trigger()` or `bdr.create_transform_trigger()` statement.

TG_ARGV[]

Data type array of text: the arguments from the `bdr.create_conflict_trigger()` or `bdr.create_transform_trigger()` statement. The index counts from 0. Invalid indexes (less than 0 or greater than or equal to `TG_NARGS`) result in a `NULL` value.

Information functions

bdr.trigger_get_row

This function returns the contents of a trigger row specified by an identifier as a `RECORD`. This function returns `NULL` if called inappropriately, that is, 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 description 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 isn't available, returns `NULL`. Always returns `NULL` for row identifier `SOURCE_OLD`.

Synopsis

```
bdr.trigger_get_committs(row_id text)
```

Parameters

- `row_id` — Identifier of the row. Can be any of `SOURCE_NEW`, `SOURCE_OLD`, and `TARGET`, depending on trigger type and operation (see description 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 isn't available, returns `NULL`. Always returns `NULL` for `SOURCE_OLD` and `SOURCE_NEW` row identifiers.

Available only for conflict triggers.

Synopsis

```
bdr.trigger_get_xid(row_id text)
```

Parameters

- `row_id` — Identifier of the row. Can be any of `SOURCE_NEW`, `SOURCE_OLD`, and `TARGET`, depending on trigger type and operation (see description of individual trigger types).

bdr.trigger_get_type

This function returns the current trigger type, which can be `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. Otherwise, returns `NULL`.

See 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 isn't valid (which means the row originated locally), returns the node id of the source or target node, depending on the trigger row argument. Always returns `NULL` for row identifier `SOURCE_OLD`. You can use this function to define conflict triggers to always favor a trusted source node.

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 description of individual trigger types).

bdr.ri_fkey_on_del_trigger

When called as a BEFORE trigger, this function uses FOREIGN KEY information to avoid FK anomalies.

Synopsis

```
bdr.ri_fkey_on_del_trigger()
```

Row contents

The `SOURCE_NEW`, `SOURCE_OLD`, and `TARGET` contents depend on the operation, REPLICA IDENTITY setting of a table, and the contents of the target table.

The TARGET row is available only in conflict triggers. The TARGET row contains data only if a row was found when applying `UPDATE` or `DELETE` in the target table. If the row isn't found, the TARGET is `NULL`.

Triggers notes

Execution order for triggers:

- Transform triggers — Execute once for each incoming row on the target.
- Normal triggers — Execute once per row.
- Conflict triggers — Execute once per row where a conflict exists.

Stream triggers manipulation interfaces

You can create stream triggers only on tables with `REPLICA IDENTITY FULL` or tables without any columns to which `TOAST` applies.

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` — Relation to fire this trigger for.
- `function` — The function to execute.
- `args` — Optional. Specifies the array of parameters the trigger function receives on 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 takes a global DML lock on the relation on which the trigger is being created.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

Similar 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 Security and roles.

bdr.create_transform_trigger

This function creates a 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` — Relation to fire this trigger for.
- `function` — The function to execute.
- `args` — Optional. Specify array of parameters the trigger function receives on 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 takes a global DML lock on the relation on which the trigger is being created.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. 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 Security and roles.

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` — The relation the trigger is defined for.
- `ifexists` — When set to `true`, this function ignores 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 takes a global DML lock on the relation on which the trigger is being created.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

Only the owner of the `relation` can execute the `bdr.drop_trigger` function.

Stream triggers examples

A conflict trigger that provides similar behavior as the `update_if_newer` conflict resolver:

```
CREATE OR REPLACE FUNCTION update_if_newer_trig_func
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF (bdr.trigger_get_committs('TARGET') >
        bdr.trigger_get_committs('SOURCE_NEW')) THEN
    RETURN TARGET;
    ELSIF
        RETURN SOURCE;
    END IF;
END;
$$;
```

A conflict trigger that 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 that 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;
$$;
```

This example 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 three or more nodes.

```
CREATE OR REPLACE FUNCTION test_conflict_trigger()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    SOURCE  record;
    TARGET  record;

    TRUSTED_NODE    bigint;
    SOURCE_NODE     bigint;
    TARGET_NODE     bigint;
BEGIN
    TARGET := bdr.trigger_get_row('TARGET');
    IF (TG_OP = 'DELETE')
        SOURCE := bdr.trigger_get_row('SOURCE_OLD');
    ELSE
        SOURCE := bdr.trigger_get_row('SOURCE_NEW');
    END IF;

    TRUSTED_NODE := current_setting('customer.trusted_node_id');

    SOURCE_NODE := bdr.trigger_get_origin_node_id('SOURCE_NEW');
    TARGET_NODE := bdr.trigger_get_origin_node_id('TARGET');

    IF (TRUSTED_NODE = SOURCE_NODE) THEN
        RETURN SOURCE;
    ELSIF (TRUSTED_NODE = TARGET_NODE) THEN
        RETURN TARGET;
    ELSE
        RETURN NULL; -- do not apply the change
    END IF;
END;
$$;
```

6.20 Explicit two-phase commit (2PC)

An application can explicitly opt to use two-phase commit with BDR. See Distributed Transaction Processing: The XA Specification.

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

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

BDR supports explicit external 2PC using the `PREPARE TRANSACTION` and `COMMIT PREPARED`/`ROLLBACK PREPARED` commands. Externally, a EDB Postgres Distributed cluster appears to be a single resource manager to the transaction manager for a single session.

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

Using `bdr.commit_scope` set to `local` might not seem to make sense with explicit two-phase commit, but the option is offered to allow you to control the tradeoff between transaction latency and robustness.

Explicit two-phase commit doesn't work with either CAMO or the global commit scope. Future releases might enable this combination.

Use

Two-phase commits with a local commit scope work exactly like standard PostgreSQL. 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';
```

6.21 Catalogs and views

Catalogs and views are presented here in alphabetical order.

User-visible catalogs and views

`bdr.conflict_history`

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

Access to this table is possible by any table owner, who 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 `bdr.subscription` table
local_xid	xid	Local transaction of the replication process at the time of conflict
local_lsn	pg_lsn	Local LSN 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)
reloid	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 `local_tuple` is set and the row isn't frozen)
local_tuple_node_id	oid	Node that produced the local conflicting row (if `local_tuple` 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 `local_tuple` 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

Name	Type	Description
conflict_type	text	Type of conflict
conflict_resolution	text	Resolution adopted

`bdr.consensus_kv_data`

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

`bdr.consensus_kv_data` Columns

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

`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	timestamptz	Decision time

`bdr.crdt_handlers`

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

`bdr.crdt_handlers` Columns

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

`bdr.ddl_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 `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

Name	Type	Description
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
response_message_type_no	oid	ID of the BDR message type for the response
response_message_type	text	Name of the BDR message type for the response
response_payload	text	BDR message payload for the response
response_errcode_no	text	SQLSTATE for the response
response_errcode	text	Error code for the response
response_message	text	Error message for the response

`bdr.global_consensus_response_journal`

This catalog table collects all the responses to the Raft messages that were received while managing global consensus.

As for the `bdr.global_consensus_journal` catalog, the payload is stored in a binary-encoded format, which can be decoded with the `bdr.decode_message_payload()` function. See the `bdr.global_consensus_journal_details` view for more details.

`bdr.global_consensus_response_journal` columns

Name	Type	Description
log_index	int8	ID of the journal entry
res_status	oid	Status code for the response
res_payload	bytea	Payload for the response
trace_context	bytea	Trace context for the response

`bdr.global_lock`

This catalog table stores the information needed for recovering the global lock state on server restart.

For monitoring usage, the `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
ddl_epoch	int8	DDL epoch for the lock
origin_node_id	oid	OID of the node where the global lock has originated
lock_type	oid	Type of the lock (DDL or DML)
nspname	name	Schema name for the locked relation
relname	name	Relation name for the locked relation
groupid	oid	OID of the top level group (for Advisory locks)

Name	Type	Description
key1	integer	First 32-bit key or lower order 32-bits of 64-bit key (for advisory locks)
key2	integer	Second 32-bit key or higher order 32-bits of 64-bit key (for advisory locks)
key_is_bigint	boolean	True if 64-bit integer key is used (for advisory locks)

`bdr.global_locks`

A view containing active global locks on this node. The `bdr.global_locks` view exposes BDR's shared-memory lock state tracking, giving administrators greater insight into BDR's global locking activity and progress.

See Monitoring global locks for more information about global locking.

`bdr.global_locks` columns

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

Column details:

- `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
log_index	int8	ID of the journal entry
log_term	int8	Raft term
snapshot	bytea	Raft snapshot data

`bdr.local_consensus_state`

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

`bdr.local_consensus_state` columns

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

`bdr.local_node`

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 `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_nspname	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
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 EDB Postgres Distributed cluster.

bdr.node_conflict_resolvers

Currently configured conflict resolution for all known conflict types.

bdr.node_conflict_resolvers columns

Name	Type	Description
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_ext	oid	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
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

Name	Type	Description
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.

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

`bdr.node_pre_commit`

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

`bdr.node_pre_commit` columns

Name	Type	Description
origin_node_id	oid	OID of the node where the transaction executed
origin_xid	oid	Transaction ID on the remote origin node
decision	char	'c' for commit, 'a' for abort

Name	Type	Description
local_xid	xid	Transaction ID on the local node
commit_ts	timestamptz	Commit timestamp of the transaction
decision_ts	timestamptz	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
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)

Name	Type	Description
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 pglogical_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 `synchronous_standby_names`)
client_addr	inet	IP address of the client connection
client_hostname	text	Hostname of the client connection
client_port	int4	Port of the client connection
backend_start	timestamptz	When the connection started
state	text	State of the replication (catchup, streaming, ...) or 'disconnected' if offline
sent_lsn	pg_lsn	Latest sent position
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)
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	timestamptz	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. Possible values: A - Table sync D - DDL S - Sequence T - Truncate Q - SQL statement
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

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 win BDR.

`bdr.schema_changes` columns

Name	Type	Description
schema_changes_ts	timestampstz	The ID of the trigger
schema_changes_change	char	A flag of change type
schema_changes_classid	oid	Class ID
schema_changes_objectid	oid	Object ID
schema_changes_subid	smallint	The subscription
schema_changes_descr	text	The object changed
schema_changes_addrnames	text[]	Location of schema change

`bdr.sequence_alloc`

A view to see the allocation details for galloc 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	timestamptz	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	timestampstz	The ID of the trigger
schema_changes_change	char	A flag of change type
schema_changes_classid	oid	Class ID
schema_changes_objectid	oid	Object ID
schema_changes_subid	smallint	The subscription
schema_changes_descr	text	The object changed
schema_changes_addrnames	text[]	Location of schema change

`bdr.sequence_alloc`

A view to see the sequences allocated.

`bdr.sequence_alloc` columns

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

`bdr.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', 'galloc')

`bdr.stat_activity`

Dynamic activity for each backend or worker process.

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

`bdr.stat_relation`

Apply statistics for each relation. 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
ntruncate	bigint	Number of truncates replicated for the relation
shared_blks_hit	bigint	Total number of shared block cache hits for the relation
shared_blks_read	bigint	Total number of shared blocks read for the relation
shared_blks_dirtied	bigint	Total number of shared blocks dirtied for the relation
shared_blks_written	bigint	Total number of shared blocks written for the relation
blk_read_time	double precision	Total time spent reading blocks for the relation, in milliseconds (if `track_io_timing` is enabled, otherwise zero)
blk_write_time	double precision	Total time spent writing blocks for the relation, in milliseconds (if `track_io_timing` is enabled, otherwise zero)
lock_acquire_time	double precision	Total time spent acquiring locks on the relation, in milliseconds (if `bdr.track_apply_lock_timing` 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 `skip_transaction` 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)

Column	Type	Description
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
shared_blks_hit	bigint	Total number of shared block cache hits by the subscription
shared_blks_read	bigint	Total number of shared blocks read by the subscription
shared_blks_dirtied	bigint	Total number of shared blocks dirtied by the subscription
shared_blks_written	bigint	Total number of shared blocks written by the subscription
blk_read_time	double precision	Total time the subscription spent reading blocks, in milliseconds (if `track_io_timing` is enabled, otherwise zero)
blk_write_time	double precision	Total time the subscription spent writing blocks, in milliseconds (if `track_io_timing` is enabled, otherwise zero)
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

Name	Type	Description
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	timestamptz	LSN of last transaction replayed on this subscription that's flushed durably to disk
last_xact_replay_timestamp	timestamptz	Timestamp of last transaction replayed on this subscription

`bdr.replication_status`

This view shows incoming replication status between the local node and all other nodes in the EDB Postgres Distributed cluster. 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

Name	Type	Description
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	timestamptz	Last commit timestamp processed by this worker if any
worker_local_timestamp	timestamptz	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

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

Name	Type	Description
replay_lag	interval	Approximate lag time for reported replay
sent_lag_bytes	int8	Bytes difference between sent_lsn and current WAL write position
write_lag_bytes	int8	Bytes difference between write_lsn and current WAL write position
flush_lag_bytes	int8	Bytes difference between flush_lsn and current WAL write position
replay_lag_byte	int8	Bytes difference between replay_lsn and current WAL write position

`bdr.group_subscription_summary`

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

`bdr.group_subscription_summary` columns

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

`bdr.group_versions_details`

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

`bdr.group_versions_details` columns

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

Internal catalogs and views

`bdr.ddl_epoch`

An internal catalog table holding state per DDL epoch.

`bdr.ddl_epoch` columns

Name	Type	Description
ddl_epoch	int8	Monotonically increasing epoch number
origin_node_id	oid	Internal node ID of the node that requested creation of this epoch
epoch_consume_timeout	timestamptz	Timeout of this epoch

Name	Type	Description
epoch_consumed	boolean	Switches to true as soon as the local node has fully processed the epoch
epoch_consumed_lsn	boolean	LSN at which the local node has 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	timestamptz	Point in time the current node observed the state change

6.22 BDR system functions

Perform BDR management primarily by using functions you call from SQL. All functions in BDR are exposed in the `bdr` schema. Schema qualify any calls to these functions instead of putting `bdr` in the `search_path`.

Version information functions

bdr.bdr_version

This function retrieves the textual representation of the BDR version currently in use.

bdr.bdr_version_num

This function retrieves the BDR version number 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 returns the version number consisting of major version, minor version, and patch release into a single numerical value:

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

System information functions

bdr.get_relation_stats

Returns the relation information.

bdr.get_subscription_stats

Returns the current subscription statistics.

System and progress information parameters

BDR exposes some parameters that you can query using `SHOW` in `psql` or using `PQparameterStatus` (or equivalent) from a client application.

bdr.local_node_id

When you initialize a session, this is set to the node id the client is connected to. This allows an application to figure out the node it's connected to, even behind a transparent proxy.

It's also used with CAMO.

bdr.last_committed_lsn

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

transaction_id

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

bdr.is_node_connected

Synopsis

```
bdr.is_node_connected(node_name name)
```

Returns boolean by checking if the walsender for a given peer is active on this node.

bdr.is_node_ready

Synopsis

```
bdr.is_node_ready(node_name name, span interval DEFAULT NULL)
```

Returns boolean by checking if the lag is lower than the given span or lower than the `bdr.global_commit_timeout` otherwise.

Consensus function

bdr.consensus_disable

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

!!! Warning Disabling consensus disables some features of BDR and affects availability of the EDB Postgres Distributed cluster if left disabled for a long time. Use this function only when working with Technical Support.

bdr.consensus_enable

Reenabled 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 you can override that by passing an explicit `version` number.

The exporting node doesn't have to be the current Raft leader, and it doesn't need to be completely up to date with the latest state on the leader. However, `bdr.consensus_snapshot_import()` might not accept such a snapshot.

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

The generated snapshot might be passed to `bdr.consensus_snapshot_import()` on any other nodes in the same BDR node group that's 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 fail or time out.
- galloc sequences don't get new values.
- Eager and CAMO transactions pause or error.
- Other functionality that needs the distributed consensus system is disrupted. The required downtime is generally very brief.

Depending on the use case, it might 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 doesn't require you to stop the consensus worker.

bdr.consensus_snapshot_import

Synopsis

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

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

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 a known good state in case of corruption or user error.

You can import the snapshot if the importing node's `apply_index` is less than or equal to the snapshot-exporting node's `commit_index` when the snapshot was generated. (See `bdr.get_raft_status()`.) A node that can't accept the snapshot because its log is already too far ahead raises an error and makes no changes. The imported snapshot doesn't have to be completely up to date, as once the snapshot is imported the node fetches 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 function to force the local node to generate a new Raft snapshot by running:

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

This approach might also truncate the 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 that was exported by `bdr.consensus_snapshot_export()`. The snapshot header contains the version with which it was generated and the node tries to verify it against the same version.

The snapshot might have been exported on the same node or any other node in the cluster. If the node verifying the snapshot doesn't support the version of the exported snapshot, then an error is 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 is internally forwarded to the current leader to transfer the leadership to the designated node. The designated node must be an ACTIVE BDR node with full voting rights.

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

If `wait_for_completion` is `true`, then the function waits until the given node becomes the new leader and possibly waits infinitely if the requested node fails to become Raft leader (for example, due to network issues). We therefore recommend that you always set a `statement_timeout` with `wait_for_completion` to prevent an infinite loop.

Utility functions

bdr.wait_slot_confirm_lsn

Allows you to wait until the last write on this session was replayed to one or all nodes.

Waits until a slot passes a certain LSN. If no position is supplied, the current write position is used on the local node.

If no slot name is passed, it waits until all BDR slots pass the LSN.

The function polls every 1000 ms for changes from other nodes.

If a slot is dropped concurrently, the wait ends for that slot. If a node is currently down and isn't updating its slot, then the wait continues. You might want 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 returns 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 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 which the peer waits. You can use this 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 were sent. In this case, transactions that are still in transit or buffered on the sender side aren't 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 to wait 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 by way of `bdr.last_committed_lsn` from the peer node.

bdr.get_node_sub_receive_lsn

You can use this function on a subscriber to get the last LSN that was received from the given origin. It can be either unfiltered or filtered to take into account only relevant LSN increments for transactions to be applied.

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 that's the source of the replication stream whose LSN is being retrieved.
- `committed` —; The default (true) makes this function take into account only commits of transactions received rather than the last LSN overall. This includes actions that have no effect on the subscriber node.

bdr.get_node_sub_apply_lsn

You can use this function on a subscriber to get the last LSN that was 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 that's the source of the replication stream whose LSN is being retrieved.

bdr.run_on_all_nodes

Function to run a query on all nodes.

!!! Warning This function runs 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. Use caution when granting privileges to this function.

Synopsis

```
bdr.run_on_all_nodes(query text)
```

Parameters

- `query` — Arbitrary query to execute.

Notes

This function connects to other nodes and executes the query, returning a result from each of them in JSON format.

Multiple rows might be returned from each node, encoded as a JSON array. Any errors, such as being unable to connect because a node is down, are shown in the response field. No explicit statement_timeout or other runtime parameters are set, so defaults are used.

This function doesn't 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 commands 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 might be blocked in a future release.

Example

It's useful to use this function in monitoring, for example, as in the following query:

```
SELECT bdr.run_on_all_nodes($$
    SELECT local_slot_name, origin_name, target_name, replay_lag_size
      FROM bdr.node_slots
     WHERE origin_name IS NOT NULL
$$);
```

This query returns something like this on a two-node cluster:

```
[
    {
        "dsn": "host=node1 port=5432 dbname=bdrdb user=postgres ",
        "node_id": "2232128708",
        "response": {
            "command_status": "SELECT 1",
            "command_tuples": [
                {
                    "origin_name": "node1",
                    "target_name": "node2",
                    "local_slot_name": "bdr_bdrdb_bdrgroup_node2",
                    "replay_lag_size": "0 bytes"
                }
            ]
        },
        "node_name": "node1"
    },
    {
        "dsn": "host=node2 port=5432 dbname=bdrdb user=postgres ",
        "node_id": "2058684375",
        "response": {
            "command_status": "SELECT 1",
            "command_tuples": [
                {
                    "origin_name": "node2",
                    "target_name": "node1",
                    "local_slot_name": "bdr_bdrdb_bdrgroup_node1",
                    "replay_lag_size": "0 bytes"
                }
            ]
        },
```

```
        "node_name": "node2"
    }
]
```

bdr.run_on_nodes

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

!!! Warning This function runs an arbitrary query on remote nodes with the privileges of the user used for the internode connections as specified in the node's DSN. Use caution 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 is executed.
- `query` — Arbitrary query to execute.

Notes

This function connects to other nodes and executes the query, returning a result from each of them in JSON format. Multiple rows can be returned from each node, encoded as a JSON array. Any errors, such as being unable to connect because a node is down, are shown in the response field. No explicit statement_timeout or other runtime parameters are set, so defaults are used.

This function doesn't 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 commands 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 might be blocked in a future release.

bdr.run_on_group

Function to run a query on a group of nodes.

!!! Warning This function runs an arbitrary query on remote nodes with the privileges of the user used for the internode connections as specified in the node's DSN. Use caution 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 is executed.
- `query` — Arbitrary query to execute.

Notes

This function connects to other nodes and executes the query, returning a result from each of them in JSON format. Multiple rows can be returned from each node, encoded as a JSON array. Any errors, such as being unable to connect because a node is down, are shown in the response field. No explicit statement_timeout or other runtime parameters are set, so defaults are used.

This function doesn't 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 commands 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 might be blocked in a future release.

bdr.global_lock_table

This function acquires 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 lock.

Notes

This function acquires 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 lower.

bdr.wait_for_xid_progress

You can use this function 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 originated.

- `origin_topxid` — XID of the transaction.
- `allnodes` — If `true` then wait for the transaction to progress on all nodes. Otherwise wait only for the current node.

Notes

You can use the function only for those transactions that replicated a DDL command because only those transactions are tracked currently. If a wrong `origin_node_id` or `origin_topxid` is supplied, the function might wait forever or until `statement_timeout` occurs.

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

Example

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

Global advisory locks

BDR supports global advisory locks. These locks are similar to the advisory locks available in PostgreSQL except that the advisory locks supported by BDR are global. They follow semantics similar to DDL locks. So an advisory lock is obtained by majority consensus and 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 only EXCLUSIVE locks are supported. 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's explicitly released before the end of the transaction. In this case, it becomes available as soon as it's released. Session-level advisory locks aren't currently supported.

Global advisory locks are reentrant. So if the same resource is locked three times, you must then unlock it three times for

it to be released for use in other sessions.

bdr.global_advisory_lock

This function acquires an EXCLUSIVE lock on the provided object. If the lock isn't available, then it waits 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 releases a previously acquired lock on the application-defined source. The lock must have been 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 an advisory lock is acquired.

Synopsis

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

Parameters

- `key1` — First part of the composite key.
- `key2` — Second part of the composite key.

Monitoring functions

bdr.monitor_group_versions

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

Synopsis

```
bdr.monitor_group_versions()
```

Notes

This function returns a record with fields `status` and `message`, as explained in Monitoring.

This function calls `bdr.run_on_all_nodes()`.

bdr.monitor_group_raft

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

Synopsis

```
bdr.monitor_group_raft()
```

Notes

This function returns a record with fields `status` and `message`, as explained in Monitoring.

This function calls `bdr.run_on_all_nodes()`.

bdr.monitor_local_replslots

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

Synopsis

```
bdr.monitor_local_replslots()
```

Notes

This function returns a record with fields `status` and `message`, as explained in Monitoring replication slots.

bdr.wal_sender_stats

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

Synopsis

```
bdr.wal_sender_stats() &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 are `NULL` if `is_using_lcr` is `FALSE`.
- `decoder_slot_name` — The name of the decoder replication slot.
- `lcr_file_name` — The name of the current LCR file.

bdr.get_decoding_worker_stat

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

Synopsis

```
bdr.get_decoding_worker_stat()  &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 up to which the decoding worker read transactional logs.
- `waiting` — Whether the decoding worker is waiting for new WAL.
- `waiting_for_lsn` — The LSN of the next expected WAL.

Notes

For further details, see Monitoring WAL senders using LCR.

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.

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. Exists only for backward 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 added 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's 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.

Use this function instead of standard `nextval` in queries that 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

Displays 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

Returns origin id of a given transaction.

Synopsis

```
bdr.pg_xact_origin(xmin xid)
```

Parameters

- `xid` — Transaction id whose origin is returned,

bdr.difference_fix_origin_create

Creates a replication origin with a given name passed as an argument but adding a `bdr_` prefix. It returns the internal id of the origin. This performs the 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 and returns void. This function performs the same functionality as `pg_replication_origin_session_setup()` except that this function requires bdr_superuser rather than postgres superuser permissions. The earlier form of the function, `bdr.difference_fix_session_setup(text)`, was 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 function has the same functionality as `pg_replication_origin_session_reset()` except this function 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 committed at LSN '0/0' and timestamp '2000-01-01'. This function has 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 or resync the relation data.
- `relation` — The relation to copy from the remote node.

Notes

This function 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.

The following are supported:

- Resynchronizing partitioned tables with identical partition definitions
- Resynchronizing partitioned table to nonpartitioned table and vice versa
- Resynchronizing referenced tables by temporarily dropping and recreating foreign key constraints

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

Furthermore, it supports resynchronization of tables 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 executed only 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 `ttl`. If `ttl` is `NULL`, then this returns `infinity`. If the value was deleted, it returns `-infinity`.

Synopsis

```
bdr.consensus_kv_store(key text, value jsonb,
        prev_value jsonb DEFAULT NULL, ttl int DEFAULT NULL)
```

Parameters

- `key` — An arbitrary unique key to insert, update, or delete.
- `value` — JSON value to store. If NULL, any existing record is deleted.
- `prev_value` — If set, the write operation is done only if the current value is equal to `prev_value`.
- `ttl` — Time to live of the new value, in milliseconds.

Notes

This is an internal function, mainly used by HARP.

!!! Warning Don't use this function in 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 Don't use this function in 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, fix such problems by making changes to the target side. `CREATE` any missing table that's blocking replication, `CREATE` a needed role, `GRANT` a necessary permission, and so on. But occasionally a problem can't be fixed that way and it might 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 that follows, or peek the change stream with the logical decoding functions.

Unless a transaction made only one change, you often need to manually apply the transaction's effects on the target side, so it's important to save the problem transaction whenever possible, as shown in the examples that follow.

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 a convenience function. It does do a faster skip, although it might bypass some kinds of errors in logical decoding.

This function works only on disabled subscriptions.

The usual sequence of steps is:

1. Identify the problem subscription and LSN of the problem commit.
2. Disable the subscription.
3. Save a copy of the transaction using `pg_catalog.pg_logical_slot_peek_changes` on the source node, if possible.
4. `bdr.alter_subscription_skip_changes_upto` on the target node.
5. Apply repaired or equivalent changes on the target manually, if necessary.

6. Reenable the subscription.

!!! Warning It's easy to make problems worse when using this function. Don't do anything unless you're certain 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 can't 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 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, you have the information you 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');
```

You can't skip only parts of the transaction: it's all or nothing. So we strongly recommend that you save a record of it by copying it out on the provider side first, using the subscription's slot name.

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

This example is broken into multiple lines for readability, but issue it in a single line. `\copy` doesn't support multi-line commands.

You can skip the change by changing `peek` to `get`, 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');
```

You can apply the same changes (or repaired versions of them) manually to the target node, using the dumped

transaction contents as a guide.

Finally, reenable the subscription:

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

7 High Availability Routing for Postgres (HARP)

High Availability Routing for Postgres (HARP) is new approach for managing high availabiliity for EDB Postgres Distributed clusters versions 3.6 or later. All application traffic within a single location (data center or region) is routed to only one BDR node at a time in a semi-exlusive manner. This node, designated the lead master, acts as the principle write target to reduce the potential for data conflicts.

HARP leverages a distributed consensus model to determine availability of the BDR nodes in the cluster. On failure or unavailability of the lead master, HARP elects a new lead master and redirects application traffic.

Together with the core capabilities of the BDR extension, this mechanism of routing application traffic to the lead master node enables fast failover and switchover without risk of data loss.

The importance of quorum

The central purpose of HARP is to enforce full quorum on any EDB Postgres Distriibuted cluster it manages. Quorum is a term applied to a voting body that mandates a certain minimum of attendees are available to make a decision. More simply: majority rules.

For any vote to end in a result other than a tie, an odd number of nodes must constitute the full cluster membership. Quorum, however, doesn't strictly demand this restriction; a simple majority is enough. This means that in a cluster of N nodes, quorum requires a minimum of N/2+1 nodes to hold a meaningful vote.

All of this ensures the cluster is always in agreement regarding the node that is "in charge." For a EDB Postgres Distributed cluster consisting of multiple nodes, this determines the node that is the primary write target. HARP designates this node as the lead master.

Reducing write targets

The consequence of ignoring the concept of quorum, or not applying it well enough, can lead to a "split brain" scenario where the "correct" write target is ambiguous or unknowable. In a standard EDB Postgres Distributed cluster, it's important that only a single node is ever writable and sending replication traffic to the remaining nodes.

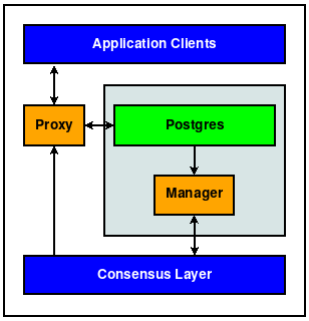
Even in multi-master-capable approaches such as BDR, it can be helpful to reduce the amount of necessary conflict management to derive identical data across the cluster. In clusters that consist of multiple BDR nodes per physical location or region, this usually means a single BDR node acts as a "leader" and remaining nodes are "shadow." These shadow nodes are still writable, but writing to them is discouraged unless absolutely necessary.

By leveraging quorum, it's possible for all nodes to agree on the exact Postgres node to represent the entire cluster or a local BDR region. Any nodes that lose contact with the remainder of the quorum, or are overruled by it, by definition can't become the cluster leader.

This restriction prevents split-brain situations where writes unintentionally reach two Postgres nodes. Unlike technologies such as VPNs, proxies, load balancers, or DNS, you can't circumvent a quorum-derived consensus by misconfiguration or network partitions. So long as it's possible to contact the consensus layer to determine the state of the quorum maintained by HARP, only one target is ever valid.

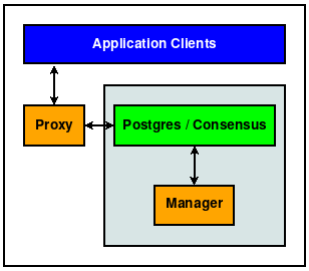
Basic architecture

The design of HARP comes in essentially two parts, consisting of a manager and a proxy. The following diagram describes how these interact with a single Postgres instance:



The consensus layer is an external entity where Harp Manager maintains information it learns about its assigned Postgres node, and HARP Proxy translates this information to a valid Postgres node target. Because Proxy obtains the node target from the consensus layer, several such instances can exist independently.

While using BDR as the consensus layer, each server node resembles this variant instead:

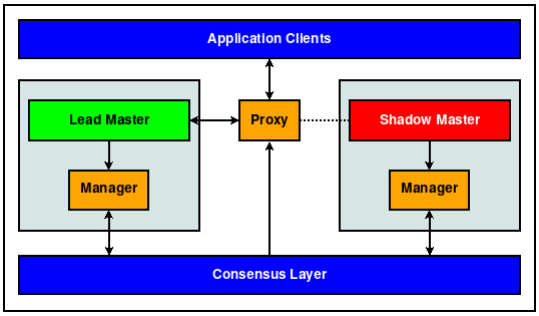


In either case, each unit consists of the following elements:

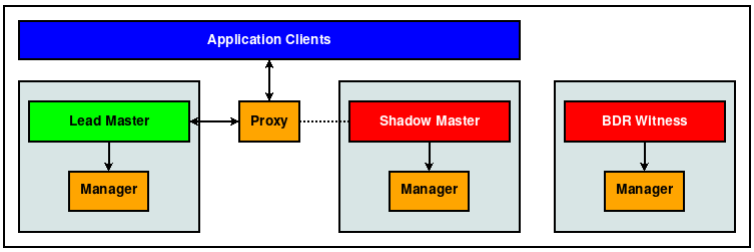
- A Postgres instance
- A consensus layer resource, meant to track various attributes of the Postgres instance
- A HARP Manager process to convey the state of the Postgres node to the consensus layer
- A HARP Proxy service that directs traffic to the proper lead master node, as derived from the consensus layer

Not every application stack has access to additional node resources specifically for the Proxy component, so it can be combined with the application server to simplify the stack.

This is a typical design using two BDR nodes in a single data center organized in a lead master/shadow master configuration:



When using BDR as the HARP consensus layer, at least three fully qualified BDR nodes must be present to ensure a quorum majority. (Not shown in the diagram are connections between BDR nodes.)



How it works

When managing a EDB Postgres Distributed cluster, HARP maintains at most one leader node per defined location. This is referred to as the lead master. Other BDR nodes that are eligible to take this position are shadow master state until they take the leader role.

Applications can contact the current leader only through the proxy service. Since the consensus layer requires quorum agreement before conveying leader state, proxy services direct traffic to that node.

At a high level, this mechanism prevents simultaneous application interaction with multiple nodes.

Determining a leader

As an example, consider the role of lead master in a locally subdivided BDR Always-On group as can exist in a single data center. When any Postgres or Manager resource is started, and after a configurable refresh interval, the following must occur:

1. The Manager checks the status of its assigned Postgres resource.
    - If Postgres isn't running, try again after configurable timeout.
    - If Postgres is running, continue.

2. The Manager checks the status of the leader lease in the consensus layer.
    - If the lease is unclaimed, acquire it and assign the identity of the Postgres instance assigned to this manager. This lease duration is configurable, but setting it too low can result in unexpected leadership transitions.
    - If the lease is already claimed by us, renew the lease TTL.
    - Otherwise do nothing.

A lot more occurs, but this simplified version explains what's happening. The leader lease can be held by only one node, and if it's held elsewhere, HARP Manager gives up and tries again later.

!!! Note Depending on the chosen consensus layer, rather than repeatedly looping to check the status of the leader lease, HARP subscribes to notifications. In this case, it can respond immediately any time the state of the lease changes rather than polling. Currently this functionality is restricted to the etcd consensus layer.

This means HARP itself doesn't hold elections or manage quorum, which is delegated to the consensus layer. A quorum of the consensus layer must acknowledge the act of obtaining the lease, so if the request succeeds, that node leads the cluster in that location.

Connection routing

Once the role of the lead master is established, connections are handled with a similar deterministic result as reflected by HARP Proxy. Consider a case where HARP Proxy needs to determine the connection target for a particular backend resource:

1. HARP Proxy interrogates the consensus layer for the current lead master in its configured location.
2. If this is unset or in transition:
    - New client connections to Postgres are barred, but clients accumulate and are in a paused state until a lead master appears.
    - Existing client connections are allowed to complete current transactions and are then reverted to a similar pending state as new connections.
3. Client connections are forwarded to the lead master.

The interplay shown in this case doesn't require any interaction with either HARP Manager or Postgres. The consensus layer is the source of all truth from the proxy's perspective.

Colocation

The arrangement of the work units is such that their organization must follow these principles:

1. The manager and Postgres units must exist concomitantly in the same node.
2. The contents of the consensus layer dictate the prescriptive role of all operational work units.

This arrangement delegates cluster quorum responsibilities to the consensus layer, while HARP leverages it for critical role assignments and key/value storage. Neither storage nor retrieval succeeds if the consensus layer is inoperable or unreachable, thus preventing rogue Postgres nodes from accepting connections.

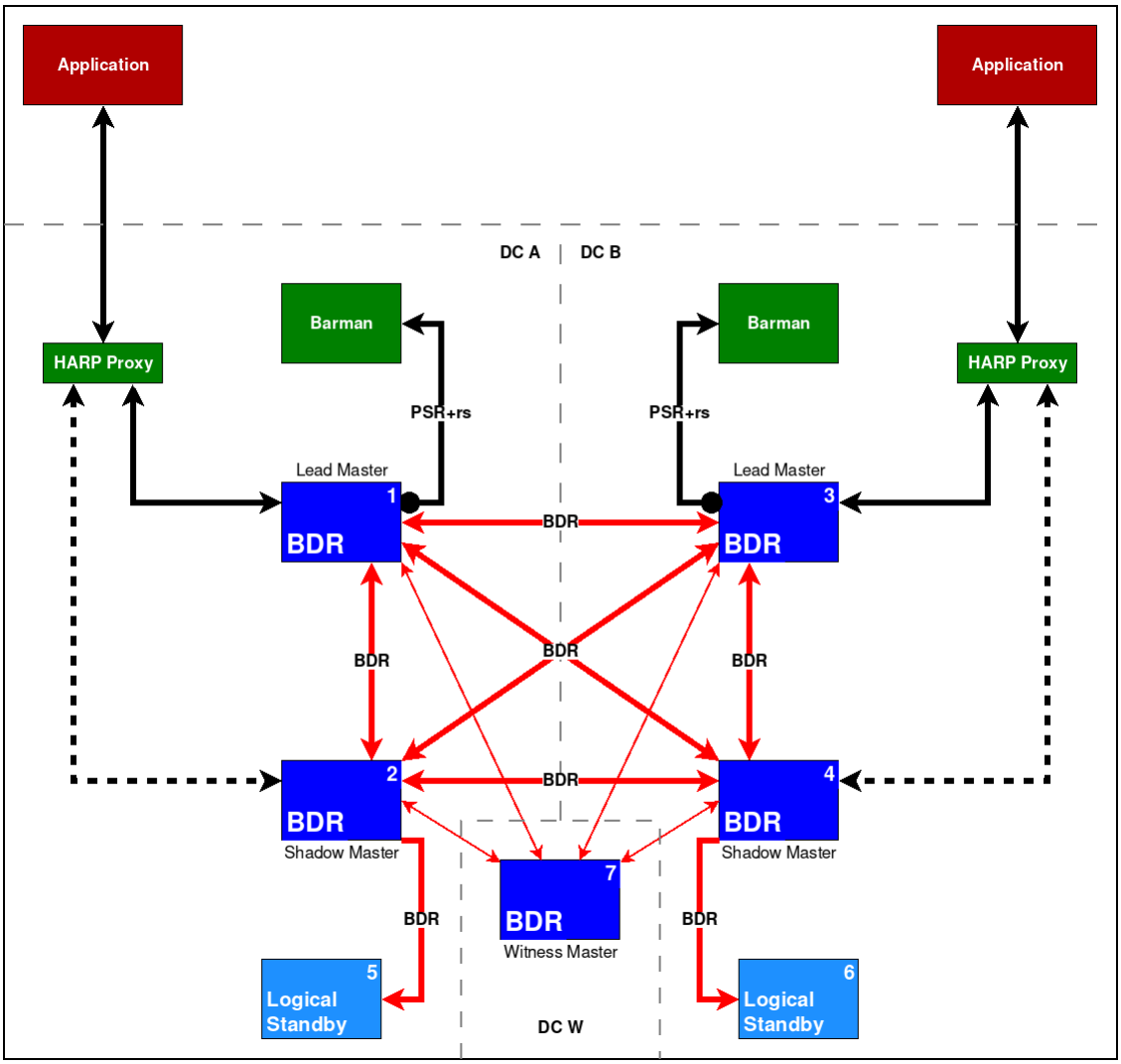
As a result, the consensus layer generally exists outside of HARP or HARP-managed nodes for maximum safety. Our reference diagrams show this separation, although it isn't required.

!!! Note To operate and manage cluster state, BDR contains its own implementation of the Raft Consensus model. You can configure HARP to leverage this same layer to reduce reliance on external dependencies and to preserve server resources. However, certain drawbacks to this approach are discussed in Consensus layer.

Recommended architecture and use

HARP was primarily designed to represent a BDR Always On architecture that resides in two or more data centers and consists of at least five BDR nodes. This configuration doesn't count any logical standby nodes.

The following diagram shows the current and standard representation:



In this diagram, HARP Manager exists on BDR Nodes 1-4. The initial state of the cluster is that BDR Node 1 is the lead master of DC A, and BDR Node 3 is the lead master of DC B.

This configuration results in any HARP Proxy resource in DC A connecting to BDR Node 1 and the HARP Proxy resource in DC B connecting to BDR Node 3.

!!! Note While this diagram shows only a single HARP Proxy per DC, this is an example only and should not be considered a single point of failure. Any number of HARP Proxy nodes can exist, and they all direct application traffic to the same node.

Location configuration

For multiple BDR nodes to be eligible to take the lead master lock in a location, you must define a location in the `config.yml` configuration file.

To reproduce the BDR Always-On reference architecture shown in the diagram, include these lines in the `config.yml` configuration for BDR Nodes 1 and 2:

```
location: dca
```

For BDR Nodes 3 and 4, add:

```
location: dcb
```

This applies to any HARP Proxy nodes that are designated in those respective data centers as well.

7.1 Installation

A standard installation of HARP includes two system services:

- HARP Manager (`harp-manager`) on the node being managed
- HARP Proxy (`harp-proxy`) elsewhere

There are two ways to install and configure these services to manage Postgres for proper quorum-based connection routing.

Software versions

HARP has dependencies on external software. These must fit a minimum version as listed here.

Software	Min version
etcd	3.4
PgBouncer	1.14

TPAExec

The easiest way to install and configure HARP is to use the EDB TPAexec utility for cluster deployment and management. For details on this software, see the TPAexec product page.

!!! Note TPAExec is currently available only through an EULA specifically dedicated to EDB Postgres Distributed cluster deployments. If you can't access the TPAExec URL, contact your sales or account representative.

Configure TPAexec to recognize that cluster routing is managed through HARP by ensuring the TPA `config.yml` file contains these attributes:

```
cluster_vars:
  failover_manager: harp
```

!!! Note Versions of TPAexec earlier than 21.1 require a slightly different approach:

```
```yaml
cluster_vars:
  enable_harp: true
```
```

After this, install HARP by invoking the `tpaexec` commands for making cluster modifications:

```
tpaexec provision ${CLUSTER_DIR}
tpaexec deploy ${CLUSTER_DIR}
```

No other modifications are necessary apart from cluster-specific considerations.

Package installation

Currently CentOS/RHEL packages are provided by the EDB packaging infrastructure. For details, see the HARP product page.

etcd packages

Currently `etcd` packages for many popular Linux distributions aren't available by their standard public repositories. EDB has therefore packaged `etcd` for RHEL and CentOS versions 7 and 8, Debian, and variants such as Ubuntu LTS. You need access to our HARP package repository to use these libraries.

Consensus layer

HARP requires a distributed consensus layer to operate. Currently this must be either `bdr` or `etcd`. If using fewer than three BDR nodes, you might need to rely on `etcd`. Otherwise any BDR service outage reduces the consensus layer to a single node and thus prevents node consensus and disables Postgres routing.

etcd

If you're using `etcd` as the consensus layer, `etcd` must be installed either directly on the Postgres nodes or in a separate location they can access.

To set `etcd` as the consensus layer, include this code in the HARP `config.yml` configuration file:

```
dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379
```

When using TPAExec, all configured etcd endpoints are entered here automatically.

BDR

The `bdr` native consensus layer is available from BDR 3.6.21 and 3.7.3. This consensus layer model requires no supplementary software when managing routing for a EDB Postgres Distributed cluster.

To ensure quorum is possible in the cluster, always use more than two nodes so that BDR's consensus layer remains responsive during node maintenance or outages.

To set BDR as the consensus layer, include this in the `config.yml` configuration file:

```
dcs:
  driver: bdr
  endpoints:
    - host=host1 dbname=bdrdb user=harp_user
    - host=host2 dbname=bdrdb user=harp_user
    - host=host3 dbname=bdrdb user=harp_user
```

The endpoints for a BDR consensus layer follow the standard Postgres DSN connection format.

7.2 Configuring HARP for cluster management

The HARP configuration file follows a standard YAML-style formatting that was simplified for readability. This file is located in the `/etc/harp` directory by default and is named `config.yml`

You can explicitly provide the configuration file location to all HARP executables by using the `-f`/`--config` argument.

Standard configuration

HARP essentially operates as three components:

- HARP Manager
- HARP Proxy
- harpctl

Each of these use the same standard `config.yml` configuration format, which always include the following sections:

- `cluster.name` — The name of the cluster to target for all operations.
- `dcs` — DCS driver and connection configuration for all endpoints.

This means a standard preamble is always included for HARP operations, such as the following:

```
cluster:
  name: mycluster

dcs:
  ...
```

Other sections are optional or specific to the named HARP component.

Cluster name

The `name` entry under the `cluster` heading is required for all interaction with HARP. Each HARP cluster has a name for both disambiguation and for labeling data in the DCS for the specific cluster.

HARP Manager writes information about the cluster here for consumption by HARP Proxy and harpctl. HARP Proxy services direct traffic to nodes in this cluster. The `harpctl` management tool interacts with this cluster.

DCS settings

Configuring the consensus layer is key to HARP functionality. Without the DCS, HARP has nowhere to store cluster metadata, can't hold leadership elections, and so on. Therefore this portion of the configuration is required, and certain elements are optional.

Specify all elements under a section named `dcs` with these multiple supplementary entries:

- `driver`: Required type of consensus layer to use. Currently can be `etcd` or `bdr`. Support for `bdr` as a consensus layer is experimental. Using `bdr` as the consensus layer reduces the additional software for consensus storage but expects a minimum of three full BDR member nodes to maintain quorum during database maintenance.

- `endpoints`: Required list of connection strings to contact the DCS. List every node of the DCS here if possible. This ensures HARP continues to function as long as a majority of the DCS can still operate and be reached by the network.

  Format when using `etcd` as the consensus layer is as follows:

  ```
  dcs:
    endpoints:
      - host1:2379
      - host2:2379
      - host3:2379
  ```

  Format when using the experimental `bdr` consensus layer is as follows:

  ```
  dcs:
  # only DSN format is supported
    endpoints:
      - "host=host1 port=5432 dbname=bdrdb user=postgres"
      - "host=host2 port=5432 dbname=bdrdb user=postgres"
      - "host=host3 port=5432 dbname=bdrdb user=postgres"
  ```

Currently, `bdr` consensus layer requires the first endpoint to point to the local postgres instance.

- `request_timeout`: Time in milliseconds to consider a request as failed. If HARP makes a request to the DCS and receives no response in this time, it considers the operation as failed. This can cause the issue to be logged as an error or retried, depending on the nature of the request. Default: 250.

The following DCS SSL settings apply only when `driver: etcd` is set in the configuration file:

- `ssl`: Either `on` or `off` to enable SSL communication with the DCS. Default: `off`
- `ssl_ca_file`: Client SSL certificate authority (CA) file.
- `ssl_cert_file`: Client SSL certificate file.
- `ssl_key_file`: Client SSL key file.

Example

This example shows how to configure HARP to contact an etcd DCS consisting of three nodes:

```
dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379
```

HARP Manager specific

Besides the generic service options required for all HARP components, Manager needs other settings:

- `log_level`: One of `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `CRITICAL`, which might alter the amount of log output from HARP services.

- `name`: Required name of the Postgres node represented by this Manager. Since Manager can represent only a specific node, that node is named here and also serves to name this Manager. If this is a BDR node, it must match the value used at node creation when executing the `bdr.create_node(node_name, ...)` function and as reported by the `bdr.local_node_summary.node_name` view column. Alphanumeric characters and underscores only.

- `start_command`: This can be used instead of the information in DCS for starting the database to monitor. This is required if using bdr as the consensus layer.

- `status_command`: This can be used instead of the information in DCS for the Harp Manager to determine whether the database is running. This is required if using bdr as the consensus layer.

- `stop_command`: This can be used instead of the information in DCS for stopping the database.

- `db_retry_wait_min`: The initial time in seconds to wait if Harp Manager cannot connect to the database before trying again. Harp Manager will increase the wait time with each attempt, up to the `db_retry_wait_max` value.

- `db_retry_wait_max`: The maximum time in seconds to wait if Harp Manager cannot connect to the database before trying again.

Thus a complete configuration example for HARP Manager might look like this:

```
cluster:
  name: mycluster

dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379

manager:
  name: node1
  log_level: INFO
```

This configuration is essentially the DCS contact information, any associated service customizations, the name of the cluster, and the name of the node. All other settings are associated with the node and is stored in the DCS.

Read the Node bootstrapping for more about specific node settings and initializing nodes to be managed by HARP Manager.

HARP Proxy specific

Some configuration options are specific to HARP Proxy. These affect how the daemon operates and thus are currently located in `config.yml`.

Specify Proxy-based settings under a `proxy` heading, and include:

- `location`: Required name of location for HARP Proxy to represent. HARP Proxy nodes are directly tied to the location where they are running, as they always direct traffic to the current lead master node. Specify location for any defined proxy.

- `log_level`: One of `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `CRITICAL`, which might alter the amount of log output from HARP services.

  - Default: `INFO`

- `name`: Name of this specific proxy. Each proxy node is named to ensure any associated statistics or operating state are available in status checks and other interactive events.

- `type`: Specifies whether to use pgbouncer or the experimental built-in passthrough proxy. All proxies must use the same proxy type. We recommend to experimenting with only the simple proxy in combination with the experimental BDR DCS. Can be `pgbouncer` or `builtin`.

  - Default: `pgbouncer`

- `pgbouncer_bin_dir`: Directory where PgBouncer binaries are located. As HARP uses PgBouncer binaries, it needs to know where they are located. This can be depend on the platform or distribution, so it has no default. Otherwise, the assumption is that the appropriate binaries are in the environment's `PATH` variable.

Example

HARP Proxy requires the cluster name, DCS connection settings, location, and name of the proxy in operation. For example:

```
cluster:
  name: mycluster

dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379

proxy:
  name: proxy1
  location: dc1
  pgbouncer_bin_dir: /usr/sbin
```

All other attributes are obtained from the DCS on proxy startup.

Runtime directives

While it is possible to configure HARP Manager, HARP Proxy, or harpctl with a minimum of YAML in the `config.yml` file, some customizations are held in the DCS. These values must either be initialized via bootstrap or set specifically with `harpctl set` directives.

Cluster-wide

Set these settings under a `cluster` YAML heading during bootstrap, or modify them with a `harpctl set cluster` command.

- `event_sync_interval`: Time in milliseconds to wait for synchronization. When events occur in HARP, they do so asynchronously across the cluster. HARP managers start operating immediately when they detect metadata changes, and HARP proxies might pause traffic and start reconfiguring endpoints. This is a safety interval that roughly approximates the maximum amount of event time skew that exists between all HARP components.

For example, suppose Node A goes offline and HARP Manager on Node B commonly receives this event 5 milliseconds before Node C. A setting of at least 5 ms is then needed to ensure all HARP Manager services receive the event before they begin to process it.

This also applies to HARP Proxy.

Node directives

You can change most node-oriented settings and then apply them while HARP Manager is active. These items are retained in the DCS after initial bootstrap, and thus you can modify them without altering a configuration file.

Set these settings under a `node` YAML heading during bootstrap, or modify them with a `harpctl set node` command.

- `node_type`: The type of this database node, either `bdr` or `witness`. You can't promote a witness node to leader.
- `camo_enforcement`: Whether to strictly enforce CAMO queue state. When set to `strict`, HARP never allows switchover or failover to a BDR CAMO partner node unless it's fully caught up with the entire CAMO queue at the time of the migration. When set to `lag_only`, only standard lag thresholds such as `maximum_camo_lag` are applied.
- `dcs_reconnect_interval`: The interval, measured in milliseconds, between attempts that a disconnected node tries to reconnect to the DCS.
  - Default: 1000.
- `dsn`: Required full connection string to the managed Postgres node. This parameter applies equally to all HARP services and enables micro-architectures that run only one service per container.

  !!! Note HARP sets the `sslmode` argument to `require` by default and prevents connections to servers that don't require SSL. To disable this behavior, explicitly set this parameter to a more permissive value such as `disable`, `allow`, or `prefer`.
- `db_data_dir`: Required Postgres data directory. This is required by HARP Manager to start, stop, or reload the Postgres service. It's also the default location for configuration files, which you can use later for controlling promotion of streaming replicas.
- `db_conf_dir`: Location of Postgres configuration files. Some platforms prefer storing Postgres configuration files away from the Postgres data directory. In these cases, set this option to that expected location.
- `db_log_file`: Location of Postgres log file.
  - Default: `/tmp/pg_ctl.out`
- `fence_node_on_dcs_failure`: If HARP can't reach the DCS, several readiness keys and the leadership lease expire. This implicitly prevents a node from routing consideration. However, such a node isn't officially fenced, and the Manager doesn't stop monitoring the database if `stop_database_when_fenced` is set to `false`.
  - Default: False
- `leader_lease_duration`: Amount of time in seconds the lead master lease persists if not refreshed. This allows any HARP Manager a certain grace period to refresh the lock, before expiration allows another node to obtain the lead master lock instead.
  - Default: 6
- `lease_refresh_interval`: Amount of time in milliseconds between refreshes of the lead master lease. This

essentially controls the time between each series of checks HARP Manager performs against its assigned Postgres node and when the status of the node is updated in the consensus layer.

  - Default: 2000

- `max_dcs_failures`: The amount of DCS request failures before marking a node as fenced according to `fence_node_on_dcs_failure`. This setting prevents transient communication disruptions from shutting down database nodes.

  - Default: 10

- `maximum_lag`: Highest allowable variance (in bytes) between last recorded LSN of previous lead master and this node before being allowed to take the lead master lock. This setting prevents nodes experiencing terminal amounts of lag from taking the lead master lock. Set to `-1` to disable this check.

  - Default: -1

- `maximum_camo_lag`: Highest allowable variance (in bytes) between last received LSN and applied LSN between this node and its CAMO partners. This applies only to clusters where CAMO is both available and enabled. Thus this applies only to BDR EE clusters where `pg2q.enable_camo` is set. For clusters with particularly stringent CAMO apply queue restrictions, set this very low or even to `0` to avoid any unapplied CAMO transactions. Set to `-1` to disable this check.

  - Default: -1

- `ready_status_duration`: Amount of time in seconds the node's readiness status persists if not refreshed. This is a failsafe that removes a node from being contacted by HARP Proxy if the HARP Manager in charge of it stops operating.

  - Default: 30

- `db_bin_dir`: Directory where Postgres binaries are located. As HARP uses Postgres binaries, such as `pg_ctl`, it needs to know where they're located. This can depend on the platform or distribution, so it has no default. Otherwise, the assumption is that the appropriate binaries are in the environment's `PATH` variable.

- `priority`: Any numeric value. Any node where this option is set to `-1` can't take the lead master role, even when attempting to explicitly set the lead master using `harpctl`.

  - Default: 100

- `stop_database_when_fenced`: Rather than removing a node from all possible routing, stop the database on a node when it is fenced. This is an extra safeguard to prevent data from other sources than HARP Proxy from reaching the database or in case proxies can't disconnect clients for some other reason.

  - Default: False

- `consensus_timeout`: Amount of milliseconds before aborting a read or write to the consensus layer. If the consensus layer loses quorum or becomes unreachable, you want near-instant errors rather than infinite timeouts. This prevents blocking behavior in such cases. When using `bdr` as the consensus layer, the highest recognized timeout is 1000 ms.

  - Default: 250

- `use_unix_socket`: Specifies for HARP Manager to prefer to use Unix sockets to connect to the database.

  - Default: False

All of these runtime directives can be modified via `harpctl`. Consider if you want to decrease the `lease_refresh_interval` to 100ms on `node1`:

```
harpctl set node node1 lease_refresh_interval=100
```

Proxy directives

You can change certain settings to the proxy while the service is active. These items are retained in the DCS after initial bootstrap, and thus you can modify them without altering a configuration file. Many of these settings are direct mappings to their PgBouncer equivalent, and we will note these where relevant.

Set these settings under a `proxies` YAML heading during bootstrap, or modify them with a `harpctl set proxy` command. Properties set by `harpctl set proxy` require a restart of the proxy.

- `auth_file`: The full path to a PgBouncer-style `userlist.txt` file. HARP Proxy uses this file to store a `pgbouncer` user that has access to PgBouncer's Admin database. You can use this for other users as well. Proxy modifies this file to add and modify the password for the `pgbouncer` user.
    - Default: `/etc/harp/userlist.txt`
- `auth_type`: The type of Postgres authentication to use for password matching. This is actually a PgBouncer setting and isn't fully compatible with the Postgres `pg_hba.conf` capabilities. We recommend using `md5`, `pam` `cert`, or `scram-sha-256`.
    - Default: `md5`
- `auth_query`: Query to verify a user's password with Postgres. Direct access to `pg_shadow` requires admin rights. It's better to use a non-superuser that calls a `SECURITY DEFINER` function instead. If using TPAexec to create a cluster, a function named `pgbouncer_get_auth` is installed on all databases in the `pg_catalog` namespace to fulfill this purpose.
- `auth_user`: If `auth_user` is set, then any user not specified in `auth_file` is queried through the `auth_query` query from `pg_shadow` in the database, using `auth_user`. The password of `auth_user` is taken from `auth_file`.
- `client_tls_ca_file`: Root certificate file to validate client certificates. Requires `client_tls_sslmode` to be set.
- `client_tls_cert_file`: Certificate for private key. Clients can validate it. Requires `client_tls_sslmode` to be set.
- `client_tls_key_file`: Private key for PgBouncer to accept client connections. Requires `client_tls_sslmode` to be set.
- `client_tls_protocols`: TLS protocol versions allowed for client connections. Allowed values: `tlsv1.0`, `tlsv1.1`, `tlsv1.2`, `tlsv1.3`. Shortcuts: `all` (tlsv1.0,tlsv1.1,tlsv1.2,tlsv1.3), `secure` (tlsv1.2,tlsv1.3), `legacy` (all).
    - Default: `secure`
- `client_tls_sslmode`: Whether to enable client SSL functionality. Possible values are `disable`, `allow`, `prefer`, `require`, `verify-ca`, and `verify-full`.
    - Default: `disable`

- `database_name`: Required name that represents the database clients use when connecting to HARP Proxy. This is a stable endpoint that doesn't change and points to the current node, database name, port, etc., necessary to connect to the lead master. You can use the global value `*` here so all connections get directed to this target regardless of database name.
- `default_pool_size`: The maximum number of active connections to allow per database/user combination. This is for connection pooling purposes but does nothing in session pooling mode. This is a PgBouncer setting.
  - Default: 25
- `ignore_startup_parameters`: By default, PgBouncer allows only parameters it can keep track of in startup packets: `client_encoding`, `datestyle`, `timezone`, and `standard_conforming_strings`. All other parameters raise an error. To allow other parameters, you can specify them here so that PgBouncer knows that they are handled by the admin and it can ignore them. Often, you need to set this to `extra_float_digits` for Java applications to function properly.
  - Default: `extra_float_digits`
- `listen_address`: IP addresses where Proxy should listen for connections. Used by pgbouncer and builtin proxy.
  - Default: 0.0.0.0
- `listen_port`: System port where Proxy listens for connections. Used by pgbouncer and builtin proxy.
  - Default: 6432
- `max_client_conn`: The total maximum number of active client connections that are allowed on the proxy. This can be many orders of magnitude greater than `default_pool_size`, as these are all connections that have yet to be assigned a session or have released a session for use by another client connection. This is a PgBouncer setting.
  - Default: 100
- `monitor_interval`: Time in seconds between Proxy checks of PgBouncer. Since HARP Proxy manages PgBouncer as the actual connection management layer, it needs to periodically check various status and stats to verify it's still operational. You can also log or register some of this information to the DCS.
  - Default: 5
- `server_tls_protocols`: TLS protocol versions are allowed for server connections. Allowed values: `tlsv1.0`, `tlsv1.1`, `tlsv1.2`, `tlsv1.3`. Shortcuts: `all` (tlsv1.0,tlsv1.1,tlsv1.2,tlsv1.3), `secure` (tlsv1.2,tlsv1.3), `legacy` (all).
  - Default: `secure`
- `server_tls_sslmode`: Whether to enable server SSL functionality. Possible values are `disable`, `allow`, `prefer`, `require`, `verify-ca`, and `verify-full`.
  - Default: `disable`
- `session_transfer_mode`: Method by which to transfer sessions. Possible values are `fast`, `wait`, and `reconnect`.
  - Default: `wait`

  This property isn't used by the builtin proxy.
- `server_transfer_timeout`: The number of seconds Harp Proxy waits before giving up on a PAUSE and issuing

a KILL command.

  - Default: 30

The following two options apply only when using the built-in proxy.

- `keepalive`: The number of seconds the built-in proxy waits before sending a keepalive message to an idle leader connection.
  - Default: 5
- `timeout`: The number of seconds the built-in proxy waits before giving up on connecting to the leader.
  - Default: 1

When using `harpctl` to change any of these settings for all proxies, use the `global` keyword in place of the proxy name. For example:

```
harpctl set proxy global max_client_conn=1000
```

7.3 Cluster bootstrapping

To use HARP, a minimum amount of metadata must exist in the DCS. The process of "bootstrapping" a cluster essentially means initializing node, location, and other runtime configuration either all at once or on a per-resource basis.

This process is governed through the `harpctl apply` command. For more information, see harpctl command-line tool.

Set up the DCS and make sure it is functional before bootstrapping.

!!! Important You can combine any or all of these example into a single YAML document and apply it all at once.

Cluster-wide bootstrapping

Some settings are applied cluster-wide and you can specify them during bootstrapping. Currently this applies only to the `event_sync_interval` runtime directive, but others might be added later.

The format is as follows:

```
cluster:
  name: mycluster
  event_sync_interval: 100
```

Assuming that file was named `cluster.yml`, you then apply it with the following:

```
harpctl apply cluster.yml
```

If the cluster name isn't already defined in the DCS, this also initializes that value.

!!! Important The cluster name parameter specified here always overrides the cluster name supplied in `config.yml`.

The assumption is that the bootstrap file supplies all necessary elements to bootstrap a cluster or some portion of its larger configuration. A `config.yml` file is primarily meant to control the execution of HARP Manager, HARP Proxy, or `harpctl` specifically.

Location bootstrapping

Every HARP node is associated with at most one location. This location can be a single data center, a grouped region consisting of multiple underlying servers, an Amazon availability zone, and so on. This is a logical structure that allows HARP to group nodes together such that only one represents the nodes in that location as the lead master.

Thus it is necessary to initialize one or more locations. The format for this is as follows:

```
cluster:
  name: mycluster

locations:
  - location: dc1
  - location: dc2
```

Assuming that file was named `locations.yml`, you then apply it with the following:

```
harpctl apply locations.yml
```

When performing any manipulation of the cluster, include the name as a preamble so the changes are directed to the right place.

Once locations are bootstrapped, they show up with a quick examination:

```
> harpctl get locations

Cluster   Location Leader Previous Leader Target Leader Lease Renewals
-------   -------- ------ --------------- ------------- --------------
mycluster dc1                                          <nil>
mycluster dc2                                          <nil>
```

Both locations are recognized by HARP and available for node and proxy assignment.

Node bootstrapping

HARP nodes exist in a named cluster and must have a designated name. Beyond this, all other settings are retained in the DCS, as they are dynamic and can affect how HARP interacts with them. To this end, bootstrap each node using one or more of the runtime directives discussed in Configuration.

While bootstrapping a node, there are a few required fields:

- `name`
- `location`
- `dsn`
- `pg_data_dir`

Everything else is optional and can depend on the cluster. Because you can bootstrap multiple nodes at once, the format generally fits this structure:

```
cluster:
  name: mycluster

nodes:
  - name: node1
    location: dc1
    dsn: host=node1 dbname=bdrdb user=postgres
    pg_data_dir: /db/pgdata
    leader_lease_duration: 10
    priority: 500
```

Assuming that file was named `node1.yml`, you then apply it with the following:

```
harpctl apply node1.yml
```

Once nodes are bootstrapped, they show up with a quick examination:

```
> harpctl get nodes

Cluster     Name  Location Ready Fenced Allow Routing Routing Status Role    Type
Lock Duration
-------     ----  -------- ----- ------ ------------- -------------- ----    ----
-------------
mycluster   bdra1 dc1      true  false  true          ok             primary bdr
30
```

Proxy bootstrapping

Unlike locations or nodes, proxies can also supply a configuration template that is applied to all proxies in a location. These are stored in the DCS under the `global` designation. Each proxy also requires a name to exist as an instance, but no further customization is needed unless some setting needs a specific override.

This is because there are likely to be multiple proxies that have the same default configuration settings for the cluster, and repeating these values for every single proxy isn't necessary.

Additionally, when bootstrapping the proxy template, define at least one database for connection assignments. With these notes in mind, the format for this is as follows:

```
cluster:
  name: mycluster

proxies:
  monitor_interval: 5
  default_pool_size: 20
  max_client_conn: 1000
  database_name: bdrdb
  instances:
    - name: proxy1
    - name: proxy2
      default_pool_size: 50
```

This configures HARP for two proxies: `proxy1` and `proxy2`. Only `proxy2` has a custom `default_pool_size`, while using the global settings otherwise.

Assuming that file was named `proxy.yml`, you then apply it with the following:

```
harpctl apply proxy.yml
```

Once the proxy template is bootstrapped, it shows up with a quick examination:

```
> harpctl get proxies

Cluster   Name   Pool Mode Auth Type Max Client Conn Default Pool Size
-------   ----   --------- --------- ---------------- ------------------
mycluster global session   md5       1000             20
mycluster proxy1 session   md5       1000             20
mycluster proxy2 session   md5       1000             50
```

7.4 HARP Manager

HARP Manager is a daemon that interacts with the local PostgreSQL/BDR node and stores information about its state in the consensus layer. Manager determines the node that currently holds leader status for a respective location and enforces configuration (lag, CAMO lag, etc.) constraints to prevent ineligible nodes from leader consideration.

Every Postgres node in the cluster must have an associated HARP Manager. Other nodes can exist, but they can't to participate as lead or shadow master roles or any other functionality that requires a HARP Manager.

!!! Important HARP Manager expects the be used to start and stop the database. Stopping HARP Manager will stop the database. Starting HARP Manager will start the database if it isn't already started. If another method is used to stop the database then HARP Manager will try and restart it.

How it works

Upon starting, HARP Manager uses `pg_ctl` to start Postgres if it isn't already running. After this, it periodically checks the server as defined by the `node.lease_refresh_interval` setting. HARP Manager collects various bits of data about Postgres including:

- The node's current LSN.
- If Postgres is running and accepting connections. This particular data point is considered a lease that must be periodically renewed. If it expires, HARP Proxy removes the node from any existing routing.
- The current apply LSN position for all upstream BDR peer nodes.
- If CAMO is enabled:
  - Name of the CAMO partner
  - Peer CAMO state (`is_ready`)
  - CAMO queue received and applied LSN positions

- Node type, such as whether the node is BDR or regular Postgres.
- The node's current role, such as a read/write, physical streaming replica, logical standby, and so on.
- BDR node state, which is `ACTIVE` except in limited cases.
- BDR Node ID for other metadata gathering.
- Other tracking values.

!!! Important When naming BDR nodes in HARP, the BDR node name must match the node name represented in the

`node.name` configuration attribute. This occurs in the bootstrap process.

The data collected here is fully available to other HARP Manager processes and is used to evaluate lag, partner readiness, and other criteria that direct switchover and failover behavior.

After updating the node metadata, HARP Manager either refreshes the lead master lease if it's already held by the local node or seeks to obtain the lease if it's expired. Since the current state of all nodes is known to all other nodes, the node that was the previous lead master is given automatic priority ranking if present. If not, all other nodes list themselves by LSN lag, node priority, and other criteria, and the most qualified node seizes the lead master lease.

This procedure happens for every defined location where nodes are present. Thus for locations DC1 and DC2, there is a lead master node in each, with a separate lease and election process for both.

HARP Manager repeats these Postgres status checks, lease renewals, and elections repeatedly to ensure the cluster always has a lead master target for connections from HARP Proxy.

Configuration

HARP Manager expects the `dcs`, `cluster`, and `manager` configuration stanzas. The following is a functional example:

```
cluster:
  name: mycluster

dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379

manager:
  name: node1
  postgres_bin_dir: /usr/lib/postgresql/13/bin
```

You can apply changes to the configuration file (default: `/etc/harp/config.yml`) by issuing `SIGHUP` to the running instance or by calling a service-level reload.

See Configuration for further details.

Usage

This is the basic usage for HARP Manager:

```
Usage of ./harp-manager:
  -f string
        Optional path to config file (shorthand)
  --config string
        Optional path to config file
```

There are no arguments to launch `harp-manager` as a forked daemon. This software is designed to be launched through systemd or in a container as a top-level process. This also means output is directed to STDOUT and STDERR for capture and access through journald or an attached container terminal.

Disabling and reenabling HARP Manager control of Postgres

You can temporarily pause HARP Manager control of Postgres. This results in a state where the daemon continues running but doesn't perform any operations that can affect existing behavior of the cluster. Reenabling management causes it to resume operation.

An example of temporarily disabling node management is:

```
harpctl unmanage node node1
```

See harpctl command-line tool for more details.

Node management by HARP Manager is enabled by default.

7.5 HARP Proxy

HARP Proxy is a daemon that acts as an abstraction layer between the client application and Postgres. It interfaces with the consensus layer to obtain the identity of the current lead master node and directs traffic to that location. During a planned switchover or unplanned failover, it redirects to the new lead master node as dictated by the DCS.

You can select between `pgbouncer` or `builtin` for HARP Proxy. If you don't specify a proxy type, the default is `builtin`. When using `pgbouncer`, HARP Proxy is an interface layer between the DCS and PgBouncer. As such, PgBouncer is a prerequisite and must also be installed for HARP Proxy to fully manage its activity.

The builtin proxy doesn't require any additional software. When using builtin, HARP Proxy functions as a level 4 pass-through proxy.

Builtin proxy: how it works

Upon starting, HARP Proxy listens for incoming connections on the listening address and listening port specified in the bootstrap file per proxy instance.
All application client traffic then passes through builtin proxy into the current lead master node for the location where this proxy is operating.

If the lead master lease isn't set, HARP Proxy disconnects all connection traffic until a new lead master is established. This also applies to circumstances when `harpctl promote` is used to invoke a planned transition to a new lead master. The disconnect is immediate.

Configuration

Choose the built-in proxy by setting the proxy type to `builtin`. The only other option that applies to the built-in proxy is `max_client_conn`, which specifies the maximum allowed client connections. If `max_client_conn` is higher than what the system can handle, it is lowered to a setting that's within the capability of the system that the proxy is on.

PgBouncer: how it works

!!! Note If you need more configurability of pgbouncer than what Harp Proxy provides, the recommended setup is to use builtin proxy and have pgbouncer point to it.

Upon starting, HARP Proxy launches PgBouncer if it's not already running and leaves client connections paused. After, it contacts the DCS to determine the identity of the lead master, configure PgBouncer to use this as the target for database connections, and resume connection activity. All application client traffic then passes through PgBouncer into the current lead master node for the location where this proxy is operating.

While PgBouncer is running, HARP Proxy checks its status based on the `monitor_interval` configuration setting in the DCS and stores it in the DCS for monitoring purposes. This configuration allows interrogation with `harpctl` to retrieve status of all configured proxies or any one proxy.

If the lead master lease isn't set, HARP Proxy pauses all connection traffic until a new lead master is established. This also applies to circumstances when `harpctl promote` is used to invoke a planned transition to a new lead master. It uses a PgBouncer `PAUSE` command for this, so existing sessions are allowed to complete any pending transactions before they're held in stasis.

PgBouncer configuration file

When HARP Proxy uses PgBouncer for connection management and redirection, a `pgbouncer.ini` file must exist. HARP Manager builds this file based on various runtime directives as defined in Proxy directives.

This file is located in the same folder as the `config.yml` used by HARP Proxy. Any PgBouncer process launched by HARP Proxy uses this configuration file, and you can use it for debugging or information purposes. Modifications to this automatically generated `pgbouncer.ini` file are lost any time HARP Proxy restarts, so use `harpctl set proxy` to alter these settings instead. Calling `harpctl set proxy` doesn't update the `pgbouncer.ini` file until the proxy restarts.

Disabling and reenabling HARP Proxy node management

You can temporarily pause HARP Proxy control of PgBouncer. This results in a state where the daemon continues running but doesn't perform any operations that can affect existing behavior of the cluster. Reenabling management causes it to resume operation.

An example of temporarily disabling management of a specific proxy is:

```
harpctl unmanage proxy proxy1
```

See harpctl command-line tool for more details.

Proxy node management is enabled by default.

Passthrough user authentication

With pgbouncer, we strongly recommend configuring HARP Proxy to use the `auth_user` and `auth_query` runtime directives. If these aren't set, the PgBouncer `userlist.txt` file must include username and password hash combinations for every user PgBouncer needs to authenticate on behalf of Postgres.

Do *not* use the `pgbouncer` user, as this this is used by HARP Proxy as an admin-level user to operate the underlying PgBouncer service.

Configuration

HARP Proxy expects the `dcs`, `cluster`, and `proxy` configuration stanzas. The following is a functional example:

```
cluster:
  name: mycluster

dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379

proxy:
  name: proxy1
```

Each proxy connects to the DCS to retrieve the hosts and ports to listen on for connections.

Usage

This is the basic usage for HARP Proxy:

```
Usage of ./harp-proxy:
  -f string
        Optional path to config file (shorthand)
  --config string
        Optional path to config file
```

There are no arguments to launch `harp-proxy` as a forked daemon. This software is designed to be launched through systemd or in a container as a top-level process. This also means output is directed to STDOUT and STDERR for capture and access through journald or an attached container terminal.

7.6 harpctl command-line tool

`harpctl` is a command-line tool for directly manipulating the consensus layer contents to fit desired cluster geometry. You can use it to, for example, examine node status, "promote" a node to lead master, disable/enable cluster management, bootstrap cluster settings, and so on.

Synopsis

```
$ harpctl --help

Usage:
  harpctl [command]

Available Commands:
  apply       Command to set up a cluster
  completion  generate the autocompletion script for the specified shell
  fence       Fence specified node
  get         Command to get resources
  help        Help about any command
  manage      Manage Cluster
  promote     Promotes the sepcified node to be primary,
  set         Command to set resources.
  unfence     unfence specified node
  unmanage    Unmanage Cluster
  version     Command to get version information

Flags:
  -c, --cluster string       name of cluster.
  -f, --config-file string   config file (default is /etc/harp/config.yml)
      --dcs stringArray      Address of dcs endpoints. ie: 127.0.0.1:2379
  -h, --help                 help for harpctl
  -o, --output string         'json, yaml'.
  -t, --toggle               Help message for toggle

Use "harpctl [command] --help" for more information about a command.
```

In addition to this basic synopsis, each of the available commands has its own series of allowed subcommands and flags.

Configuration

`harpctl` must interact with the consensus layer to operate. This means a certain minimum amount of settings must be defined in `config.yml` for successful execution. This includes:

- `dcs.driver`
- `dcs.endpoints`
- `cluster.name`

As an example using etcd:

```
cluster:
  name: mycluster
dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379
```

See Configuration for details.

Execution

Execute `harpctl` like this:

```
harpctl command [flags]
```

Each command has its own series of subcommands and flags. Further help for these are available by executing this command:

```
harpctl <command> --help
```

`harpctl apply`

You must use an `apply` command to "bootstrap" a HARP cluster using a file that defines various attributes of the intended cluster.

Execute an `apply` command like this:

```
harpctl apply <filename>
```

This essentially creates all of the initial cluster metadata, default or custom management settings, and so on. This is done here because the DCS is used as the ultimate source of truth for managing the cluster, and this makes it possible to change these settings dynamically.

This can either be done once to bootstrap the entire cluster, once per type of object, or even on a per-node basis for the sake of simplicity.

This is an example of a bootstrap file for a single node:

```
cluster:
  name: mycluster
nodes:
  - name: firstnode
    dsn: host=host1 dbname=bdrdb user=harp_user
    location: dc1
    priority: 100
    db_data_dir: /db/pgdata
```

As seen here, it is good practice to always include a cluster name preamble to ensure all changes target the correct HARP cluster, in case several are operating in the same environment.

Once `apply` completes without error, the node is integrated with the rest of the cluster.

!!! Note You can also use this command to bootstrap the entire cluster at once since all defined sections are applied at the same time. However, we don't encourage this use for anything but testing as it increases the difficulty of validating each portion of the cluster during initial definition.

`harpctl fence`

Marks the local or specified node as fenced. A node with this status is essentially completely excluded from the cluster. HARP Proxy doesn't send it traffic, its representative HARP Manager doesn't claim the lead master lease, and further steps are also taken. If running, HARP Manager stops Postgres on the node as well.

Execute a `fence` command like this:

```
harpctl fence (<node-name>)
```

The node-name is optional; if omitted, `harpctl` uses the name of the locally configured node.

harpctl get

Fetches information stored in the consensus layer for various elements of the cluster. This includes nodes, locations, the cluster, and so on. The full list includes:

- `cluster` — Returns the cluster state.
- `leader` — Returns the current or specified location leader.
- `location` — Returns current or specified location information.
- `locations` — Returns list of all locations.
- `node` — Returns the specified Postgres node.
- `nodes` — Returns list of all Postgres nodes.
- `proxy` — Returns current or specified proxy information.
- `proxies` — Returns list of all proxy nodes.

harpctl get cluster

Fetches information stored in the consensus layer for the current cluster:

```
> harpctl get cluster

Name      Enabled
----      -------
mycluster true
```

harpctl get leader

Fetches node information for the current lead master stored in the DCS for the specified location. If no location is passed, `harpctl` attempts to derive it based on the location of the current node where it was executed.

Example:

```
> harpctl get leader dc1

Cluster   Name   Ready Role    Type Location Fenced Lock Duration
-------   ----   ----- ----    ---- -------- ------ -------------
mycluster mynode true  primary bdr  dc1      false  30
```

harpctl get location

Fetches location information for the specified location. If no location is passed, `harpctl` attempts to derive it based on the location of the current node where it was executed.

Example:

```
> harpctl get location dc1

Cluster   Location Leader Previous Leader Target Leader Lease Renewals
-------   -------- ------ --------------- ------------- --------------
mycluster dc1      mynode mynode                        <nil>
```

harpctl get locations

Fetches information for all locations currently present in the DCS.

Example:

```
> harpctl get locations

Cluster   Location Leader   Previous Leader Target Leader Lease Renewals
-------   -------- ------   --------------- ------------- --------------
mycluster dc1      mynode   mynode                        <nil>
mycluster dc2      thatnode thatnode                      <nil>
```

harpctl get node

Fetches node information stored in the DCS for the specified node.

Example:

```
> harpctl get node mynode
Cluster    Name   Location Ready Fenced Allow Routing Routing Status Role    Type
Lock Duration
-------    ----   -------- ----- ------ ------------- -------------- ----    ----
-------------
mycluster  mynode dc1      true  false  true          ok               primary bdr
30
```

harpctl get nodes

Fetches node information stored in the DCS for the all nodes in the cluster.

Example:

```
> harpctl get nodes

Cluster    Name  Location Ready Fenced Allow Routing Routing Status Role    Type
Lock Duration
-------    ----  -------- ----- ------ ------------- -------------- ----    ---- -
------------
myclusters bdra1 dc1      true  false  true          ok             primary bdr  30
myclusters bdra2 dc1      true  false  false         N/A            primary bdr  30
myclusters bdra3 dc1      true  false  false         N/A            primary bdr  30
```

harpctl get proxy

Fetches proxy information stored in the DCS for specified proxy. Specify `global` to see proxy defaults for this cluster.

Example:

```
> harpctl get proxy proxy1

Cluster   Name   Pool Mode Auth Type Max Client Conn Default Pool Size
-------   ----   --------- --------- --------------- -----------------
mycluster proxy1 session   md5       1000            20
```

`harpctl get proxies`

Fetches proxy information stored in the DCS for all proxies in the cluster. Additionally, lists the `global` pseudo-proxy for default proxy settings.

Example:

```
> harpctl get proxies

Cluster   Name   Pool Mode Auth Type Max Client Conn Default Pool Size
-------   ----   --------- --------- --------------- -----------------
mycluster global session   md5       500             25
mycluster proxy1 session   md5       1000            20
mycluster proxy2 session   md5       1500            30
```

`harpctl manage`

If a cluster isn't in a managed state, instructs all HARP Manager services to resume monitoring Postgres and updating the consensus layer. Do this after maintenance is complete following HARP software updates or other significant changes that might affect the whole cluster.

Execute a `manage` command like this:

```
harpctl manage cluster
```

!!! Note Currently you can enable or disable cluster management only at the `cluster` level. Later versions will also make it possible to do this for individual nodes or proxies.

`harpctl promote`

Promotes the next available node that meets leadership requirements to lead master in the current Location. Since this is a requested event, it invokes a smooth handover where:

1. The existing lead master releases the lead master lease, provided:
   - If CAMO is enabled, the promoted node must be up to date and CAMO ready, and the CAMO queue must have less than `node.maximum_camo_lag` bytes remaining to be applied.
   - Replication lag between the old lead master and the promoted node is less than `node.maximum_lag`.

2. The promoted node is the only valid candidate to take the lead master lease and does so as soon as it is released by the current holder. All other nodes ignore the unset lead master lease.
   - If CAMO is enabled, the promoted node temporarily disables client traffic until the CAMO queue is fully applied,

even though it holds the lead master lease.

3. HARP Proxy, if using pgbouncer, will `PAUSE` connections to allow ongoing transactions to complete. Once the lead master lease is claimed by the promoted node, it reconfigures PgBouncer for the new connection target and resumes database traffic. If HARP Proxy is using the builtin proxy, it terminates existing connections and creates new connections to the lead master as new connections are requested from the client.

Execute a `promote` command like this:

```
harpctl promote (<node-name>)
```

Provide the `--force` option to forcibly set a node to lead master, even if it doesn't meet the criteria for becoming lead master. This circumvents any verification of CAMO status or replication lag and causes an immediate transition to the promoted node. This is the only way to specify an exact node for promotion.

The node must be online and operational for this to succeed. Use this option with care.

`harpctl set`

Sets a specific attribute in the cluster to the supplied value. This is used to tweak configuration settings for a specific node, proxy, location, or the cluster rather than using `apply`. You can use this for the following object types:

- `cluster` — Sets cluster-related attributes.
- `location` — Sets specific location attributes.
- `node` — Sets specific node attributes.
- `proxy` — Sets specific proxy attributes.

`harpctl set cluster`

Sets cluster-related attributes only.

Example:

```
harpctl set cluster event_sync_interval=200
```

`harpctl set node`

Sets node-related attributes for the named node. Any options mentioned in Node directives are valid here.

Example:

```
harpctl set node mynode priority=500
```

`harpctl set proxy`

Sets proxy-related attributes for the named proxy. Any options mentioned in the Proxy directives are valid here. Properties set this way require a restart of the proxy before the new value takes effect.

Example:

```
harpctl set proxy proxy1 max_client_conn=750
```

Use `global` for cluster-wide proxy defaults:

```
harpctl set proxy global default_pool_size=10
```

`harpctl unfence`

Removes the `fenced` attribute from the local or specified node. This removes all previously applied cluster exclusions from the node so that it can again receive traffic or hold the lead master lease. Postgres is also started if it isn't running.

Execute an `unfence` command like this:

```
harpctl unfence (<node-name>)
```

The node-name is optional. If you omit it, `harpctl` uses the name of the locally configured node.

`harpctl unmanage`

Instructs all HARP Manager services in the cluster to remain running but no longer actively monitoring Postgres, or modify the contents of the consensus layer. This means that any ordinary failover event such as a node outage doesn't result in a leadership migration. This is intended for system or HARP maintenance prior to making changes to HARP software or other significant changes to the cluster.

Execute an `unmanage` command like this:

```
harpctl unmanage cluster
```

!!! Note Currently you can enable or disable cluster management at only the `cluster` level. Later versions will also make it possible to do this for individual nodes or proxies.

7.7 Consensus layer considerations

HARP is designed so that it can work with different implementations of consensus layer, also known as Distributed Control Systems (DCS).

Currently the following DCS implementations are supported:

- etcd
- BDR

This information is specific to HARP's interaction with the supported DCS implementations.

BDR driver compatibility

The `bdr` native consensus layer is available from BDR versions 3.6.21 and 3.7.3.

For the purpose of maintaining a voting quorum, BDR Logical Standby nodes don't participate in consensus

communications in a EDB Postgres Distributed cluster. Don't count these in the total node list to fulfill DCS quorum requirements.

Maintaining quorum

Clusters of any architecture require at least n/2 + 1 nodes to maintain consensus via a voting quorum. Thus a three-node cluster can tolerate the outage of a single node, a five-node cluster can tolerate a two-node outage, and so on. If consensus is ever lost, HARP becomes inoperable because the DCS prevents it from deterministically identifying the node that is the lead master in a particular location.

As a result, whichever DCS is chosen, more than half of the nodes must always be available *cluster-wide*. This can become a non-trivial element when distributing DCS nodes among two or more data centers. A network partition prevents quorum in any location that can't maintain a voting majority, and thus HARP stops working.

Thus an odd-number of nodes (with a minimum of three) is crucial when building the consensus layer. An ideal case distributes nodes across a minimum of three independent locations to prevent a single network partition from disrupting consensus.

One example configuration is to designate two DCS nodes in two data centers coinciding with the primary BDR nodes, and a fifth DCS node (such as a BDR witness) elsewhere. Using such a design, a network partition between the two BDR data centers doesn't disrupt consensus thanks to the independently located node.

Multi-consensus variant

HARP assumes one lead master per configured location. Normally each location is specified in HARP using the `location` configuration setting. By creating a separate DCS cluster per location, you can emulate this behavior independently of HARP.

To accomplish this, configure HARP in `config.yml` to use a different DCS connection target per desired Location.

HARP nodes in DC-A use something like this:

```
location: dca
dcs:
  driver: etcd
  endpoints:
    - dcs-a1:2379
    - dcs-a2:2379
    - dcs-a3:2379
```

While DC-B uses different hostnames corresponding to nodes in its canonical location:

```
location: dcb
dcs:
  driver: etcd
  endpoints:
    - dcs-a1:2379
    - dcs-a2:2379
    - dcs-a3:2379
```

There's no DCS communication between different data centers in this design, and thus a network partition between them doesn't affect HARP operation. A consequence of this is that HARP is completely unaware of nodes in the other location, and each location operates essentially as a separate HARP cluster.

This isn't possible when using BDR as the DCS, as BDR maintains a consensus layer across all participant nodes.

A possible drawback to this approach is that `harpctl` can't interact with nodes outside of the current location. It's impossible to obtain node information, get or set the lead master, or perform any other operation that targets the other location. Essentially this organization renders the `--location` parameter to `harpctl` unusable.

TPAexec and consensus

These considerations are integrated into TPAexec as well. When deploying a cluster using etcd, it constructs a separate DCS cluster per location to facilitate high availability in favor of strict consistency.

Thus this configuration example groups any DCS nodes assigned to the `first` location together, and the `second` location is a separate cluster:

```
cluster_vars:
  failover_manager: harp
  harp_consensus_protocol: etcd

locations:
  - Name: first
  - Name: second
```

To override this behavior, configure the `harp_location` implicitly to force a particular grouping.

Thus this example returns all etcd nodes into a single cohesive DCS layer:

```
cluster_vars:
  failover_manager: harp
  harp_consensus_protocol: etcd

locations:
  - Name: first
  - Name: second
  - Name: all_dcs

instance_defaults:
  vars:
    harp_location: all_dcs
```

The `harp_location` override might also be necessary to favor specific node groupings when using cloud providers such as Amazon that favor availability zones in regions over traditional data centers.

7.8 Security and roles

Beyond basic package installation and configuration, HARP requires Postgres permissions to operate. These allow it to gather information about Postgres or BDR as needed to maintain node status in the consensus layer.

Postgres permissions

Create the role specified in the `node.dsn` parameter in one of the following ways:

- `CREATE USER ...`
- `CREATE ROLE ... WITH LOGIN`

This syntax ensures the role can log into the database to gather diagnostic information.

Similarly, an entry must exist in `pg_hba.conf` for this role. You can do this in many ways. As an example, consider a VPN subnet where all database hosts are located somewhere in `10.10.*`. In such a case, the easiest approach is to add a specific line:

```
## TYPE      DATABASE    USER          ADDRESS          METHOD
hostssl     all         harp_user     10.10.1.1/16     scram-sha-256
```

!!! Note In this case we've used the more modern `scram-sha-256` authentication rather than `md5`, which is now deprecated. We've also elected to require SSL authentication by specifying `hostssl`.

BDR permissions

BDR nodes have metadata and views that are visible only when certain roles are granted to the HARP-enabled user. In this case, the HARP user requires the following:

```
GRANT bdr_monitor TO harp_user;
```

The `bdr_monitor` BDR role is meant for status monitoring tools to maintain ongoing information on cluster operation, thus it is well-suited to HARP.

BDR consensus permissions

When the `dcs.driver` configuration parameter is set to `bdr`, HARP uses BDR as the consensus layer. As such, it requires access to API methods that are currently available only to the `bdr_superuser` role. This means the HARP-enabled user requires the following:

```
GRANT bdr_superuser TO foobar;
```

Currently access to the BDR consensus model requires superuser equivalent permission.

!!! Important BDR superusers *are not* Postgres superusers. The `bdr_superuser` role is merely granted elevated privileges in BDR, such as access to restricted functions, tables, views, and other objects.

---

8 Choosing your architecture

Always On architectures reflect EDB's Trusted Postgres architectures that encapsulate practices and help you to achieve the highest possible service availability in multiple configurations. These configurations range from single-location architectures to complex distributed systems that protect from hardware failures and data center failures. The architectures leverage EDB Postgres Distributed's multi-master capability and its ability to achieve 99.999% availability, even during maintenance operations.

You can use EDB Postgres Distributed for architectures beyond the examples described here. Use-case-specific variations have been successfully deployed in production. However, these variations must undergo rigorous architecture review first. Also, EDB's standard deployment tool for Always On architectures, TPAExec, must be enabled to support the variations before they can be supported in production environments.

Standard EDB Always On architectures

EDB has identified four standard Always On architectures:

- Bronze: Single location with two data nodes, a witness node, and a local backup
- Silver: Single location with three data nodes and second location with an offsite backup
- Gold: Two locations with two data nodes each and a third location with a witness
- Platinum: Two locations with three data nodes each (two masters plus additional redundant hardware in a hot standby mode)

!!! Note "Note: A location is either a data center or availability zone [AZ]." !!! All Always On architectures protect a progressively robust range of failure situations. For example, Always On Bronze protects against local hardware failure but doesn't provide protection from location (data center or AZ) failure. Always On Silver makes sure that a backup is kept at a different location, thus providing some protection in case of the catastrophic loss of a location. However, the database still must be restored from backup first, which might violate Recovery Time Objective (RTO) requirements. Always On Gold provides two locations connected in a multi-master mesh network, making sure that service remains available even in case a location goes offline. Finally, Always On Platinum adds redundant hot standby hardware in both locations to maintain local high availability in case of a hardware failure.

Each architecture can provide zero Recovery Point Objective (RPO), as data can be streamed synchronously to at least one local master, thus guaranteeing zero data loss in case of local hardware failure. However, synchronous replication is highly discouraged for Bronze architectures due to the extended RPO on failure of either data node.

Increasing the availability guarantee drives additional cost for hardware and licenses, networking requirements, and operational complexity. Carefully consider your availability and compliance requirements before choosing an architecture.

Architecture details

EDB Postgres Distributed uses a Raft-based consensus architecture. While regular database operations (insert, select, delete) don't require cluster-wide consensus, EDB Postgres Distributed benefits from an odd number of nodes to make decisions that require consensus, such as generating new global sequences, or distributed DDL operations. Even the simpler architectures always have three nodes within a location, even if not all of them are storing data. Always On Gold and Platinum, which use two locations, introduce a fifth node as a witness node to support the RAFT requirements.

Applications connect to the standard Always On architectures by way of multi-host connection strings, where each HARP-proxy server is a distinct entry in the multi-host connection string. Other connection mechanisms have been successfully deployed in production, but they're not part of the standard Always On architectures.

Choosing your architecture

All architectures provide the following:

- Hardware failure protection
- Zero downtime upgrades
- Support for availability zones in public/private cloud

Use these criteria to help you to select the appropriate Always On architecture.

	Bronze	Silver	Gold	Platinum
Minimum Locations Needed	1	2	3	2
Location failure protection	No - unless offsite backup	Yes - Recovery from backup	Yes - instant failover to fully functional site	Yes - instant failover to fully functional site
Failover to DR or full DC	NA (DR only if offsite backup)	DR using offsite backup	Full DC	Full DC
Fast local restoration of high availability after device failure	No; time to restore HA: (1) VM prov + (2) approx 60 min/500GB	Yes; three local data nodes allow to maintain HA after device failure	No; time to restore HA: (1) VM prov + (2) approx 60 min/500GB	Yes; logical standbys can quickly be promoted to master data nodes
Cross location network traffic	None (unless offsite backup then backup traffic only)	Backup traffic only	Full replication traffic	Full replication traffic
License cost	2 data nodes	3 data nodes	4 data nodes	4 data nodes 2 logical standbys

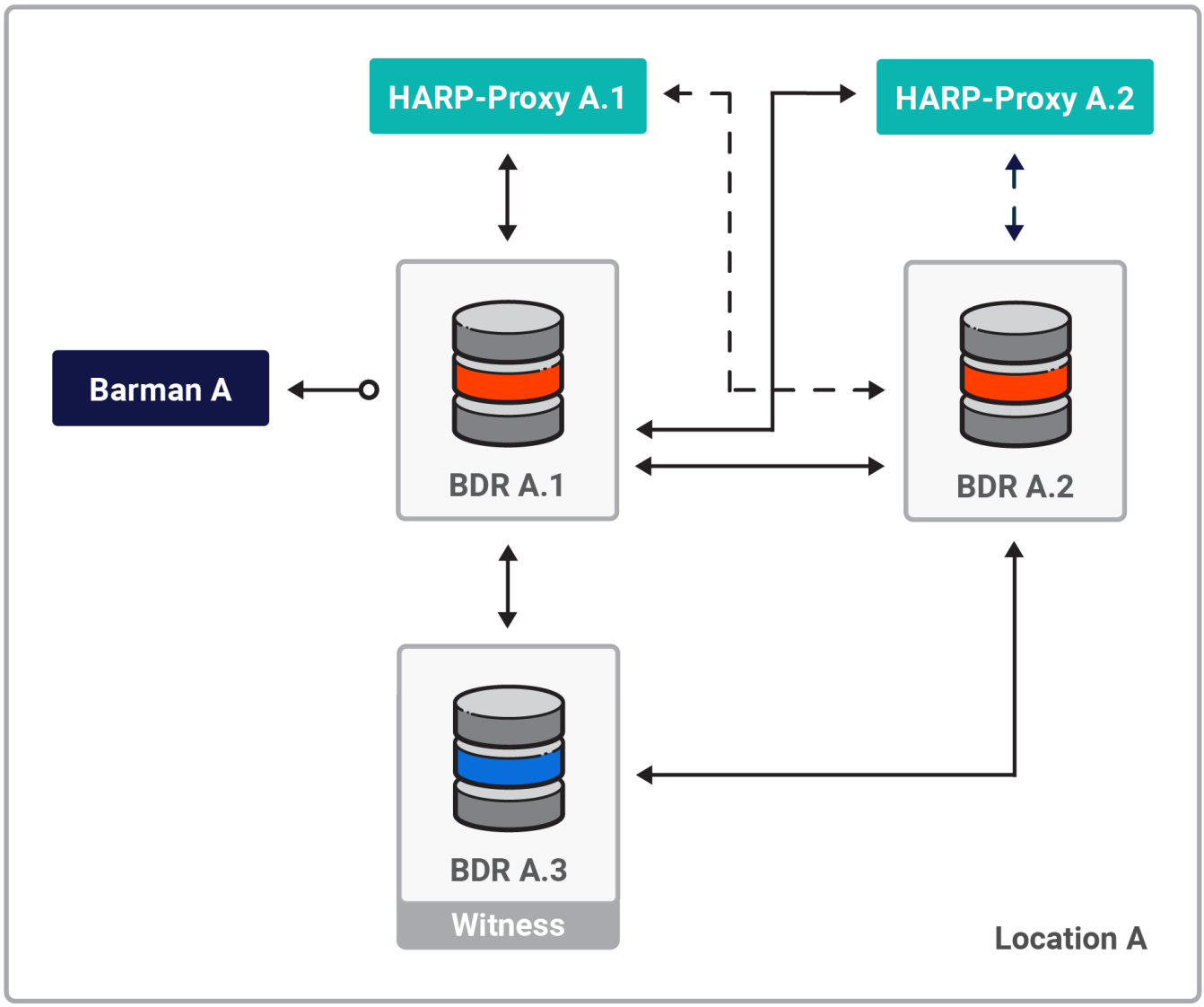
8.1 Always On Bronze

The Always On Bronze architecture includes the following:

- Two BDR data nodes
- One BDR witness node that doesn't hold data but is used for consensus
- Two HARP-Proxy nodes for routing application traffic to the "lead" master
- One barman node for backup and recovery can be onsite or offsite

BDR and HARP-Proxy nodes should be spread across availability zones.

Always On Bronze



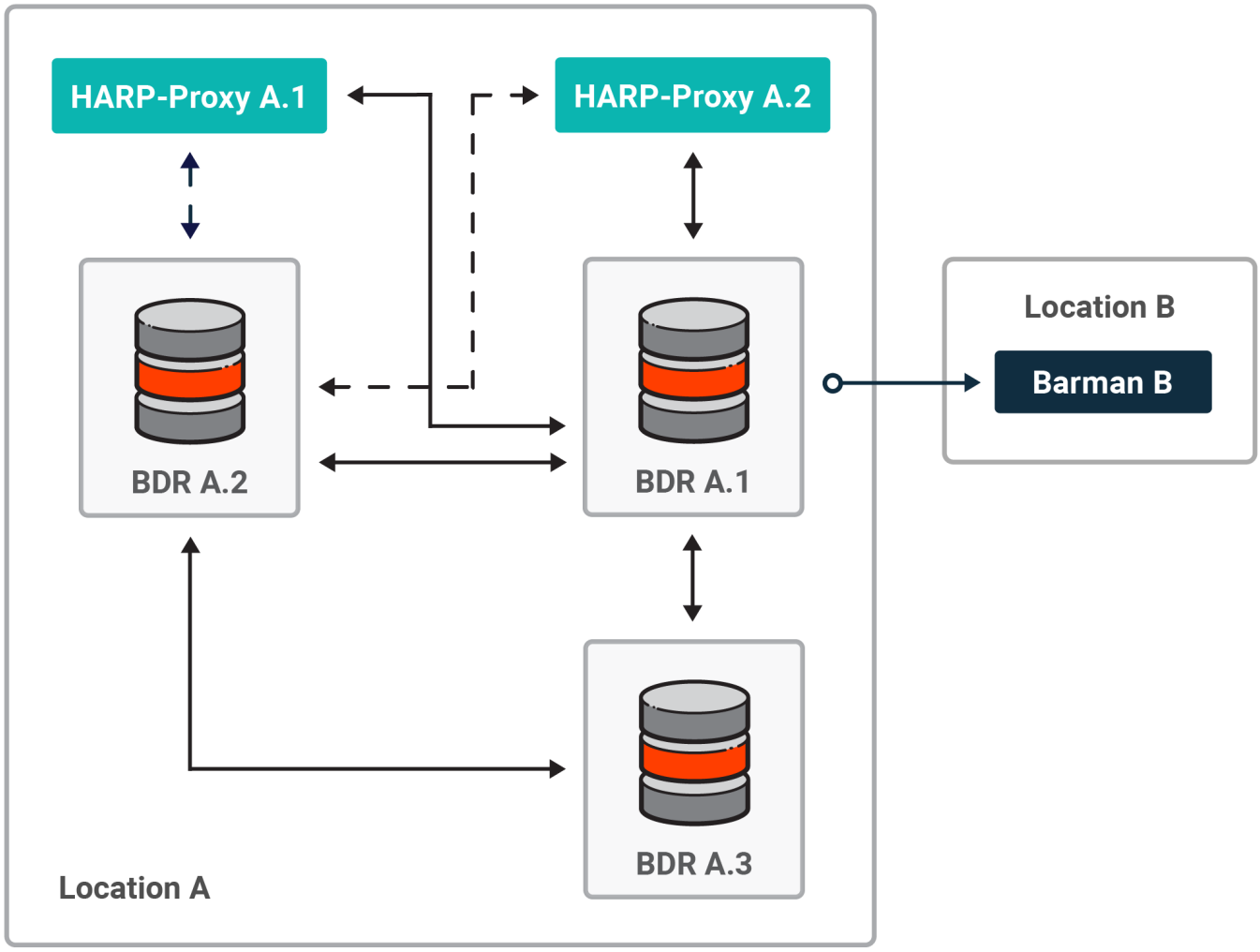
8.2 Always On Silver

The Always On Silver architecture includes the following:

- Three BDR data nodes
- Two HARP-Proxy nodes
- Barman offsite (offsite is optional but recommended)

BDR and HARP nodes should be spread across AZs. Barman can be located in location A.

Always On Silver



8.3 Always On Gold

This architecture favors local resiliency/redundancy first and remote locations only when and an entire location is offline or fails.

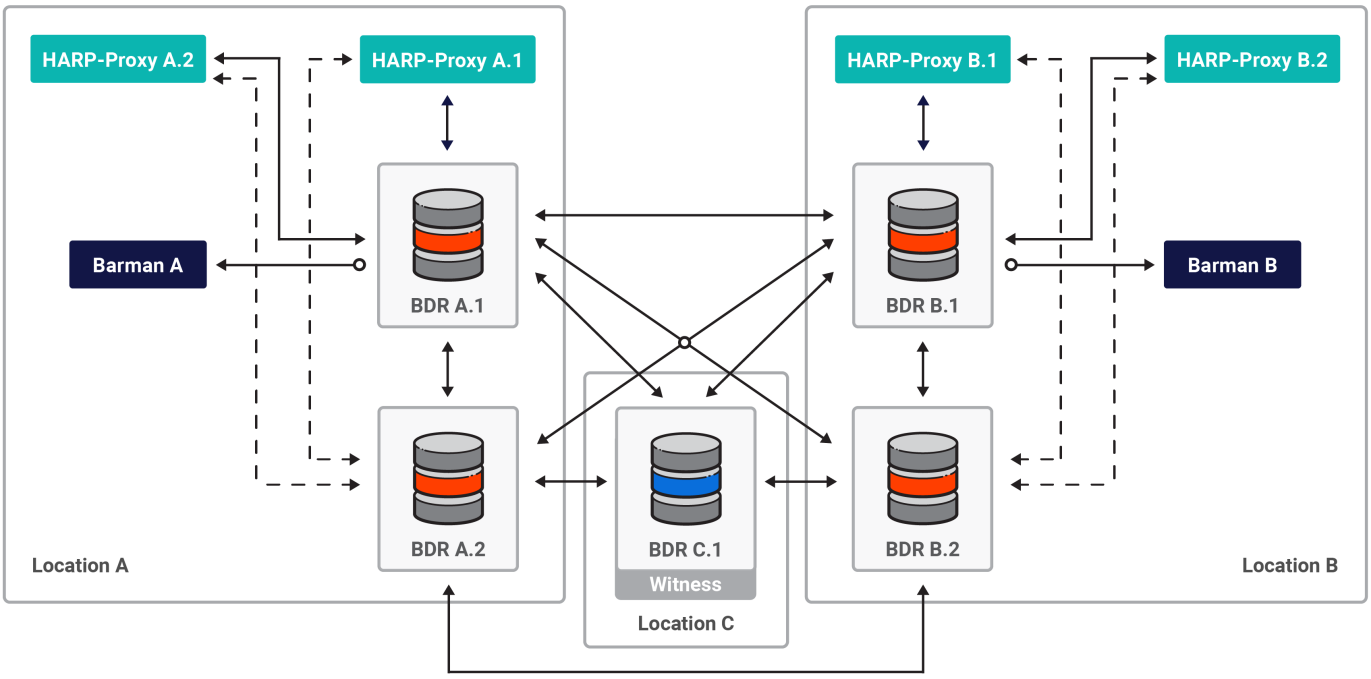
This architecture enables geo-distributed writes where no/low conflict handling is expected

The Always On Gold architecture requires intervention to move between locations but all data will be replicated and available to the application when failover is initiated with full capacity.

The Always On Gold architecture includes the following:

- Four BDR data nodes (two in location A, two in location B)
- One BDR witness node in location C (optional but recommended)
- Four HARP-Proxy nodes (two in location A, two in location B)
- Two Barman nodes (one in location A, one in location B)

Always On Gold 3DC, 2 Lead Masters



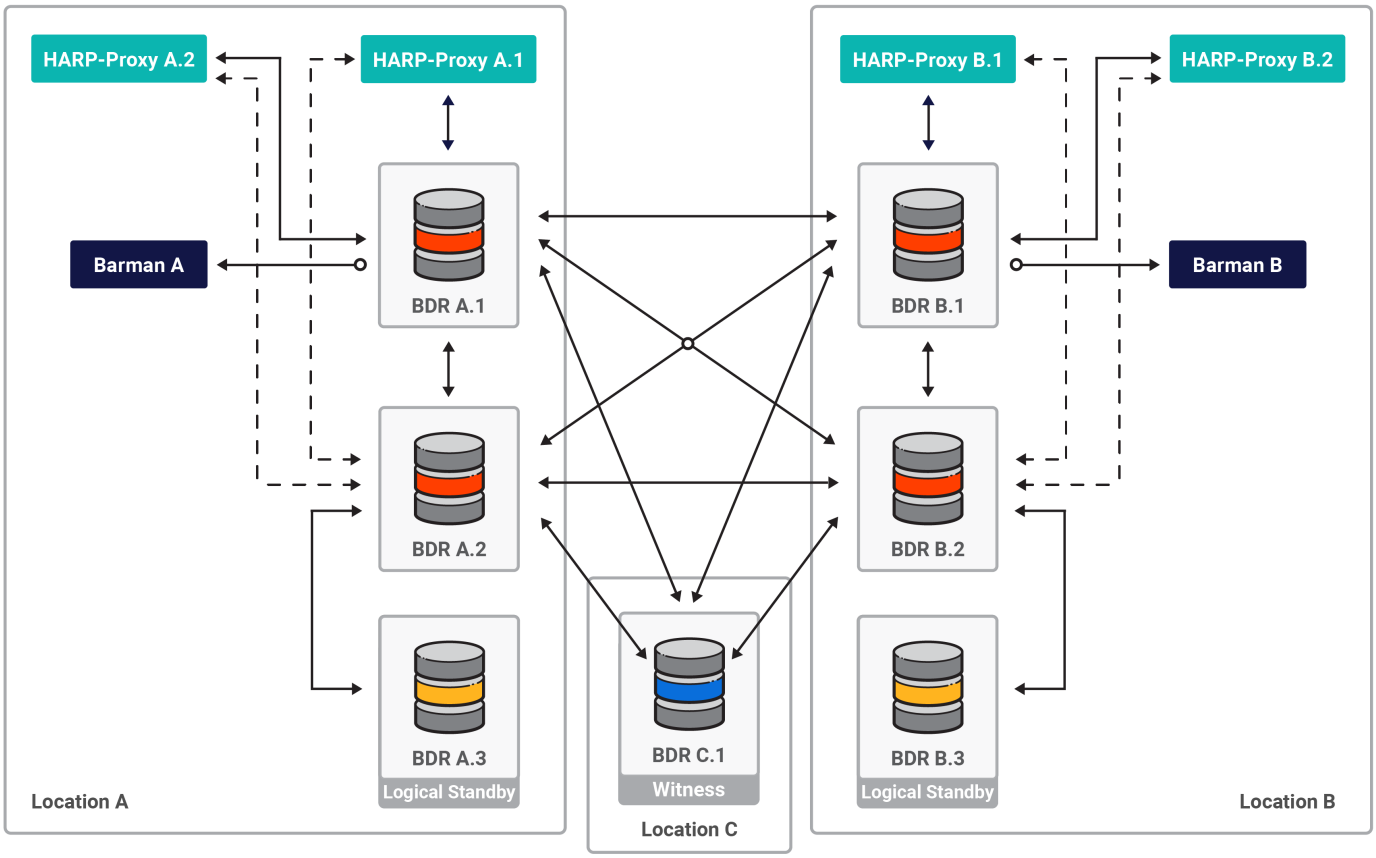
8.4 Always On Platinum

The Always On Platinum architecture includes the following:

- Four BDR data nodes (two in location A, two in location B)
- Two logical standby nodes (one in location A, one in location B)
- One witness node in location C (optional but recommended)
- Four HARP-Proxy nodes (two in location A, two in location B)
- Two Barman nodes (one in location A, one in location B)

Both locations have redundant hot standby hardware to maintain local high availability in case of a hardware failure without waiting to restore a BDR node from backup.

Always On Platinum



9 Choosing a Postgres distribution

EDB Postgres Distributed can be deployed with three different Postgres distributions: PostgreSQL, EDB Postgres Extended Server, or EDB Postgres Advanced Server. The availability of particular EDB Postgres Distributed features depends on which Postgres distribution is used. Therefore, it is essential to adopt the Postgres distribution best suited to your business needs. For example, if having the feature "Commit At Most Once (CAMO)" is mission critical to your use case, you should not adopt open source PostgreSQL because it does not have the core capabilities required to handle CAMO.

The following table lists features of EDB Postgres Distributed that are dependent on the Postgres distribution and version.

Feature	PostgreSQL	EDB Postgres Extended	EDB Postgres Advanced
Commit At Most Once (CAMO)	N	Y	14+
Eager Replication	N	Y	14+
Decoding Worker	N	13+	14+
Assessment Tooling	N	Y	14+
Lag Tracker	N	Y	14+
Lag Control	N	Y	14+
Timestamp Snapshots	N	Y	14+

Feature	PostgreSQL	EDB Postgres Extended	EDB Postgres Advanced
Transaction Streaming	14+	13+	14+
Missing Partition Conflict	N	Y	14+
No need for UPDATE Trigger on tables with TOAST	N	Y	14+
Automatically hold back FREEZE	N	Y	14+

10 Choosing durability

EDB Postgres Distributed allows you to choose from several replication configurations based on your consistency, availability, and performance needs:

- Asynchronous
- Synchronous (using `synchronous_standby_names`)
- Commit at Most Once
- Eager
- Group Commit

For more information, see Durability.

11 Other considerations

Review these other considerations when planning your deployment.

Deployment and sizing considerations

For production deployments, EDB recommends a minimum of 12 cores for each Postgres data node and each logical standby. Witness nodes don't participate in the data replication operation and don't have to meet this requirement. Always size logical standbys exactly like the data nodes to avoid performance degradations in case of a node promotion. In production deployments, HARP proxy nodes require a minimum of four cores each. EDB recommends detailed benchmarking of performance requirements, working with EDB's Professional Services team.

For development purposes, don't assign Postgres data nodes fewer than two cores. The sizing of Barman nodes depends on the database size and the data change rate.

You can deploy Postgres data nodes, logical standbys, Barman nodes, and HARP proxy nodes on virtual machines or in a bare metal deployment mode. However, maintain anti-affinity between data nodes and HARP proxy nodes. Don't deploy multiple data nodes on VMs that are on the same physical hardware, as that reduces resiliency. Also don't deploy multiple HARP proxy nodes on VMs on the same physical hardware, as that, too, reduces resiliency.

You can deploy single HARP proxy nodes with single data nodes on the same physical hardware but should be deployed as VMs to ensure proper CPU and memory resource assignment.

Clocks and timezones

EDB Postgres Distributed 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 EDB Postgres Distributed provides controls to report and manage any skew that exists. EDB Postgres Distributed also provides Row Version Conflict Detection, as described in Conflict Detection.

12 Deployment options

You can deploy and install EDB Postgres Distributed products using the following methods:

- TPAexec is an orchestration tool that uses Ansible to build Postgres clusters as specified by TPA (Trusted Postgres Architecture), a set of reference architectures that document how to set up and operate Postgres in various scenarios. TPA represents the best practices followed by EDB, and its recommendations are as applicable to quick testbed setups as to production environments.

Coming soon:

- BigAnimal is a fully managed database-as-a-service with built-in Oracle compatibility, running in your cloud account and operated by the Postgres experts. BigAnimal makes it easy to set up, manage, and scale your databases. The addition of extreme high availability support through EDB Postres Distributed allows single-region Always On Gold clusters: two BDR groups in different availability zones in a single cloud region, with a witness node in a third availability zone.

- EDB Postgres Distributed for Kubernetes will be a Kubernetes operator is designed, developed, and supported by EDB that covers the full lifecycle of a highly available Postgres database clusters with a multi-master architecture, using BDR replication. It is based on the open source CloudNativePG operator, and provides additional value such as compatibility with Oracle using EDB Postgres Advanced Server and additional supported platforms such as IBM Power and OpenShift.

12.1 TPAexec

The standard way of deploying EDB Postgres Distributed in a self managed setting, including physical and virtual machines, both self-hosted and in the cloud (EC2), is to use EDB's deployment tool called TPAexec.

TPAexec is an orchestration tool that uses Ansible to build Postgres clusters as specified by Trusted Postgres Architecture (TPA), a set of reference architectures that documents how to set up and operate Postgres in various scenarios. TPA represents the best practices followed by EDB and its recommendations are as applicable to quick testbed setups as to production environments.

TPAexec packages are only available to EDB customers with an active BDR subscription. Please contact your account

manager to request access.

Incremental changes

TPAexec is carefully designed so that provisioning, deployment, and testing are idempotent. You can run through them, make a change to config.yml, and run through the process again to deploy the change. If nothing has changed in the configuration or on the instances, then rerunning the entire process does not change anything either.

Extensible through Ansible

TPAexec supports a variety of configuration options, so you can do a lot just by editing config.yml and re-running provision/deploy/test. Should you need to go beyond what is already implemented, you can write hook scripts to extend the deployment process.

This mechanism places the full range of Ansible functionality at your disposal during every stage of the deployment. For example, any tasks in hooks/pre-deploy.yml are executed before the main deployment; and there are also post-deploy and many other hooks.

Cluster management

Once your cluster is up and running, TPAexec provides convenient cluster management functions, including configuration changes, switchover, and zero-downtime minor-version upgrades. These features make it easier and safer to manage your cluster than making the changes by hand.

It's just Postgres

TPAexec can create complex clusters with many features configured, but the result is just Postgres. The installation follows some conventions designed to make life simpler, but there is no hidden magic or anything standing in the way between you and the database. You can do everything on a TPA cluster that you could do on any other Postgres installation.

Getting started

See these topics to install TPAexec and use it to deploy your first EDB Postgres Distributed cluster:

- Installing TPAexec
- Using TPAexec
- Example: Deploying EDB Postgres Distributed

For more information, see TPAexec.

12.1.1 Installing TPAexec

The TPAexec install package in available from the EDB customer portal. You must have a subscription to access the repository and the TPAexec install package.

Prerequisites

Access

Setting up the repository requires root user permissions. If you do not have direct root access, log in as superuser.

```
## To log in as a superuser:
sudo su -
```

If you are behind an HTTPS proxy, see How to install RPM/APT repositories through a HTTPS proxy.

Token

You need your token to complete the install process. It is available from the EnterpriseDB customer portal. From the left navigation pane, select Company info > Company. The account information for your company displays. Copy your token.

Installing TPAexec

The examples in this section assume you are using TPAexec v22.14 and installing Postgres 14.

1. Install the EnterpriseDB repository for TPAexec. Replace `token` with your company token. Replace `pgVersion` with the version of Postgres you are installing.

   Syntax:

   ```
   curl
   https://techsupport.enterprisedb.com/api/repository/<token>/products/tpa/release/<p
   m | bash
   ```

   Example:

   ```
   curl
   https://techsupport.enterprisedb.com/api/repository/xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
   m | bash
   ```

2. Install TPAexec:

   - On RedHat

     ```
     yum install tpaexec
     ```

   - On Debian/Ubuntu

     ```
     apt-get install tpaexec
     ```

3. Install additional dependencies:

   ```
   /opt/EDB/TPA/bin/tpaexec setup
   ```

4. Verify the installation:

```
/opt/EDB/TPA/bin/tpaexec selftest
```

12.1.2 Using TPAexec

With TPAexec you configure, provision, and deploy your EDB Postgres Distributed clusters.

Configure

The `tpaexec configure` command generates a simple YAML configuration file to describe a cluster, based on the options you select. The configuration is ready for immediate use and you can modify it to better suit your needs. Editing the configuration file is the usual way to make any configuration changes to your cluster both before and after it's created.

The syntax is:

```
tpaexec configure <cluster_dir> --architecture <architecture_name> --layout
<layout> [options]
```

The required configuration options include:

Flags	Description
`--architecture`	Required. Set to `BDR-Always-ON` for EDB Postgres Distributed deployments.
`--layout`	Required. Specify one of the four supported architectures: bronze, silver, gold, and platinum. See Choosing your architecture for more information.
`--harp-consensus-protocol`	Required. Use `bdr` if your layout is bronze or silver. Use `etcd` if your layout is gold or platinum.

For example:

```
[tpa]$ tpaexec configure ~/clusters/speedy \
         --architecture BDR-Always-ON \
         --layout gold \
         --harp-consensus-protocol etcd \
```

The first argument must be the cluster directory, for example, `speedy` or `~/clusters/speedy` (the cluster is named `speedy` in both cases). We recommend that you keep all your clusters in a common directory, for example, `~/clusters`. The next argument must be `--architecture` to select an architecture, followed by `--layout`.

The command creates a directory named ~/clusters/speedy and generates a configuration file named `config.yml` that follows the layout of the BDR-Always-ON architecture and gold layout. You can use the `tpcaexec info` command to see what values are supported for the configuration options based on what you specified when running the configure command.

Common configuration options

Other configuration options include:

Owner

Every cluster must be directly traceable to a person responsible for the provisioned resources.

By default, a cluster is tagged as being owned by the login name of the user running `tpaexec provision`. If this name does not identify a person (for example, `postgres`, `ec2-user`), you must specify `--owner SomeId` to set an identifiable owner.

You may use your initials, or "Firstname Lastname", or anything else that identifies you uniquely.

Platform options

The default value for `--platform` is `aws`. It is the platform supported by the BDR-Always-ON architecture.

Specify `--region` to specify any existing AWS region that you have access to (and that permits the required number of instances to be created). The default region is eu-west-1.

Specify `--instance-type` with any valid instance type for AWS. The default is t3.micro.

Subnet selection

By default, each cluster is assigned a random /28 subnet under 10.33/16, but depending on the architecture, there may be one or more subnets, and each subnet may be anywhere between a /24 and a /29.

Specify `--subnet` to use a particular subnet. For example, `--subnet 192.0.2.128/27`.

Alternatively, specify `--subnet-pattern` to generate random subnets (as many as required by the architecture) matching the given pattern. For example, `--subnet-pattern 192.0.x.x`.

Disk space

Specify `--root-volume-size` to set the size of the root volume in GB. For example, `--root-volume-size 64`. The default is 16GB. (Depending on the image used to create instances, there may be a minimum size for the root volume.)

For architectures that support separate postgres and barman volumes:

Specify `--postgres-volume-size` to set the size of the Postgres volume in GB. The default is 16GB.

Specify `--barman-volume-size` to set the size of the Barman volume in GB. The default is 32GB.

Distribution

Specify `--os` or `--distribution` to specify the OS to be used on the cluster's instances. The value is case-sensitive.

The selected platform determines which distributions are available and which one is used by default. For more details, see `tpaexec info platforms/<platformname>`.

In general, you can use "Debian", "RedHat", and "Ubuntu" to select TPA images that have Postgres and other software preinstalled (to reduce deployment times). To use stock distribution images instead, append "-minimal" to the value, for example, `--distribution Debian-minimal`.

2ndQuadrant repositories

By default, TPAexec installs the 2ndQuadrant public repository and adds on any product repositories that the architecture requires.

Specify `--2Q-repositories source/name/release …` to specify the complete list of 2ndQuadrant repositories to install on each instance in addition to the 2ndQuadrant public repository.

If you do this, you must first export TPA_2Q_SUBSCRIPTION_TOKEN=xxx before you run tpaexec. You can get a subscription token from the EnterpriseDB customer portal (Support > Software subscriptions > Add).

Software versions

By default TPAexec uses the latest major version of Postgres. Specify `--postgres-version` to install an earlier supported major version.

By default, TPAexec always installs the latest version of every package. This is usually the desired behavior, but in some testing scenarios, it may be necessary to select specific package versions. For example,

```
--postgres-package-version 10.4-2.pgdg90+1
--repmgr-package-version 4.0.5-1.pgdg90+1
--barman-package-version 2.4-1.pgdg90+1
--pglogical-package-version '2.2.0*'
--bdr-package-version '3.0.2*'
--pgbouncer-package-version '1.8*'
```

Specify `--extra-packages` or `--extra-postgres-packages` to install additional packages. The former lists packages to install along with system packages, while the latter lists packages to install later along with postgres packages. (If you mention packages that depend on Postgres in the former list, the installation fails because Postgres is not yet installed.) The arguments are passed on to the package manager for installation without any modifications.

The `--extra-optional-packages` option behaves like `--extra-packages`, but it is not an error if the named packages cannot be installed.

Hostnames

By default, tpaexec configure randomly selects as many hostnames as it needs from a pre-approved list of several dozen names. This should be enough for most clusters.

Specify `--hostnames-from` to select names from a different list (for example, if you need more names than are available in the canned list). The file must contain one hostname per line.

Specify `--hostnames-pattern` to restrict hostnames to those matching the egrep-syntax pattern. If you choose to do this, you must ensure that the pattern matches only valid hostnames ([a-zA-Z0-9-]) and finds a sufficient number thereof.

Locations

By default, `tpaexec configure` uses the names first, second, and so on for any locations used by the selected architecture.

Specify `--location-names` to provide more meaningful names for each location.

Enable Commit At Most Once

Specify `--enable-camo` to set the pair of BDR primary instances in each region to be each other's Commit At Most Once (CAMO) partners. See Commit At Most Once (CAMO) for more information.

Provision

The `tpaexec provision` command creates instances and other resources required by the cluster. The details of the process depend on the architecture (for example, BDR-Always-ON) and platform (for example, AWS) that you selected while configuring the cluster.

For example, given AWS access with the necessary privileges, TPAexec provisions EC2 instances, VPCs, subnets, routing tables, internet gateways, security groups, EBS volumes, elastic IPs, and so on.

You can also "provision" existing servers by selecting the "bare" platform and providing connection details. Whether these are bare metal servers or those provisioned separately on a cloud platform, they can be used just as if they had been created by TPAexec.

You are not restricted to a single platform—you can spread your cluster out across some AWS instances (in multiple regions) and some on-premise servers, or servers in other data centres, as needed.

At the end of the provisioning stage, you will have the required number of instances with the basic operating system installed, which TPAexec can access via SSH (with sudo to root).

Deploy

The `tpaexec deploy` command installs and configures Postgres and other software on the provisioned servers (which may or may not have been created by TPAexec; but it doesn't matter who created them so long as SSH and sudo access is available). This includes setting up replication, backups, and so on.

At the end of the deployment stage, EDB Postgres Distributed is up and running.

Test

The `tpaexec test` command executes various architecture and platform-specific tests against the deployed cluster to ensure that it is working as expected.

At the end of the testing stage, you will have a fully-functioning cluster.

For more information, see TPAexec.

12.1.3 Example: Deploying EDB Postgres Distributed

The following steps setup EDB Postgres Distributed with the Always On Silver architecture using Amazon EC2.

1. Generate a configuration file:

   ```
   $ tpaexec configure myedbdpcluster --architecture BDR-Always-ON --layout Silver --platform aws --bdr-version 4 --2q --2Q-repositories products/2ndqpostgres/release
   ```

   This creates a subdirectory directory in current working directory called `myedbdpcluster` containing the `config.yml` configuration file TPAexec uses to create the cluster. Edit the `config.yml` as needed, for example to change the IP address range used for servers or adjust locations of nodes.

   We included options to specify using BDR 4 (the default version) and the 2ndQuadrant repositories to install EDB Postgres Extended Server. By default, TPAexec installs the 2ndQuadrant public repository and adds on any product repositories that the architecture requires. In this example, we specified `--2Q-repositories` so the complete list of 2ndQuadrant repositories is installed on each instance in addition to the 2ndQuadrant public repository. Since we did this, before the provisioning step, you must first export TPA_2Q_SUBSCRIPTION_TOKEN=xxx. You can get a subscription token from the EnterpriseDB customer portal (Support > Software subscriptions > Add).

   ```
   cd ~/clusters/myedbdpcluster

   export TPA_2Q_SUBSCRIPTION_TOKEN=<token>
   ```

2. Provision the cluster:

   ```
   tpaexec provision myedbdpcluster
   ```

   Since we specified AWS as the platform (the default platform), TPAexec provisions EC2 instances, VPCs, subnets, routing tables, internet gateways, security groups, EBS volumes, elastic IPs, and so on.

3. Deploy the cluster:

   ```
   tpaexec deploy myedbdpcluster
   ```

   TPAexec installs the needed packages, appliies the configuration and sets up the actual EDB Postgres Distributed cluster

4. Test the cluster:

   After the successful run of the `deploy` command the cluster is ready to use. You can connect to it via `psql` or any other database client.

   It's also possible to run a test that ensures the cluster is running as expected:

   ```
   tpaexec test myedbdpcluster
   ```

13 Upgrading

Because EDB Postgres Distributed consists in multiple software components, the upgrade strategy depends partially on which components are being upgraded.

In general it's possible to upgrade the cluster with almost zero downtime, by using an approach called Rolling Upgrade where nodes are upgraded one by one, and the application connections are switched over to already upgraded nodes.

Ii's also possible to stop all nodes, perform the upgrade on all nodes and only then restart the entire cluster, just like with a standard PostgreSQL setup. This strategy of upgrading all nodes at the same time avoids running with mixed versions of software and therefore is the simplest, but obviously incurs some downtime and is not recommended unless the Rolling Upgrade is not possible for some reason.

To upgrade an EDB Postgres Distributed cluster, perform the following steps:

1. Plan the upgrade.
2. Prepare for the upgrade.
3. Upgrade the server software.
4. Check and validate the upgrade.

Upgrade Planning

There are broadly two ways to upgrade each node.

- Upgrading nodes in-place to the newer software version, see Rolling Server Software Upgrades.
- Replacing nodes with ones that have the newer version installed, see Rolling Upgrade Using Node Join.

Both of these approaches can be done in a rolling manner.

Rolling Upgrade considerations

While the cluster is going through a rolling upgrade, mixed versions of software are running in the cluster. For example, nodeA has BDR 3.7.16, while nodeB and nodeC has 4.1.0. In this state, the replication and group management uses the protocol and features from the oldest version (3.7.16 in case of this example), so any new features provided by the newer version which require changes in the protocol are disabled. Once all nodes are upgraded to the same version, the new features are automatically enabled.

Similarly, when a cluster with WAL decoder enabled nodes is going through a rolling upgrade, WAL decoder on a higher version of BDR node produces LCRs with a higher pglogical version and WAL decoder on a lower version of BDR node produces LCRs with lower pglogical version. As a result, WAL senders on a higher version of BDR nodes are not expected to use LCRs due to a mismatch in protocol versions while on a lower version of BDR nodes, WAL senders may continue to use LCRs. Once all the BDR nodes are on the same BDR version, WAL senders use LCRs.

A rolling upgrade starts with a cluster with all nodes at a prior release, then proceeds by upgrading one node at a time to the newer release, until all nodes are at the newer release. There should never be more than two versions of any component running at the same time, which means the new upgrade must not be initiated until the previous upgrade process has fully finished on all nodes.

An upgrade process may take an extended period of time when the user decides caution is required to reduce business risk, though it's not recommended to run the mixed versions of the software indefinitely.

While Rolling Upgrade can be used for upgrading major version of the software it is not supported to mix PostgreSQL, EDB Postgres Extended and EDB Postgres Advanced Server in one cluster, so this approach cannot be used to change the Postgres variant.

!!! Warning Downgrades of the EDB Postgres Distributed are *not* supported and require manual rebuild of the cluster.

Rolling Server Software Upgrades

A rolling upgrade is the process where the Server Software Upgrade process is performed on each node in the cluster one after another, while keeping the remainder of the cluster operational.

The actual procedure depends on whether the Postgres component is being upgraded to a new major version or not.

During the upgrade process, the application can be switched over to a node which is currently not being upgraded to provide continuous availability of the database for applications.

Rolling Upgrade Using Node Join

The other method of upgrade of the server software, is to join a new node to the cluster and later drop one of the existing nodes running the older version of the software.

For this approach, the procedure is always the same, however because it includes node join, the potentially large data transfer is required.

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

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

Upgrading a CAMO-Enabled Cluster

CAMO protection requires at least one of the nodes of a CAMO pair to be operational. For upgrades, we recommend to ensure that no CAMO protected transactions are running concurrent to the upgrade, or to use a rolling upgrade strategy, giving the nodes enough time to reconcile in between the upgrades and the corresponding node downtime due to the upgrade.

Upgrade Preparation

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

Please see individual changes mentioned in release notes and any version specific upgrade notes in this topic.

Server Software Upgrade

The upgrade of EDB Postgres Distributed on individual nodes happens in-place. There is no need for backup and restore when upgrading the BDR extension.

BDR Extension Upgrade

BDR extension upgrade process consists of few simple steps.

Stop Postgres

During the upgrade of binary packages, it's usually best to stop the running Postgres server first to ensure that mixed versions don't get loaded in case of unexpected restart during the upgrade.

Upgrade Packages

The first step in the upgrade is to install the new version of the BDR packages, which installs both the new binary and the extension SQL script. This step is operating system-specific.

Start Postgres

Once packages are upgraded the Postgres instance can be started, the BDR extension is automatically upgraded upon start when the new binaries detect older version of the extension.

Postgres Upgrade

The process of in-place upgrade of Postgres highly depends on whether you are upgrading to new minor version of Postgres of to new major version of Postgres.

Minor Version Postgres Upgrade

Upgrading to a new minor version of Postgres is similar to upgrading the BDR extension. Stopping Postgres, upgrading packages, and starting Postgres again is typically all that's needed.

However, sometimes additional steps like reindexing may be recommended for specific minor version upgrades. Refer to the Release Notes of the specific version of Postgres you are upgrading to.

Major Version Postgres Upgrade

Upgrading to a new major version of Postgres is a more complicated process.

EDB Postgres Distributed provides a `bdr_pg_upgrade` command line utility, which can be used to do a In-place Postgres Major Version Upgrades.

!!! Note When upgrading to new major version of any software, including Postgres, the BDR extension and others, it's always important to ensure the compatibility of your application with the target version of a given software.

Upgrade Check and Validation

After this procedure, your BDR node is upgraded. You can verify the current version of BDR4 binary like this:

```
SELECT bdr.bdr_version();
```

Always check the monitoring after upgrade of a node to confirm that the upgraded node is working as expected.

Application Schema Upgrades

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

To eliminate this downtime, BDR offers ways to perform a rolling application schema upgrade.

Rolling Application Schema Upgrades

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

!!! Warning Rolling Application Schema Upgrades have to be managed outside of BDR. Careful scripting is required to make this work correctly on production clusters. Extensive testing is advised.

See Replicating between nodes with differences for details.

When one node runs DDL that adds a new table, nodes that have not yet received the latest DDL need to handle the extra table. In view of this, the appropriate setting for rolling schema upgrades is to configure all nodes to apply the `skip` resolver in case of a `target_table_missing` conflict. This must be performed before any node has additional tables added and is intended to be a permanent setting.

This is done with the following query, that must be executed separately on each node, after replacing `node1` with the actual node name:

```
SELECT bdr.alter_node_set_conflict_resolver('node1',
       'target_table_missing', 'skip');
```

When one node runs DDL that adds a column to a table, nodes that have not yet received the latest DDL need to handle the extra columns. In view of this, the appropriate setting for rolling schema upgrades is to configure all nodes to apply the `ignore` resolver in case of a `target_column_missing` conflict. This must be performed before one node has additional columns added and is intended to be a permanent setting.

This is done with the following query, that must be executed separately on each node, after replacing `node1` with the actual node name:

```
SELECT bdr.alter_node_set_conflict_resolver('node1',
       'target_column_missing', 'ignore');
```

When one node runs DDL that removes a column from a table, nodes that have not yet received the latest DDL need to handle the missing column. This situation will cause a `source_column_missing` conflict, which uses the `use_default_value` resolver. Thus, columns that neither accept NULLs nor have a DEFAULT value require a two step process:

1. Remove NOT NULL constraint or add a DEFAULT value for a column on all nodes.
2. Remove the column.

Constraints can be removed in a rolling manner. There is currently no supported way for handling adding table constraints in a rolling manner, one node at a time.

When one node runs a DDL that changes the type of an existing column, depending on the existence of binary coercibility between the current type and the target type, the operation may not rewrite the underlying table data. In that

case, it will be only a metadata update of the underlying column type. Rewrite of a table is normally restricted. However, in controlled DBA environments, it is possible to change the type of a column to an automatically castable one by adopting a rolling upgrade for the type of this column in a non-replicated environment on all the nodes, one by one. See ALTER TABLE for more details. section.

13.1 In-place Postgres Major Version Upgrades

Upgrading a BDR Node to a newer major version of Postgres is possible using the command-line utility `bdr_pg_upgrade`.

`bdr_pg_upgrade` internally uses the standard `pg_upgrade` with BDR specific logic to ensure a smooth upgrade.

Terminology

Various terminology is used in this documentation to describe the upgrade process and components involved.

old cluster - The existing Postgres cluster node to be upgraded, which data will be migrated from.

new cluster - The new Postgres cluster, which data will be migrated to. This cluster node must be one (1) major version ahead of the old cluster.

Precautions

Standard Postgres major version upgrade precautions apply, including the fact that all the requirements for `pg_upgrade` must be met by both clusters.

Additionaly, `bdr_pg_upgrade` should not be used if there are other tools using replication slots and replication origins, only BDR slots and origins will be restored after the upgrade.

There are several prerequisites for `bdr_pg_upgrade` that have to be met:

- Applications using the old cluster have been disconnected, it can for example, be redirected to another node in the cluster
- Peer authentication is configured for both clusters, `bdr_pg_upgrade` requires peer authentication
- BDR versions on both clusters must be exactly the same and must be version 4.1.0 or above
- The new cluster must be in a shutdown state
- BDR packages must be installed in the new cluster
- The new cluster must be already initialized and configured as needed to match the old cluster configuration
- Databases, tables, and other objects must not exist in the new cluster

It is also recommended to have the old cluster up prior to running `bdr_pg_upgrade` as the CLI will start the old cluster if it is shutdown.

Usage

To upgrade to a newer major version of Postgres, the new version must first be installed.

bdr_pg_upgrade command-line

`bdr_pg_upgrade` passes all parameters to `pg_upgrade`. Therefore, you can specify any parameters supported by `pg_upgrade`.

Synopsis

```
bdr_pg_upgrade [OPTION] ...
```

Options

In addition to the options for `pg_upgrade`, the following parameters are can be passed to `bdr_pg_upgrade`:

- `-b, --old-bindir` - old cluster bin directory (required)
- `-B, --new-bindir` - new cluster bin directory (required)
- `-d, --old-datadir` - old cluster data directory (required)
- `-D, --new-datadir` - `REQUIRED` new cluster data directory (required)
- `--database` - BDR database name (required)
- `-p, --old-port` - old cluster port number
- `-s, --socketdir` - directory to use for postmaster sockets during upgrade
- `--check` - only perform checks, do not modify clusters

Environment Variables

Environment variables can be used in place of command line parameters.

- `PGBINOLD` - old cluster bin directory
- `PGBINNEW` - new cluster bin directory
- `PGDATAOLD` - old cluster data directory
- `PGDATANEW` - new cluster data directory
- `PGPORTOLD` - old cluster port number
- `PGSOCKETDIR` - directory to use for postmaster sockets during upgrade

Example

Given a scenario where:

- Old cluster bin directory is `/usr/lib/postgresql/13/bin`
- New cluster bin directory is `/usr/lib/postgresql/14/bin`
- Old cluster data directory is `/var/lib/postgresql/13/main`
- New cluster data directory is `/var/lib/postgresql/14/main`
- Database name is `bdrdb`

The following command could be used to upgrade the cluster:

```
bdr_pg_upgrade \
--old-bindir /usr/lib/postgresql/13/bin \
--new-bindir /usr/lib/postgresql/14/bin \
--old-datadir /var/lib/postgresql/13/main \
--new-datadir /var/lib/postgresql/14/main \
--database bdrdb
```

Steps Performed

Steps performed when running `bdr_pg_upgrade`.

!!! Note When `--check` is supplied as an argument to `bdr_pg_upgrade`, the CLI will 1skip` steps that modify the database.

BDR Postgres Checks

Steps	`--check` supplied
Collecting pre-upgrade new cluster control data	`run`
Checking new cluster state is shutdown	`run`
Checking BDR versions	`run`
Starting old cluster (if shutdown)	`skip`
Connecting to old cluster	`skip`
Checking if bdr schema exists	`skip`
Turning DDL replication off	`skip`
Terminating connections to database.	`skip`
Disabling connections to database	`skip`
Waiting for all slots to be flushed	`skip`
Disconnecting from old cluster	`skip`
Stopping old cluster	`skip`
Starting old cluster with BDR disabled	`skip`
Connecting to old cluster	`skip`
Collecting replication origins	`skip`
Collecting replication slots	`skip`
Disconnecting from old cluster	`skip`
Stopping old cluster	`skip`

`pg_upgrade` Steps

Standard `pg_upgrade` steps are performed

!!! Note `--check` is passed to pg_upgrade if supplied

BDR Post-Upgrade Steps

Steps	`--check` supplied
Collecting old cluster control data	`skip`
Collecting new cluster control data	`skip`
Advancing LSN of new cluster	`skip`
Starting new cluster with BDR disabled	`skip`
Connecting to new cluster	`skip`
Creating replication origin Repeated for each origin	`skip`
Advancing replication origin Repeated for each origin	`skip`

Steps	`--check` supplied
Creating replication slot Repeated for each slot	`skip`
Stopping new cluster	`skip`

13.2 Supported BDR upgrade paths

3.7.13.1	3.7.14	3.7.15	3.7.16	4.0.0 and later	Target BDR version
		X	X	X	4.2.0
		X	X	X	4.1.1
		X	X	X	4.1.0
		X		X	4.0.2
	X			X	4.0.1
X					4.0.0

14 Backup and recovery

In this chapter we discuss the backup and restore of a EDB Postgres Distributed cluster.

BDR is designed to be a distributed, highly available system. If one or more nodes of a cluster are lost, the best way to replace them is to clone new nodes directly from the remaining nodes.

The role of backup and recovery in BDR is to provide for Disaster Recovery (DR), such as in the following situations:

- Loss of all nodes in the cluster
- Significant, uncorrectable data corruption across multiple nodes as a result of data corruption, application error or security breach

Backup

`pg_dump`

`pg_dump`, sometimes referred to as "logical backup", can be used normally with BDR.

Note that `pg_dump` dumps both local and global sequences as if they were local sequences. This is intentional, to allow a BDR schema to be dumped and ported to other PostgreSQL databases. This means that sequence kind metadata is lost at the time of dump, so a restore would effectively reset all sequence kinds to the value of `bdr.default_sequence_kind` at time of restore.

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

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

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

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

Physical Backup

Physical backups of a node in a EDB Postgres Distributed cluster can be taken using standard PostgreSQL software, such as Barman.

A physical backup of a BDR node can be performed with the same procedure that applies to any PostgreSQL node: a BDR node is just a PostgreSQL node running the BDR extension.

There are some specific points that must be considered when applying PostgreSQL backup techniques to BDR:

- BDR operates at the level of a single database, while a physical backup includes all the databases in the instance; you should plan your databases to allow them to be easily backed-up and restored.
- Backups will make a copy of just one node. In the simplest case, every node has a copy of all data, so you would need to backup only one node to capture all data. However, the goal of BDR will not be met if the site containing that single copy goes down, so the minimum should be at least one node backup per site (obviously with many copies etc.).
- However, each node may have un-replicated local data, and/or the definition of replication sets may be complex so that all nodes do not subscribe to all replication sets. In these cases, backup planning must also include plans for how to backup any unreplicated local data and a backup of at least one node that subscribes to each replication set.

Eventual Consistency

The nodes in a EDB Postgres Distributed cluster are *eventually consistent*, but not entirely *consistent*; a physical backup of a given node will provide Point-In-Time Recovery capabilities limited to the states actually assumed by that node (see the [Example] below).

The following example shows how two nodes in the same EDB Postgres Distributed cluster might not (and usually do not) go through the same sequence of states.

Consider a cluster with two nodes `N1` and `N2`, which is initially in state `S`. If transaction `W1` is applied to node `N1`, and at the same time a non-conflicting transaction `W2` is applied to node `N2`, then node `N1` will go through the following states:

```
(N1)   S  -->  S + W1  -->  S + W1 + W2
```

...while node `N2` will go through the following states:

```
(N2)   S  -->  S + W2  -->  S + W1 + W2
```

That is: node `N1` will *never* assume state `S + W2`, and node `N2` likewise will never assume state `S + W1`, but both nodes will end up in the same state `S + W1 + W2`. Considering this situation might affect how you decide upon your backup strategy.

Point-In-Time Recovery (PITR)

In the example above, the changes are also inconsistent in time, since `W1` and `W2` both occur at time `T1`, but the change `W1` is not applied to `N2` until `T2`.

PostgreSQL PITR is designed around the assumption of changes arriving from a single master in COMMIT order. Thus, PITR is possible by simply scanning through changes until one particular point-in-time (PIT) is reached. With this scheme, you can restore one node to a single point-in-time from its viewpoint, e.g. `T1`, but that state would not include other data from other nodes that had committed near that time but had not yet arrived on the node. As a result, the recovery might be considered to be partially inconsistent, or at least consistent for only one replication origin.

To request this, use the standard syntax:

```
recovery_target_time = T1
```

BDR allows for changes from multiple masters, all recorded within the WAL log for one node, separately identified using replication origin identifiers.

BDR allows PITR of all or some replication origins to a specific point in time, providing a fully consistent viewpoint across all subsets of nodes.

Thus for multi-origins, we view the WAL stream as containing multiple streams all mixed up into one larger stream. There is still just one PIT, but that will be reached as different points for each origin separately.

We read the WAL stream until requested origins have found their PIT. We apply all changes up until that point, except that we do not mark as committed any transaction records for an origin after the PIT on that origin has been reached.

We end up with one LSN "stopping point" in WAL, but we also have one single timestamp applied consistently, just as we do with "single origin PITR".

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

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

In the specific example above, `N1` would be restored to `T1`, but would also include changes from other nodes that have been committed by `T1`, even though they were not applied on `N1` until later.

To request multi-origin PITR, use the standard syntax in the recovery.conf file:

```
recovery_target_time = T1
```

The list of replication origins which would be restored to `T1` need either to be specified in a separate multi_recovery.conf file via the use of a new parameter `recovery_target_origins`:

```
recovery_target_origins = '*'
```

...or one can specify the origin subset as a list in `recovery_target_origins`.

```
recovery_target_origins = '1,3'
```

Note that the local WAL activity recovery to the specified `recovery_target_time` is always performed implicitly. For origins that are not specified in `recovery_target_origins`, recovery may stop at any point, depending on when the target for the list mentioned in `recovery_target_origins` is achieved.

In the absence of the `multi_recovery.conf` file, the recovery defaults to the original PostgreSQL PITR behaviour that is designed around the assumption of changes arriving from a single master in COMMIT order.

!!! Note This is feature is only available on EDB Postgres Extended and Barman does not currently automatically create a `multi_recovery.conf` file.

Restore

While you can take a physical backup with the same procedure as a standard PostgreSQL node, what is slightly more complex is restoring the physical backup of a BDR node.

EDB Postgres Distributed Cluster Failure or Seeding a New Cluster from a Backup

The most common use case for restoring a physical backup involves the failure or replacement of all the BDR nodes in a cluster, for instance in the event of a datacentre failure.

You may also want to perform this procedure to clone the current contents of a EDB Postgres Distributed cluster to seed a QA or development instance.

In that case, BDR capabilities can be restored based on a physical backup of a single BDR node, optionally plus WAL archives:

- If you still have some BDR nodes live and running, fence off the host you restored the BDR node to, so it cannot connect to any surviving BDR nodes. This ensures that the new node does not confuse the existing cluster.
- Restore a single PostgreSQL node from a physical backup of one of the BDR nodes.
- If you have WAL archives associated with the backup, create a suitable `recovery.conf` and start PostgreSQL in recovery to replay up to the latest state. You can specify a alternative `recovery_target` here if needed.
- Start the restored node, or promote it to read/write if it was in standby recovery. Keep it fenced from any surviving nodes!
- Clean up any leftover BDR metadata that was included in the physical backup, as described below.
- Fully stop and restart the PostgreSQL instance.
- Add further BDR nodes with the standard procedure based on the `bdr.join_node_group()` function call.

Cleanup BDR Metadata

The cleaning of leftover BDR metadata is achieved as follows:

1. Drop the BDR node using `bdr.drop_node`
2. Fully stop and re-start PostgreSQL (important!).

Cleanup of Replication Origins

Replication origins must be explicitly removed with a separate step because they are recorded persistently in a system catalog, and therefore included in the backup and in the restored instance. They are not removed automatically when dropping the BDR extension, because they are not explicitly recorded as its dependencies.

BDR creates one replication origin for each remote master node, to track progress of incoming replication in a crash-safe way. Therefore we need to run:

```
SELECT pg_replication_origin_drop('bdr_dbname_grpname_nodename');
```

...once for each node in the (previous) cluster. Replication origins can be listed as follows:

```
SELECT * FROM pg_replication_origin;
```

...and those created by BDR are easily recognized by their name, as in the example shown above.

Cleanup of Replication Slots

If a physical backup was created with `pg_basebackup`, replication slots will be omitted from the backup.

Some other backup methods may preserve replications slots, likely in outdated or invalid states. Once you restore the backup, just:

```
SELECT pg_drop_replication_slot(slot_name)
FROM pg_replication_slots;
```

...to drop *all* replication slots. If you have a reason to preserve some, you can add a `WHERE slot_name LIKE 'bdr%'` clause, but this is rarely useful.

!!! Warning Never run this on a live BDR node.

---

15 Monitoring

Monitoring replication setups is important to ensure that your system performs optimally and does not run out of disk space or encounter other faults that may halt operations.

It is important to have automated monitoring in place to ensure that if, for example, replication slots start falling badly behind, the administrator is alerted and can take proactive action.

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

Monitoring Overview

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

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

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

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

BDR allows you to monitor each of the nodes individually, or to monitor the whole group by access to a single node. If

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

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

Monitoring Node Join and Removal

By default, the node management functions wait for the join or part operation to complete. This can be turned off using the respective `wait_for_completion` function argument. If waiting is turned off, then to see when a join or part operation finishes, check the node state indirectly via `bdr.node_summary` and `bdr.state_journal_details`.

When called, the helper function `bdr.wait_for_join_completion()` will cause a PostgreSQL session to pause until all outstanding node join operations complete.

Here is an example output of a `SELECT` query from `bdr.node_summary` that indicates that two nodes are active and another one is joining:

```
## SELECT node_name, interface_connstr, peer_state_name,
##     node_seq_id, node_local_dbname
## FROM bdr.node_summary;
-[ RECORD 1 ]-----+----------------------------------------
node_name         | node1
interface_connstr | host=localhost dbname=postgres port=7432
peer_state_name   | ACTIVE
node_seq_id       | 1
node_local_dbname | postgres
-[ RECORD 2 ]-----+----------------------------------------
node_name         | node2
interface_connstr | host=localhost dbname=postgres port=7433
peer_state_name   | ACTIVE
node_seq_id       | 2
node_local_dbname | postgres
-[ RECORD 3 ]-----+----------------------------------------
node_name         | node3
interface_connstr | host=localhost dbname=postgres port=7434
peer_state_name   | JOINING
node_seq_id       | 3
node_local_dbname | postgres
```

Also, the table `bdr.node_catchup_info` will give information on the catch-up state, which can be relevant to joining nodes or parting nodes.

When a node is parted, it could be that some nodes in the cluster did not receive all the data from that parting node. So it will create a temporary slot from a node that already received that data and can forward it.

The `catchup_state` can be one of the following:

```
10 = setup
20 = start
30 = catchup
40 = done
```

Monitoring Replication Peers

There are two main views used for monitoring of replication activity:

- `bdr.node_slots` for monitoring outgoing replication
- `bdr.subscription_summary` for monitoring incoming replication

Most of the information provided by `bdr.node_slots` can be also obtained by querying the standard PostgreSQL replication monitoring views `pg_catalog.pg_stat_replication` and `pg_catalog.pg_replication_slots`.

Each node has one BDR group slot which should never have a connection to it and will very rarely be marked as active. This is normal, and does not imply something is down or disconnected. See `Replication Slots created by BDR`.

Monitoring Outgoing Replication

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

- `bdr.node_replication_rates` for monitoring outgoing replication

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

```
## SELECT * FROM bdr.node_replication_rates;
-[ RECORD 1 ]----+-----------
peer_node_id     | 112898766
target_name      | node1
sent_lsn         | 0/28AF99C8
replay_lsn       | 0/28AF99C8
replay_lag       | 00:00:00
replay_lag_bytes | 0
replay_lag_size  | 0 bytes
apply_rate       | 822
catchup_interval | 00:00:00
-[ RECORD 2 ]----+-----------
peer_node_id     | 312494765
target_name      | node3
sent_lsn         | 0/28AF99C8
replay_lsn       | 0/28AF99C8
replay_lag       | 00:00:00
replay_lag_bytes | 0
replay_lag_size  | 0 bytes
apply_rate       | 853
catchup_interval | 00:00:00
```

The `apply_rate` above refers to the rate in bytes per second. It is the rate at which the peer is consuming data from the local node. The `replay_lag` when a node reconnects to the cluster is immediately set to zero. We are working on fixing this information; as a workaround, we suggest you use the `catchup_interval` column that refers to the time required for the peer node to catch up to the local node data. The other fields are also available via the `bdr.node_slots` view, as explained below.

!!! Note This catalog is only present when bdr-enteprise extension is installed.

Administrators may query `bdr.node_slots` for outgoing replication from the local node. It shows information about replication status of all other nodes in the group that are known to the current node, as well as any additional replication slots created by BDR on the current node.

```
## SELECT node_group_name, target_dbname, target_name, slot_name, active_pid,
##     catalog_xmin, client_addr, sent_lsn, replay_lsn, replay_lag,
##     replay_lag_bytes, replay_lag_size
## FROM bdr.node_slots;
-[ RECORD 1 ]---+-----------------------------
node_group_name | bdrgroup
target_dbname   | postgres
target_name     | node3
slot_name       | bdr_postgres_bdrgroup_node3
active_pid      | 15089
catalog_xmin    | 691
client_addr     | 127.0.0.1
sent_lsn        | 0/23F7B70
replay_lsn      | 0/23F7B70
replay_lag      | [NULL]
replay_lag_bytes| 120
replay_lag_size | 120 bytes
-[ RECORD 2 ]---+-----------------------------
node_group_name | bdrgroup
target_dbname   | postgres
target_name     | node2
slot_name       | bdr_postgres_bdrgroup_node2
active_pid      | 15031
catalog_xmin    | 691
client_addr     | 127.0.0.1
sent_lsn        | 0/23F7B70
replay_lsn      | 0/23F7B70
replay_lag      | [NULL]
replay_lag_bytes| 84211
replay_lag_size | 82 kB
```

Note that because BDR is a mesh network, to get full view of lag in the cluster, this query has to be executed on all nodes participating.

`replay_lag_bytes` reports the difference in WAL positions between the local server's current WAL write position and `replay_lsn`, the last position confirmed replayed by the peer node. `replay_lag_size` is just a human-readable form of the same. It is important to understand that WAL usually contains a lot of writes that are not replicated but still count in `replay_lag_bytes`, including `VACUUM` activity, index changes, writes associated with other databases on the same node, writes for tables that are not part of a replication set, etc. So the lag in bytes reported here is not the amount of data that must be replicated on the wire to bring the peer node up to date, only the amount of server-side WAL that must be processed.

Similarly, `replay_lag` is not a measure of how long the peer node will take to catch up, or how long it will take to replay from its current position to the write position at the time `bdr.node_slots` was queried. It measures the delay between when the peer confirmed the most recent commit and the current wall-clock time. We suggest that you monitor `replay_lag_bytes` and `replay_lag_size` or `catchup_interval` in `bdr.node_replication_rates`, as this column is set to zero immediately after the node reconnects.

The lag in both bytes and time does not advance while logical replication is streaming a transaction. It only changes when a commit is replicated. So the lag will tend to "sawtooth", rising as a transaction is streamed, then falling again as the peer node commits it, flushes it, and sends confirmation. The reported LSN positions will "stair-step" instead of advancing smoothly, for similar reasons.

When replication is disconnected (`active` = `'f'`), the `active_pid` column will be `NULL`, as will `client_addr` and other fields that only make sense with an active connection. The `state` field will be `'disconnected'`. The `_lsn` fields will be the same as the `confirmed_flush_lsn`, since that is the last position that the client is known for certain to have replayed to and saved. The `_lag` fields will show the elapsed time between the most recent confirmed flush on the client and the current time, and the `_lag_size` and `_lag_bytes` fields will report the distance between `confirmed_flush_lsn` and the local server's current WAL insert position.

Note: It is normal for `restart_lsn` to be behind the other `lsn` columns; this does not indicate a problem with replication or a peer node lagging. The `restart_lsn` is the position that PostgreSQL's internal logical decoding must be reading WAL at if interrupted, and generally reflects the position of the oldest transaction that is not yet replicated and flushed. A very old `restart_lsn` can make replication slow to restart after disconnection and force retention of more WAL than is desirable, but will otherwise be harmless. If you are concerned, look for very long running transactions and forgotten prepared transactions.

Monitoring Incoming Replication

Incoming replication (also called subscription) can be monitored by querying the `bdr.subscription_summary` view. This shows the list of known subscriptions to other nodes in the EDB Postgres Distributed cluster and the state of the replication worker, e.g.:

```
## SELECT node_group_name, origin_name, sub_enabled, sub_slot_name,
##     subscription_status
## FROM bdr.subscription_summary;
-[ RECORD 1 ]-------+---------------------------
node_group_name     | bdrgroup
origin_name         | node2
sub_enabled         | t
sub_slot_name       | bdr_postgres_bdrgroup_node1
subscription_status | replicating
-[ RECORD 2 ]-------+---------------------------
node_group_name     | bdrgroup
origin_name         | node3
sub_enabled         | t
sub_slot_name       | bdr_postgres_bdrgroup_node1
subscription_status | replicating
```

Monitoring WAL senders using LCR

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

```
postgres=# SELECT * FROM bdr.wal_sender_stats();
   pid   | is_using_lcr |       decoder_slot_name       |
lcr_file_name
---------+--------------+-------------------------------+-------------------------
-----------------
 2059904 | f            |                               |
 2059909 | t            | bdr_postgres_bdrgroup_decoder |
000000000000000000000014000000000000000
 2059916 | t            | bdr_postgres_bdrgroup_decoder |
000000000000000000000014000000000000000
(3 rows)
```

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

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

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

Monitoring BDR Replication Workers

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

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

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

The view `bdr.worker_errors` shows last error (if any) reported by any worker which has a problem continuing the work. This is persistent information, so it's important to note the time of the error not just the existence of one, because most errors are transient in their nature and BDR workers will retry the failed operation.

Monitoring BDR Writers

There is another system view `bdr.writers` to monitor writer activities. This views shows the current status of only writer workers. It includes:

- `sub_name` to identify the subscription which the writer belongs to
- `pid` of the writer process
- `streaming_allowed` to know if the writer supports application of in-progress streaming transactions
- `is_streaming` to know if the writer is currently applying a streaming transaction
- `commit_queue_position` to check the position of the writer in the commit queue.

BDR honours commit ordering by following the same commit order as happened on the origin. In case of parallel writers, multiple writers could be applying different transactions at the same time. The `commit_queue_position` shows in which order they will commit. Value `0` means that the writer is the first one to commit. Value `-1` means that the commit position is not yet known. This can happen for a streaming transaction or when the writer is not applying any transaction at the moment.

Monitoring Global Locks

The global lock, which is currently only used for DDL replication, is a heavyweight lock that exists across the whole BDR group.

There are currently two types of global locks:

- DDL lock, used for serializing all DDL operations on permanent (not temporary) objects (i.e. tables) in the database

- DML relation lock, used for locking out writes to relations during DDL operations that change the relation definition

Either or both entry types may be created for the same transaction, depending on the type of DDL operation and the value of the `bdr.ddl_locking` setting.

Global locks held on the local node are visible in the `bdr.global_locks` view. This view shows the type of the lock; for relation locks it shows which relation is being locked, the PID holding the lock (if local), and whether the lock has been globally granted or not. In case of global advisory locks, `lock_type` column shows `GLOBAL_LOCK_ADVISORY` and `relation` column shows the advisory key(s) on which the lock is acquired.

The following is an example output of `bdr.global_locks` while running an `ALTER TABLE` statement with `bdr.ddl_locking = on`:

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

See the catalog documentation for details on all fields including lock timing information.

Monitoring Conflicts

Replication conflicts can arise when multiple nodes make changes that affect the same rows in ways that can interact with each other. The BDR system should be monitored to ensure that conflicts are identified and, where possible, application changes are made to eliminate them or make them less frequent.

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

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

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

External Monitoring

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

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

bdr.set_node_location

This function inserts node metadata into `bdr.node_location`

Synopsis

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

Parameters

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

bdr.set_network_path_info

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

Synopsis

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

Parameters

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

bdr.network_monitoring view

This view collects information about the network path between nodes.

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

Apply Statistics

BDR collects statistics about replication apply, both for each subscription and for each table.

Two monitoring views exist: `bdr.stat_subscription` for subscription statistics and `bdr.stat_relation` for relation statistics. These views both provide:

- Number of INSERTs/UPDATEs/DELETEs/TRUNCATEs replicated
- Block accesses and cache hit ratio
- Total I/O time for read/write
- Number of in-progress transactions streamed to file
- Number of in-progress transactions streamed to writers
- Number of in-progress streamed transactions committed/aborted

and for relations only, these statistics:

- Total time spent processing replication for the relation
- Total lock wait time to acquire lock (if any) for the relation (only)

and for subscriptions only, these statistics:

- Number of COMMITs/DDL replicated for the subscription
- Number of times this subscription has connected upstream

Tracking of these statistics is controlled by the BDR GUCs `bdr.track_subscription_apply` and `bdr.track_relation_apply` respectively.

The example output from these would look like this:

```
## SELECT sub_name, nconnect, ninsert, ncommit, nupdate, ndelete, ntruncate, nddl
FROM bdr.stat_subscription;
-[ RECORD 1 ]----------------------------------
sub_name  | bdr_regression_bdrgroup_node1_node2
nconnect  | 3
ninsert   | 10
ncommit   | 5
nupdate   | 0
ndelete   | 0
ntruncate | 0
nddl      | 2
```

In this case the subscription connected 3 times to the upstream, inserted 10 rows and did 2 DDL commands inside 5 transactions.

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

Standard PostgreSQL Statistics Views

Statistics on table and index usage are updated normally by the downstream master. This is essential for the correct function of autovacuum. If there are no local writes on the downstream master and statistics have not been reset, these

two views should show corresponding results between upstream and downstream:

- `pg_stat_user_tables`
- `pg_statio_user_tables`

!!! Note We don't necessarily expect the upstream table statistics to be *similar* to the downstream ones; we only expect them to *change* by the same amounts. Consider the example of a table whose statistics show 1M inserts and 1M updates; when a new node joins the BDR group, the statistics for the same table in the new node will show 1M inserts and zero updates. However, from that moment, the upstream and downstream table statistics will change by the same amounts, because all changes on one side will be replicated to the other side.

Since indexes are used to apply changes, the identifying indexes on the downstream side may appear more heavily used with workloads that perform `UPDATE`s and `DELETE`s than non-identifying indexes are.

The built-in index monitoring views are:

- `pg_stat_user_indexes`
- `pg_statio_user_indexes`

All these views are discussed in detail in the PostgreSQL documentation on the statistics views.

Monitoring BDR Versions

BDR allows running different Postgres versions as well as different BDR versions across the nodes in the same cluster. This is useful for upgrading.

The view `bdr.group_versions_details` uses the function `bdr.run_on_all_nodes()` to retrieve Postgres and BDR versions from all nodes at the same time. For example:

```
bdrdb=# SELECT node_name, postgres_version, bdr_version
        FROM bdr.group_versions_details;
 node_name | postgres_version | bdr_version
-----------+------------------+-------------
 node1     | 14.1             | 4.0.0
 node2     | 14.1             | 4.0.0
```

The recommended setup is to try to have all nodes running the same latest versions as soon as possible. It is recommended that the cluster does not run different BDR versions for too long.

For monitoring purposes, we recommend the following alert levels:

- status=UNKNOWN, message=This node is not part of any BDR group
- status=OK, message=All nodes are running same BDR versions
- status=WARNING, message=There is at least 1 node that is not accessible
- status=WARNING, message=There are node(s) running different BDR versions when compared to other nodes

The described behavior is implemented in the function `bdr.monitor_group_versions()`, which uses BDR version information returned from the view `bdr.group_version_details` to provide a cluster-wide version check. For example:

```
bdrdb=# SELECT * FROM bdr.monitor_group_versions();
 status |                message
--------+---------------------------------------
 OK     | All nodes are running same BDR versions
```

Monitoring Raft Consensus

Raft Consensus should be working cluster-wide at all times. The impact of running a EDB Postgres Distributed cluster without Raft Consensus working might be as follows:

- BDR data changes replication may still be working correctly
- Global DDL/DML locks will not work
- Galloc sequences will eventually run out of chunks
- Eager Replication will not work
- Cluster maintenance operations (join node, part node, promote standby) are still allowed but they might not finish (simply hang)
- Node statuses might not be correctly synced among the BDR nodes
- BDR group replication slot does not advance LSN, thus keeps WAL files on disk

The view `bdr.group_raft_details` uses the functions `bdr.run_on_all_nodes()` and `bdr.get_raft_status()` to retrieve Raft Consensus status from all nodes at the same time. For example:

```
bdrdb=# SELECT node_id, node_name, state, leader_id
FROM bdr.group_raft_details;
  node_id   | node_name |     state      | leader_id
------------+-----------+----------------+------------
 1148549230 | node1     | RAFT_LEADER    | 1148549230
 3367056606 | node2     | RAFT_FOLLOWER | 1148549230
```

We can say that Raft Consensus is working correctly if all below conditions are met:

- A valid state (`RAFT_LEADER` or `RAFT_FOLLOWER`) is defined on all nodes
- Only one of the nodes is the `RAFT_LEADER`
- The `leader_id` is the same on all rows and must match the `node_id` of the row where `state = RAFT_LEADER`

From time to time, Raft Consensus will start a new election to define a new `RAFT_LEADER`. During an election, there might be an intermediary situation where there is no `RAFT_LEADER` and some of the nodes consider themselves as `RAFT_CANDIDATE`. The whole election should not take longer than `bdr.raft_election_timeout` (by default it is set to 6 seconds). If the query above returns an in-election situation, then simply wait for `bdr.raft_election_timeout` and run the query again. If after `bdr.raft_election_timeout` has passed and some the conditions above are still not met, then Raft Consensus is not working.

Raft Consensus might not be working correctly on a single node only; for example one of the nodes does not recognize the current leader and considers itself as a `RAFT_CANDIDATE`. In this case, it is important to make sure that:

- All BDR nodes are accessible to each other through both regular and replication connections (check file `pg_hba.conf`)
- BDR versions are the same on all nodes
- `bdr.raft_election_timeout` is the same on all nodes

In some cases, especially if nodes are geographically distant from each other and/or network latency is high, the default value of `bdr.raft_election_timeout` (6 seconds) might not be enough. If Raft Consensus is still not working even after making sure everything is correct, consider increasing `bdr.raft_election_timeout` to, say, 30 seconds on all nodes. From BDR 3.6.11 onwards, setting `bdr.raft_election_timeout` requires only a server reload.

Given how Raft Consensus affects cluster operational tasks, and also as Raft Consensus is directly responsible for advancing the group slot, we can define monitoring alert levels as follows:

- status=UNKNOWN, message=This node is not part of any BDR group

- status=OK, message=Raft Consensus is working correctly
- status=WARNING, message=There is at least 1 node that is not accessible
- status=WARNING, message=There are node(s) as RAFT_CANDIDATE, an election might be in progress
- status=WARNING, message=There is no RAFT_LEADER, an election might be in progress
- status=CRITICAL, message=There is a single node in Raft Consensus
- status=CRITICAL, message=There are node(s) as RAFT_CANDIDATE while a RAFT_LEADER is defined
- status=CRITICAL, message=There are node(s) following a leader different than the node set as RAFT_LEADER

The described behavior is implemented in the function `bdr.monitor_group_raft()`, which uses Raft Consensus status information returned from the view `bdr.group_raft_details` to provide a cluster-wide Raft check. For example:

```
bdrdb=# SELECT * FROM bdr.monitor_group_raft();
 status |               message
--------+--------------------------------------
 OK     | Raft Consensus is working correctly
```

Monitoring Replication Slots

Each BDR node keeps:

- One replication slot per active BDR peer
- One group replication slot

For example:

```
bdrdb=# SELECT slot_name, database, active, confirmed_flush_lsn
FROM pg_replication_slots ORDER BY slot_name;
        slot_name         | database | active | confirmed_flush_lsn
--------------------------+----------+--------+---------------------
 bdr_bdrdb_bdrgroup       | bdrdb    | f      | 0/3110A08
 bdr_bdrdb_bdrgroup_node2 | bdrdb    | t      | 0/31F4670
 bdr_bdrdb_bdrgroup_node3 | bdrdb    | t      | 0/31F4670
 bdr_bdrdb_bdrgroup_node4 | bdrdb    | t      | 0/31F4670
```

Peer slot names follow the convention `bdr_<DATABASE>_<GROUP>_<PEER>`, while the BDR group slot name follows the convention `bdr_<DATABASE>_<GROUP>`, which can be accessed using the function `bdr.local_group_slot_name()`.

Peer replication slots should be active on all nodes at all times. If a peer replication slot is not active, then it might mean:

- The corresponding peer is shutdown or not accessible; or
- BDR replication is broken.

Grep the log file for `ERROR` or `FATAL` and also check `bdr.worker_errors` on all nodes. The root cause might be, for example, an incompatible DDL was executed with DDL replication disabled on one of the nodes.

The BDR group replication slot is however inactive most of the time. BDR maintains this slot and advances its LSN when all other peers have already consumed the corresponding transactions. Consequently it is not necessary to monitor the status of the group slot.

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

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

One of the following status summaries will be returned:

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

Monitoring Transaction COMMITs

By default, BDR transactions commit only on the local node. In that case, transaction `COMMIT` will be processed quickly.

BDR can be used with standard PostgreSQL synchronous replication, while BDR also provides two new transaction commit modes: CAMO and Eager replication. Each of these modes provides additional robustness features, though at the expense of additional latency at `COMMIT`. The additional time at `COMMIT` can be monitored dynamically using the `bdr.stat_activity` catalog, where processes report different `wait_event` states. A transaction in `COMMIT` waiting for confirmations from one or more synchronous standbys reports a `SyncRep` wait event, whereas the two new modes report `EagerRep`.

---

16 EDB Postgres Distributed Command Line Interface

The EDB Postgres Distributed Command Line Interface (PGD CLI) is a tool to manage your EDB Postgres Distributed cluster. It allows you to run commands against EDB Postgres Distributed clusters.

See the Command reference for the available commands to inspect, manage, and get information on cluster resources.

See Installing PGD CLI for information about how TPAexec deploys PGD CLI, how to install PGD CLI on a standalone server manually, and specifying connection strings.

Requirements

The PGD CLI requires postgres superuser privileges to run.

Using the PGD CLI

`pgd` is the command name for the PGD command line interface. See pgd in the Command reference for a description of the command options. See the following sections for sample use cases.

Specifying a configuration file

If you rename the file or move it to another location, specify the new name and location using the optional `-f` or `--config-file` flag. For example:

```
  pgd show-nodes -f /opt/my-config.yml
```

Passing a database connection string

Use the `--dsn` flag to pass a database connection string directly to a command. You don't need a configuration file if you pass the connection string with this flag. The flag takes precedence if a configuration file is present. For example:

```
pgd show-nodes --dsn "host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
```

Specifying the output format

The PGD CLI supports the following output formats:

Format	Considerations
tabular	Default format. Presents the data in tabular form.
json	Presents the raw data with no formatting. For some commands, the json output may show more data than shown in the tabular output such as extra fields and more detailed messages.
yaml	Same as json except field order is alphabetical. Experimental and may not be fully supported in future versions.

Use the `-o` or `--output` flag to change the default output format to json or yaml. For example:

```
pgd show-nodes -o json
```

Accessing the command line help

To list the supported commands, enter:

```
pgd help
```

For help for a specific command and its parameters, enter `pgd help <command_name>`. For example:

```
pgd help show-nodes
```

16.1 Installing PGD CLI

TPAexec installs and configures PGD CLI on each BDR node, by default. If you wish to install PGD CLI on any non-BDR instance in the cluster, you simply attach the pgdcli role to that instance in TPAexec's configuration file before

deploying. See TPAexec for more information.

Installing manually

You can manually install the PGD CLI on any Linux machine using `.deb` and `.rpm` packages available from the BDR repository. The package name is `edb-pgd-cli`. For example:

```
## for Debian
sudo apt-get install edb-pgd-cli
```

When the PGD CLI is configured by TPAexec, it connects automatically, but with a manual installation to a standalone EDB Postgres Distributed cluster you need to provide a connection string.

Specifying database connection strings

You can either use a configuration file to specify the database connection strings for your cluster (see following section) or pass the connection string directly to a command (see the sample use case).

Using a configuration file

Use the `pgd-config.yml` configuration file to specify the database connection string for your cluster. The configuration file should contain the database connection string for at least one BDR node in the cluster. The cluster name is optional and not validated.

For example:

```
cluster:
  name: cluster-name
  endpoints:
  - "host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
  - "host=bdr-b1 port=5432 dbname=bdrdb user=postgres "
  - "host=bdr-c1 port=5432 dbname=bdrdb user=postgres "
```

The `pgd-config.yml`, is located in the `/etc/edb` directory, by default. The PGD CLI searches for `pgd-config.yml` in the following locations (precedence order - higher to lower):

1. `/etc/edb` (default)
2. `$HOME/.edb`
3. `.` (working directory)

If you rename the file or move it to another location, specify the new name and location using the optional `-f` or `--config-file` flag when entering a command. See the sample use case.

16.2 Command reference

pgd is the command name for the PGD command line interface.

Synopsis

The EDB Postgres Distributed Command Line Interface (PGD CLI) is a tool to manage your EDB Postgres Distributed cluster. It allows you to run commands against EDB Postgres Distributed clusters. You can use it to inspect and manage cluster resources.

Options

```
  -f, --config-file string   config file; ignored if
                             --dsn flag is present (default "/etc/edb/pgd-
config.yml")
      --dsn string           database connection string
                             e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres
"
  -h, --help                 help for pgd
  -L, --log-level string     logging level: debug, info, warn, error (default
"error")
  -o, --output string        output format: json, yaml
```

See also

- check-health
- show-camo
- show-clockscrew
- show-events
- show-nodes
- show-raft
- show-replslots
- show-subscriptions
- show-version

16.2.1 check-health

Checks the health of the EDB Postgres Distributed cluster.

Synopsis

Performs various checks such as if all nodes are accessible, all replication slots are working, and CAMO pairs are connected.

Please note that the current implementation of clock skew may return an inaccurate skew value if the cluster is under high load while running this command or has large number of nodes in it.

```
pgd check-health [flags]
```

Examples

```
Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down; bdr-a1
and bdr-b1 are CAMO partners)

$ pgd check-health

Check      Status   Message
-----      ------   -------
CAMO       Critical At least 1 CAMO partner is not connected
ClockSkew  Critical Clockskew cannot be determined for at least 1 BDR node pair
Connection Critical The node bdr-b1 is not accessible
Raft       Warning  There is no RAFT_LEADER, an election might be in progress
Replslots  Critical There is at least 1 BDR replication slot which is inactive
Version    Warning  There is at least 1 node that is not accessible


Example 2 (3 node cluster, all nodes are up but system clocks are not in sync)

$ pgd check-health

Check      Status  Message
-----      ------  -------
CAMO       Ok      All CAMO pairs are connected
ClockSkew  Warning At least 1 BDR node pair has clockskew greater than 2 seconds
Connection Ok      All BDR nodes are accessible
Raft       Ok      Raft Consensus is working correctly
Replslots  Ok      All BDR replication slots are working correctly
Version    Ok      All nodes are running same BDR versions


Example 3 (3 node cluster, all nodes are up and all checks are Ok)

$ pgd check-health

Check      Status Message
-----      ------ -------
CAMO       Ok     All CAMO pairs are connected
ClockSkew  Ok     All BDR node pairs have clockskew within permissible limit
Connection Ok     All BDR nodes are accessible
Raft       Ok     Raft Consensus is working correctly
Replslots  Ok     All BDR replication slots are working correctly
Version    Ok     All nodes are running same BDR versions
```

Options

```
  -h, --help   help for check-health
```

Options inherited from parent commands

```
  -f, --config-file string   config file; ignored if
                             --dsn flag is present (default "/etc/edb/pgd-
```

```
config.yml")
      --dsn string           database connection string
                             e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres
"
  -L, --log-level string     logging level: debug, info, warn, error (default
"error")
  -o, --output string        output format: json, yaml
```

16.2.2 show-camo

Shows BDR CAMO (Commit at Most Once) details.

Synopsis

Shows BDR CAMO (Commit at Most Once) details such as the name of the CAMO partner, connection and readiness status, any pending and unresolved CAMO transactions, and differences between apply_lsn and receive_lsn. This command is available only for EDB Postgres Extended Server and EDB Postgres Advanced Server (v14 and later).

```
pgd show-camo [flags]
```

Examples

```
  Example 1 (3 node cluster, bdr-a1 and bdr-b1 are CAMO partner but bdr-b1 is down)

  $ pgd show-camo

  Node   CAMO Partner Connected Ready Transactions Resolved Apply LSN  Receive LSN
Apply Queue Size
  ----   ------------ --------- ----- -------------------- ---------  -----------
----------------
  bdr-a1 bdr-b1       false     false true                 0/E42C99B0 0/E42C99B0
0


  Example 2 (3 node cluster, bdr-b1 was down and it has just been restarted)

  $ pgd show-camo

  Node   CAMO Partner Connected Ready Transactions Resolved Apply LSN  Receive LSN
Apply Queue Size
  ----   ------------ --------- ----- -------------------- ---------  -----------
----------------
  bdr-a1 bdr-b1       true      true  true                 0/E533DAB8 0/E533DAB8
0
  bdr-b1 bdr-a1       true      false true                 3/7AE81A28 3/7AE81A28
0
```

```
  Example 3 (3 node cluster, all nodes are up and in 'streaming' state)

  $ pgd show-camo

  Node   CAMO Partner Connected Ready Transactions Resolved Apply LSN  Receive LSN
Apply Queue Size
  ----   ------------ --------- ----- -------------------- ---------  -----------
----------------
  bdr-a1 bdr-b1       true      true  true                 0/E56AE520 0/E56AE520
0
  bdr-b1 bdr-a1       true      true  true                 3/7B180BA8 3/7B180BA8
0
```

Options

```
  -h, --help   help for show-camo
```

Options inherited from parent commands

```
  -f, --config-file string   config file; ignored if
                             --dsn flag is present (default "/etc/edb/pgd-
config.yml")
      --dsn string           database connection string
                             e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres
"
  -L, --log-level string     logging level: debug, info, warn, error (default
"error")
  -o, --output string        output format: json, yaml
```

16.2.3 show-clockskew

Shows the status of clock skew between each BDR node pair.

Synopsis

Shows the status of clock skew between each BDR node pair in the cluster.

Please note that the current implementation of clock skew may return an inaccurate skew value if the cluster is under high load while running this command or has large number of nodes in it.

```
Symbol      Meaning
-------     --------
*           ok
~           warning (skew > 2 seconds)
!           critical (skew > 5 seconds)
x           down / unreachable
```

```
?           unknown
-           n/a
```

```
pgd show-clockskew [flags]
```

Examples

```
  Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

  $ pgd show-clockskew

  Node   bdr-a1 bdr-b1 bdr-c1 Current Time
  ----   ------ ------ ------ ------------
  bdr-a1 *      ?      *      2022-03-30 07:02:21.334472
  bdr-b1 x      *      x      x
  bdr-c1 *      ?      *      2022-03-30 07:02:21.186809


  Example 2 (3 node cluster, all nodes are up)

  $ pgd show-clockskew

  Node   bdr-a1 bdr-b1 bdr-c1 Current Time
  ----   ------ ------ ------ ------------
  bdr-a1 *      *      *      2022-03-30 07:04:54.147017
  bdr-b1 *      *      *      2022-03-30 07:04:54.340543
  bdr-c1 *      *      *      2022-03-30 07:04:53.90451
```

Options

```
  -h, --help   help for show-clockskew
```

Options inherited from parent commands

```
  -f, --config-file string   config file; ignored if
                             --dsn flag is present (default "/etc/edb/pgd-
config.yml")
      --dsn string           database connection string
                             e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres
"
  -L, --log-level string     logging level: debug, info, warn, error (default
"error")
  -o, --output string        output format: json, yaml
```

16.2.4 show-events

Shows events such as background worker errors and node membership changes.

Synopsis

Shows events such as background worker errors and node membership changes. Output is sorted by Time column in descending order. Message column is truncated after a few lines. To view complete message use json output format ('-o json').

For more details on each node state, see show-nodes command help ('pgd show-nodes -h').

```
pgd show-events [flags]
```

Examples

```
  Example 1 (3 node cluster)

  $ pgd show-events --lines 10

  Time                          Observer Node Subject Node Type
Message
  ----                          ------------- ------------ ----                 -
------
  2022-04-19 19:45:43.077712+00 bdr-b1        bdr-c1       receiver worker error
pglogical worker received fast finish request, exiting
  2022-04-19 19:45:43.066804+00 bdr-c1        bdr-a1       receiver worker error
pglogical worker received fast finish request, exiting
  2022-04-19 19:45:43.057598+00 bdr-b1        bdr-a1       receiver worker error
pglogical worker received fast finish request, exiting
  2022-04-19 19:45:43.046515+00 bdr-c1        bdr-b1       receiver worker error
pglogical worker received fast finish request, exiting
  2022-04-19 19:45:43.033369+00 bdr-a1        bdr-c1       receiver worker error
pglogical worker received fast finish request, exiting
  2022-04-19 19:45:43.013203+00 bdr-a1        bdr-b1       receiver worker error
pglogical worker received fast finish request, exiting
  2022-04-19 19:45:40.024662+00 bdr-c1        bdr-c1       node state change
ACTIVE
  2022-04-19 19:45:40.024575+00 bdr-b1        bdr-c1       node state change
ACTIVE
  2022-04-19 19:45:40.022788+00 bdr-a1        bdr-c1       node state change
ACTIVE
  2022-04-19 19:45:38.961424+00 bdr-c1        bdr-c1       node state change
PROMOTING
```

Options

```
  -h, --help        help for show-events
  -n, --lines int   show top n lines
```

Options inherited from parent commands

```
  -f, --config-file string   config file; ignored if
                             --dsn flag is present (default "/etc/edb/pgd-
config.yml")
      --dsn string           database connection string
                             e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres
"
  -L, --log-level string     logging level: debug, info, warn, error (default
"error")
  -o, --output string        output format: json, yaml
```

16.2.5 show-nodes

Shows all nodes in the EDB Postgres Distributed cluster and their summary.

Synopsis

Shows all nodes in the EDB Postgres Distributed cluster and their summary including name, node id, group, and current/target state.

Node States

- NONE: Node state is unset when the worker starts, expected to be set quickly to the current known state.
- CREATED: bdr.create_node() has been executed, but the node is not a member of any EDB Postgres Distributed cluster yet.
- JOIN_START: bdr.join_node_group() begins to join the local node to an existing EDB Postgres Distributed cluster.
- JOINING: The node join has started and is currently at the initial sync phase, creating the schema and data on the node.
- CATCHUP: Initial sync phase is completed; now the join is at the last step of retrieving and applying transactions that were performed on the upstream peer node since the join started.
- STANDBY: Node join has finished, but not yet started to broadcast changes. All joins spend some time in this state, but if defined as a Logical Standby, the node will continue in this state.
- PROMOTE: Node was a logical standby and we just called bdr.promote_node to move the node state to ACTIVE. These two PROMOTE states have to be coherent to the fact, that only one node can be with a state higher than STANDBY but lower than ACTIVE.
- PROMOTING: Promotion from logical standby to full BDR node is in progress.
- ACTIVE: The node is a full BDR node and is currently ACTIVE. This is the most common node status.
- PART_START: Node was ACTIVE or STANDBY and we just called bdr.part_node to remove the node from the EDB Postgres Distributed cluster.
- PARTING: Node disconnects from other nodes and plays no further part in consensus or replication.
- PART_CATCHUP: Non-parting nodes synchronize any missing data from the recently parted node.
- PARTED: Node parting operation is now complete on all nodes.

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

Note that the read-only state of a node, as shown in the Current State or in the Target State columns, is indicated as STANDBY.

```
pgd show-nodes [flags]
```

Examples

```
  Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

  $ pgd show-nodes

  Node   Node ID    Node Group Current State Target State Status      Seq ID
  ----   -------    ---------- ------------- ------------ ------      ------
  bdr-a1 3136956818 bdrgroup   ACTIVE        ACTIVE       Up          1
  bdr-b1 2380210996 bdrgroup   ACTIVE        ACTIVE       Unreachable 2
  bdr-c1 1804769977 bdrgroup   ACTIVE        ACTIVE       Up          3


  Example 2 (3 node cluster, all nodes are up)

  $ pgd show-nodes

  Node   Node ID    Node Group Current State Target State Status Seq ID
  ----   -------    ---------- ------------- ------------ ------ ------
  bdr-a1 3136956818 bdrgroup   ACTIVE        ACTIVE       Up     1
  bdr-b1 2380210996 bdrgroup   ACTIVE        ACTIVE       Up     2
  bdr-c1 1804769977 bdrgroup   ACTIVE        ACTIVE       Up     3


  Example 3 (cluster with witness, logical standby and subscriber-only nodes)
  Note: In contrast to logical standby, the subscriber-only nodes are fully
  joined node to the cluster

  $ pgd show-nodes

  Node               Node ID    Node Group      Current State Target State Status
Seq ID
  ----               -------    ----------      ------------- ------------ ------
------
  bdr-a1             3136956818 bdrgroup        ACTIVE        ACTIVE       Up     1
  bdr-b1             2380210996 bdrgroup        ACTIVE        ACTIVE       Up     2
  witness-c1         1450365472 bdrgroup        ACTIVE        ACTIVE       Up     3
  logical-standby-a1 1140256918 bdrgroup        STANDBY       STANDBY      Up     4
  logical-standby-b1 3541792022 bdrgroup        STANDBY       STANDBY      Up     5
  subscriber-only-c1 2448841809 subscriber-only ACTIVE        ACTIVE       Up     6
```

Options

```
  -h, --help   help for show-nodes
```

Options inherited from parent commands

```
  -f, --config-file string   config file; ignored if
                             --dsn flag is present (default "/etc/edb/pgd-
config.yml")
      --dsn string           database connection string
                             e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres
```

```
"
  -L, --log-level string     logging level: debug, info, warn, error (default
"error")
  -o, --output string        output format: json, yaml
```

16.2.6 show-raft

Shows BDR Raft (consensus protocol) details.

Synopsis

Shows BDR Raft (consensus protocol) details such as Raft state (leader, follower), Raft election id, and number of voting nodes.

Note: In some cases such as network partition, output may vary based on the node to which the CLI is connected.

```
pgd show-raft [flags]
```

Examples

```
  Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

  $ pgd show-raft

  Node   Raft State    Raft Term Commit Index Nodes Voting Nodes Protocol Version
  ----   ----------    --------- ------------ ----- ------------ ----------------
  bdr-c1 RAFT_FOLLOWER 29        6081272      3     3            4002
  bdr-a1 RAFT_LEADER   29        6081272      3     3            4002
  bdr-b1


  Example 2 (3 node cluster, all nodes are up)

  $ pgd show-raft

  Node   Raft State    Raft Term Commit Index Nodes Voting Nodes Protocol Version
  ----   ----------    --------- ------------ ----- ------------ ----------------
  bdr-c1 RAFT_FOLLOWER 38        6132327      3     3            4002
  bdr-a1 RAFT_LEADER   38        6132331      3     3            4002
  bdr-b1 RAFT_FOLLOWER 38        6132336      3     3            4002


  Example 3 (3 node cluster, all nodes are up but bdr-a1 is not able to connect
  to other nodes; following is the output when cli is connected to bdr-a1)

  $ pgd show-raft
```

```
  Node   Raft State    Raft Term Commit Index Nodes Voting Nodes Protocol Version
  ----   ----------    --------- ------------ ----- ------------ ----------------
  bdr-c1
  bdr-a1 RAFT_FOLLOWER 40        6176769      3     3            4002
  bdr-b1


  Example 4 (cluster with witness, logical standby and subscriber-only nodes)
  Note: Unlike full-bdr (or witness node), logical standby and subscriber-only
  nodes don't have raft voting rights.

  $ pgd show-raft

  Node               Raft State    Raft Term Commit Index Nodes Voting Nodes
Protocol Version
  ----               ----------    --------- ------------ ----- ------------ -----
-----------
  bdr-a1             RAFT_LEADER   0         10268        6     3            4003
  bdr-b1             RAFT_FOLLOWER 0         10279        6     3            4003
  witness-c1         RAFT_FOLLOWER 0         10281        6     3            4003
  logical-standby-a1 RAFT_FOLLOWER 0         10281        6     3            4003
  logical-standby-b1 RAFT_FOLLOWER 0         10281        6     3            4003
  subscriber-only-c1 RAFT_FOLLOWER 0         10281        6     3            4003
```

Options

```
  -h, --help   help for show-raft
```

Options inherited from parent commands

```
  -f, --config-file string   config file; ignored if
                             --dsn flag is present (default "/etc/edb/pgd-
config.yml")
      --dsn string           database connection string
                             e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres
"
  -L, --log-level string     logging level: debug, info, warn, error (default
"error")
  -o, --output string        output format: json, yaml
```

16.2.7 show-replslots

Shows the status of BDR replication slots.

Synopsis

Shows the status of BDR replication slots. Output with the verbose flag gives details such as is slot active, replication

state (disconnected, streaming, catchup), and approximate lag.

```
Symbol      Meaning
-------     --------
*           ok
~           warning (lag > 10M)
!           critical (lag > 100M OR slot is 'inactive' OR 'disconnected')
x           down / unreachable
-           n/a

In matrix view, sometimes byte lag is shown in parentheses. It is a
maxOf(WriteLag, FlushLag, ReplayLag, SentLag).
```

```
pgd show-replslots [flags]
```

Examples

```
  Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

  $ pgd show-replslots

  Node   bdr-a1 bdr-b1  bdr-c1
  ----   ------ ------  ------
  bdr-a1 *      !(6.6G) *
  bdr-b1 x      *       x
  bdr-c1 *      !(6.9G) *


  $ pgd show-replslots --verbose

  Origin Node Target Node Status (active/state) Write Lag (bytes/duration)    Flush
Lag (bytes/duration)    Replay Lag (bytes/duration)   Sent Lag (bytes)
  ----------- ----------- --------------------- -------------------------    ----
----------------------    --------------------------   ----------------
  bdr-a1      bdr-b1      f / disconnected      6.6G / 8 days 02:58:36.243723 6.6G
/ 8 days 02:58:36.243723 6.6G / 8 days 02:58:36.243723 6.6G
  bdr-a1      bdr-c1      t / streaming         0B / 00:00:00                 0B /
00:00:00                 0B / 00:00:00                 0B
  bdr-c1      bdr-a1      t / streaming         0B / 00:00:00.000812          0B /
00:00:00.000812          0B / 00:00:00.000812          0B
  bdr-c1      bdr-b1      f / disconnected      6.9G / 8 days 02:58:36.004415 6.9G
/ 8 days 02:58:36.004415 6.9G / 8 days 02:58:36.004415 6.9G


  Example 2 (3 node cluster, bdr-b1 was down and it has just been restarted)

  $ pgd show-replslots

  Node   bdr-a1 bdr-b1  bdr-c1
  ----   ------ ------  ------
  bdr-a1 *      !(6.9G) *
  bdr-b1 *      *       *
  bdr-c1 *      !(5.8G) *
```

```
  $ pgd show-replslots --verbose

  Origin Node Target Node Status (active/state) Write Lag (bytes/duration) Flush
Lag (bytes/duration) Replay Lag (bytes/duration) Sent Lag (bytes)
  ----------- ----------- --------------------- -------------------------- -------
------------------ -------------------------- ----------------
  bdr-a1      bdr-b1      t / catchup           6.9G / 00:00:00.000778     6.9G /
00:00:00.000778     6.9G / 00:00:00.000778      6.9G
  bdr-a1      bdr-c1      t / streaming         0B / 00:00:00.104121       0B /
00:00:00.104133       0B / 00:00:00.104133        0B
  bdr-b1      bdr-a1      t / streaming         0B / 00:00:00              0B /
00:00:00              0B / 00:00:00                0B
  bdr-b1      bdr-c1      t / streaming         0B / 00:00:00              0B /
00:00:00              0B / 00:00:00                0B
  bdr-c1      bdr-a1      t / streaming         6.8K / 00:00:00            6.8K /
00:00:00            6.8K / 00:00:00              6.8K
  bdr-c1      bdr-b1      t / catchup           5.5G / 00:00:00.008257     5.5G /
00:00:00.008257     5.5G / 00:00:00.008257      5.5G


  Example 3 (3 node cluster, all nodes are up and in 'streaming' state)

  $ pgd show-replslots

  Node   bdr-a1 bdr-b1 bdr-c1
  ----   ------ ------ ------
  bdr-a1 *      *      *
  bdr-b1 *      *      *
  bdr-c1 *      *      *


  $ pgd show-replslots --verbose

  Origin Node Target Node Status (active/state) Write Lag (bytes/duration) Flush
Lag (bytes/duration) Replay Lag (bytes/duration) Sent Lag (bytes)
  ----------- ----------- --------------------- -------------------------- -------
------------------ -------------------------- ----------------
  bdr-a1      bdr-b1      t / streaming         0B / 00:00:00              0B /
00:00:00              0B / 00:00:00                0B
  bdr-a1      bdr-c1      t / streaming         0B / 00:00:00              0B /
00:00:00              0B / 00:00:00                0B
  bdr-b1      bdr-a1      t / streaming         0B / 00:00:00              0B /
00:00:00              0B / 00:00:00                0B
  bdr-b1      bdr-c1      t / streaming         0B / 00:00:00              0B /
00:00:00              0B / 00:00:00                0B
  bdr-c1      bdr-a1      t / streaming         0B / 00:00:00              528B /
00:00:00            528B / 00:00:00              0B
  bdr-c1      bdr-b1      t / streaming         528B / 00:00:00            528B /
00:00:00            528B / 00:00:00              0B


  Example 4 (cluster with witness, logical standby and subscriber-only nodes;
  upstream for logical-standby-a1 is bdr-a1 and for logical-standby-b1 it is bdr-b1)
```

```
  Note:
  1. A logical standby is sent data only by one source node, but no other
  nodes receive replication changes from it
  2. Subscriber-only node subscribes to replication changes from other nodes
  in the cluster, but no other nodes receive replication changes from it

  $ pgd show-replslots

  Node               bdr-a1 bdr-b1 logical-standby-a1 logical-standby-b1
subscriber-only-c1 witness-c1
  ----               ------ ------ ------------------ ------------------ ---------
--------- ----------
  bdr-a1             *      *      *                  -                  *
*
  bdr-b1             *      *      -                  *                  *
*
  logical-standby-a1 -      -      *                  -                  -
-
  logical-standby-b1 -      -      -                  *                  -
-
  subscriber-only-c1 -      -      -                  -                  *
-
  witness-c1         *      *      -                  -                  *
*
```

Options

```
   -h, --help      help for show-replslots
   -v, --verbose   verbose output (default true)
```

Options inherited from parent commands

```
   -f, --config-file string   config file; ignored if
                              --dsn flag is present (default "/etc/edb/pgd-
config.yml")
       --dsn string           database connection string
                              e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres
"
   -L, --log-level string     logging level: debug, info, warn, error (default
"error")
   -o, --output string        output format: json, yaml
```

16.2.8 show-subscriptions

Shows BDR subscription (incoming replication) details.

Synopsis

Shows BDR subscription (incoming replication) details such as origin/target node, timestamp of the last replayed transaction, and lag between now and the timestamp of the last replayed transaction.

```
pgd show-subscriptions [flags]
```

Examples

```
Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

$ pgd show-subscriptions

Origin Node Target Node Last Transaction Replayed At  Lag Duration (seconds)
----------- ----------- ----------------------------  ----------------------
bdr-a1      bdr-c1      2022-04-23 13:13:40.854433+00 0.514275
bdr-b1      bdr-a1
bdr-b1      bdr-c1
bdr-c1      bdr-a1      2022-04-23 13:13:40.852233+00 0.335464


Example 2 (3 node cluster, bdr-b1 was down and it has just been restarted)

$ pgd show-subscriptions

Origin Node Target Node Last Transaction Replayed At  Lag Duration (seconds)
----------- ----------- ----------------------------  ----------------------
bdr-a1      bdr-b1      2022-04-23 13:14:45.669254+00 0.001686
bdr-a1      bdr-c1      2022-04-23 13:14:46.157913+00 -0.002009
bdr-b1      bdr-a1
bdr-b1      bdr-c1
bdr-c1      bdr-a1      2022-04-23 13:14:45.698472+00 0.259521
bdr-c1      bdr-b1      2022-04-23 13:14:45.667979+00 0.002961


Example 3 (3 node cluster, all nodes are up and in 'streaming' state)

$ pgd show-subscriptions

Origin Node Target Node Last Transaction Replayed At  Lag Duration (seconds)
----------- ----------- ----------------------------  ----------------------
bdr-a1      bdr-b1      2022-04-23 13:15:39.732375+00 0.034462
bdr-a1      bdr-c1      2022-04-23 13:15:40.179618+00 0.002647
bdr-b1      bdr-a1      2022-04-23 13:15:39.719994+00 0.305814
bdr-b1      bdr-c1      2022-04-23 13:15:40.180886+00 0.001379
bdr-c1      bdr-a1      2022-04-23 13:15:39.714397+00 0.311411
bdr-c1      bdr-b1      2022-04-23 13:15:39.714397+00 0.052440


Example 4 (cluster with witness, logical standby and subscriber-only nodes;
upstream for logical-standby-a1 is bdr-a1 and for logical-standby-b1 it is bdr-b1)
Note: Logical standby and subscriber-only nodes receive changes but do not
send changes made locally to other nodes

$ pgd show-subscriptions
```

```
  Origin Node Target Node        Last Transaction Replayed At  Lag Duration
(seconds)
  ----------- -----------        ----------------------------  -------------------
---
  bdr-a1      bdr-b1             2022-04-23 13:40:49.106411+00 0.853665
  bdr-a1      logical-standby-a1 2022-04-23 13:40:50.72036+00  0.138430
  bdr-a1      logical-standby-b1
  bdr-a1      subscriber-only-c1 2022-04-23 13:40:50.72036+00  0.016226
  bdr-a1      witness-c1         2022-04-23 13:40:50.470142+00 0.001514
  bdr-b1      bdr-a1             2022-04-23 13:40:49.10174+00  1.095422
  bdr-b1      logical-standby-a1
  bdr-b1      logical-standby-b1 2022-04-23 13:40:50.713666+00 0.271213
  bdr-b1      subscriber-only-c1 2022-04-23 13:40:50.713666+00 0.022920
  bdr-b1      witness-c1         2022-04-23 13:40:50.471789+00 -0.000133
  witness-c1  bdr-a1             2022-04-23 13:40:49.107706+00 1.089456
  witness-c1  bdr-b1             2022-04-23 13:40:49.107706+00 0.852370
  witness-c1  logical-standby-a1
  witness-c1  logical-standby-b1
  witness-c1  subscriber-only-c1 2022-04-23 13:40:50.719844+00 0.016742
```

Options

```
  -h, --help   help for show-subscriptions
```

Options inherited from parent commands

```
  -f, --config-file string   config file; ignored if
                             --dsn flag is present (default "/etc/edb/pgd-
config.yml")
      --dsn string           database connection string
                             e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres
"
  -L, --log-level string     logging level: debug, info, warn, error (default
"error")
  -o, --output string        output format: json, yaml
```

16.2.9 show-version

Shows the version of BDR and Postgres installed on each node.

Synopsis

Shows the version of BDR and Postgres installed on each node in the cluster.

```
pgd show-version [flags]
```

Examples

```
  Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

  $ pgd show-version

  Node   BDR Version Postgres Version
  ----   ----------- ----------------
  bdr-c1 4.1.0       14.2 (EDB Postgres Extended Server 14.2.0) (Debian
2:14.2.0edbpge-1.buster+1)
  bdr-a1 4.1.0       14.2 (EDB Postgres Extended Server 14.2.0) (Debian
2:14.2.0edbpge-1.buster+1)
  bdr-b1


  Example 2 (3 node cluster, all nodes are up)

  $ pgd show-version

  Node   BDR Version Postgres Version
  ----   ----------- ----------------
  bdr-c1 4.1.0       14.2 (EDB Postgres Extended Server 14.2.0) (Debian
2:14.2.0edbpge-1.buster+1)
  bdr-a1 4.1.0       14.2 (EDB Postgres Extended Server 14.2.0) (Debian
2:14.2.0edbpge-1.buster+1)
  bdr-b1 4.1.0       14.2 (EDB Postgres Extended Server 14.2.0) (Debian
2:14.2.0edbpge-1.buster+1)
```

Options

```
  -h, --help   help for show-version
```

Options inherited from parent commands

```
  -f, --config-file string   config file; ignored if
                             --dsn flag is present (default "/etc/edb/pgd-
config.yml")
      --dsn string           database connection string
                             e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres
"
  -L, --log-level string     logging level: debug, info, warn, error (default
"error")
  -o, --output string        output format: json, yaml
```